5 Minute Bedtime Stories

For Android Devs

Disclaimer

This book is an independent work and is not affiliated with, endorsed by, or sponsored by Google, Android, AWS, Jenkins, Jetbrains or any other company or organization mentioned.

All product names, trademarks, and registered trademarks are the property of their respective owners.

Some stories are inspired by real events, though details may have been altered for storytelling purposes.

The Android robot (“Bugdroid”) is reproduced or modified from work created and shared by Google and used according to terms described in the Creative Commons 3.0 Attribution License.

Android is a trademark of Google LLC.

fun code() is an independent publication by Marc Reichelt and contributors.

Kodee is used under CC BY 4.0 (© JetBrains s.r.o.).

About This Book

This book began, like many good ideas, late at night.

After long days at conferences, meetups, and talks, a group of developers would often find themselves gathered together, sharing stories. The kind of stories that only make sense if you've lived through them. Stories about bugs, strange systems, questionable decisions… and the occasional disaster.

More often than not, many of those stories were about the disasters Ash Davies brought upon himself.

At some point, it became a running joke that these stories deserved to be written down. Naturally, the only reasonable next step was to actually do it. So here we are.

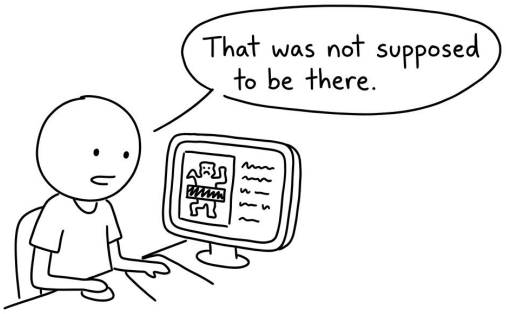
This is a collection of short bedtime stories for developers. Some are funny. Some are slightly unbelievable. Some are… suspiciously specific.

Not every story in this book is guaranteed to be entirely true. But as the Irish say:

"Never let the truth get in the way of a good story."

To protect the innocent (and the guilty), the authors are not matched to their stories. You are free to guess who wrote what.

Enjoy the stories, and hopefully get a good night's sleep.



A Few Kinks in the Code...

“I had been working the whole day before I realised there was a massive picture of an arsehole on my second screen”.

There are few sentences that will ever quite sum up the experiences I had at the most fascinating jobs I have ever (and will ever) work at, but my colleague’s remark that day came close. Perhaps also the discussion I had with our senior product manager: a well-spoken married woman in her 50s, on the topic of our primary USP being “wanking” (on the grounds that to see a user’s naughty photos you had to pay for premium membership). But actually no, there’s a much better story, and if you’ll allow me, I shall give you a small glimpse of the time I worked for a gay fetish hookup app.

I had originally started as a .NET full-stack developer, but I was determined to get into mobile. Why? It ticked all my boxes - I loved building UI, but I also liked the complexities

of backend, and mobile offers plenty of abstract complexity. But first and foremost because until smartphones became ubiquitous, the only time you could see someone using something you built is if you were stood behind them as they sat at their computer. Now you could spot someone using it in the wild - at a bar, on the train, at Christmas with your family (well, maybe not this app… or maybe… who am I to judge?). That was cool.

Working for an adult app had its challenges. The puritanism of the mobile app stores being a huge one. It always amused me how the grubby tech execs of Sillicon Valley could waltz into their offices after a night at the local strip joint and fire out missives about how vital it was to spare people from the evils of pornography and sex. But what this meant is that we were always one slight fuck-up from not just a slap on the wrist, but a full suspension. Back then, the Play Store was the worst for stuff like this - you didn't get a warning, you just got booted. You would be told in the vaguest way what rule you had broken (e.g. adult material) but not how or why so that you could analyse and rectify something you might (or might not have) done wrong. As is often the way with big tech, an entire business and the livelihoods of its employees could find themselves with nothing overnight on the whim of a single powerful individual.

So in my spare time I set about thwarting the stores. I got to working on a means in the app by which it could download a new update and then attempt to install itself. And… it worked! The proof of concept anyway. Lots of companies that find themselves on the wrong side of store policies for whatever reason have taken this approach in the years since - known as side-loading. It was just as well, because we had barely completed the work on it before we found ourselves suspended.

But that's not the real story I wanted to tell.

I'd like to tell you a story about a chap called Bob (name anonymised).

Every year our company organised London's Fetish Week - a whole plethora of events for every potential taste you might have, although primarily based upon the common gear-based kinks found in the gay male community. A number were a little too hardcore for yours truly, but one event I did like going along to was the Fetish Week Dinner: a large restaurant under the arches in Vauxhall would be booked and attendees would all enjoy a 3-course-dinner in their choice of attire (fetish or otherwise) whilst being treated to entertainment ranging from someone being roped up and lifted off the floor to a chap being completely smothered in cake. Different strokes for different folks, as the saying goes.

On this particular day I'd had the shittiest of weeks. I was behind on a key release for the company and I was kicking myself for the disastrous route I'd taken with the codebase. I sulk-walked into the restaurant with my partner and we found a table with some colleagues: my manager, his brother, and one of our wonderfully extravagant events/marketing chaps. We were then joined by two "members", as we called them: users of the app. One was dressed in skimpy gear with a standout pair of rabbit ears protruding from his head, and then there was Bob: a man in his 80s wearing little but an open armless leather waistcoat and a pair of undergarments that left very little to the imagination. I was sat next to Bob.

I love the Fetish Week Dinner because I get to speak to people who use the app I build. My first question to Bob was to ask what platform he accessed the app on. I asked if it was Android.

"No. I don't have a phone."

"Ah OK," I respond, a little deflated.

“I don’t have a computer either.”

The few seconds of confused staring and blinking were not enough to convince Bob that I probably needed a little more explanation, so I asked: if you don’t have a mobile device or a computer, how do you access the app?

“Oh I go to the library.”

I’ll leave you a few moments to allow your brain to produce all the imagery mine did in that moment.

Every day this lovely old chap would wander into his local library, presumably wave hello to the librarian, plonk himself down in the worn council-funded seats at the row of computers near the wall, and embark on his quest for cock.

I love this story not just because of how outrageous it is, and how working at this place every day gave me something new to be able to tell to my grandchildren (once they reach a certain age I suppose, and once they exist). As I chatted with Bob over dessert, he talked about how he loved the events we put on - indeed he looks forward to them every year, because he gets to see friends he met on the app. The gay community isn’t great with older people - one of the unfortunate prejudices rife in a community that should know better is that the fear of no longer being young and pretty creates a very lonely sphere for those who have reached a certain age (although it’s certainly got better over the years). Our app connected Bob with people like Bob, and our events brought them together in a physical place.

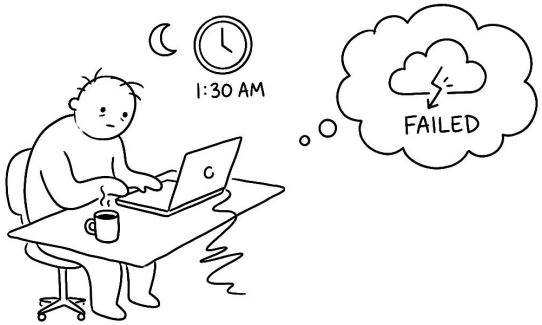
I loved working at that place. It’s where I became an Android Developer. It opened my mind in multiple senses. It always found ways of reminding me why I do what I do. In the tech industry, it was some people’s job to facilitate smoother access to a pensioner’s

finances. Or to allow a daughter living and working on the other side of the world to be able to speak to her parents face-to-face every day. My job, simply put, was to help people get laid. Is there a nobler cause than that?

Whenever I'm having a shitty day I sometimes think about Bob. I think about him sat at his library computer. I think about the library staff and whether they know what he does and whether they just let him get on with it. Maybe they think it's as hilarious as I do. Maybe they even facilitate - keeping watch for people who might interrupt Bob's penis pilgrimage.

But most of all I think about how the code I mindlessly wrote every day can translate into so much meaning for someone, in spite of the tech industry's new Evil-First™ mantra. The app I and my colleagues worked on every day wasn't just something Bob used for hedonistic purposes. It was his lifeline. It connected him with others and gave him the companionship he might otherwise not have. Bob continues to remind me that even in 2026 with the gallery of Tech Titans occupied by a battery of assholes, the things we do can still bestow great good on the world.

Bob, wherever you are (presumably a library): thanks.



An Open-Source Nightmare at Midnight

In the autumn of 2018, I had come to know a truth that visits every craftsman who shares his work with the world: there is both glory and peril in being needed.

ColorPickerView[1] had begun as a modest endeavor, a small open-source project forged in solitude. Yet it had grown beyond my humble expectations. Across distant lands, in companies whose names I had never spoken, my code lived within applications used by thousands. Each star on GitHub was a silent acknowledgment from a fellow traveler. I permitted myself a measure of pride.

But pride, as the old tales warn, often precedes the fall.

In those days, the path to Maven Central was treacherous, a winding road through dark forests of GPG keys, arcane Sonatype rituals, and XML incantations that could break even the most steadfast developer's spirit. And so, like many before me, I chose the easier road:

Bintray. JCenter stood as the great repository, the default sanctuary for Android's artifacts, and Bintray served as its welcoming gate.

The gate, I would learn, swung both ways with alarming ease.

I cannot recall precisely what occupied my attention that evening in Seoul. Routine maintenance, most likely, the quiet work of tending to versions and clearing away old artifacts. The Bintray dashboard lay open before me, and I navigated its corridors with the comfortable familiarity of one who had walked them countless times.

A click here. A click there.

And then, one click too many.

I noticed nothing amiss. The hour grew late, and I closed my laptop, shared a meal with the evening, and surrendered myself to sleep as I had done a thousand nights before.

I did not yet know that I had set a shadow loose upon the world.

The first summons came at the second hour past midnight.

Then another at the quarter hour. By half past, my device sang with such urgency that it seemed possessed by spirits demanding an audience.

Emails materialized like ravens bearing ill tidings. GitHub issues multiplied. Across the vast networks, voices cried out in confusion and alarm:

"Build failed: Could not find com.github.skydoves:colorpickerview."

"The dependency has vanished. Our pipelines lie in ruin."

"This is urgent. Production cannot proceed."

I rose from my bed, a chill settling into my bones that had nothing to do with the autumn air. With trembling hands, I opened my laptop and gazed upon the Bintray dashboard.

The latest version of ColorPickerView had ceased to exist.

Not archived. Not deprecated. Erased from the registry as though it had never been.

That careless gesture during my evening's work—that single, unremarkable click—had banished my artifact into the void. And unlike Maven Central, which guards its published treasures with the vigilance of ancient dragons, Bintray had offered no resistance. No solemn warning. No plea for reconsideration.

The artifact simply... was no more.

While I had slumbered peacefully in Seoul, dawn had broken across the United States and Europe. Developers in those distant realms had begun their day's labor. They summoned their code, invoked their builds, and watched in bewilderment as Gradle spoke of dependencies that could not be found.

Some believed it a fleeting disturbance in the networks. Others suspected corruption in their local caches. But in time, all arrived at the same grim conclusion: the library itself had disappeared from the face of the earth.

And every one of them knew where to find me.

Never have my fingers moved with such desperate purpose.

Sleep had fled from me entirely, banished by the weight of what I had done. My hands betrayed a slight tremor as I prepared a new release. The bitter irony was not lost on me, dozens of urgent messages demanded I repair the damage, yet the only remedy was to perform the very act that had brought this calamity upon us all.

Every click now felt like handling ancient, volatile magic. I examined each button as one might inspect a blade for poison. I read every confirmation as though my life depended upon its meaning.

By the fourth hour of morning, the new version rose from the ashes. I tested it from a fresh project, holding my breath as Gradle performed its work. When at last the words "PUBLISH SUCCESSFUL" appeared, I released a breath that seemed to have been trapped within me for an eternity.

Then began the long walk of atonement.

I answered every message. Every issue. Every inquiry. "Forgive me. The matter is resolved. The fault was mine alone."

Most who received my words offered understanding, we have all stumbled in the darkness, they said. Some expressed gratitude for my swift response. A few, I imagine, quietly inscribed "pin all dependency versions" into their team's sacred texts.

Dawn painted the Seoul sky in shades of gold and rose as I finally closed my laptop. Exhaustion wrapped around me like a heavy cloak, and shame sat beside me like an unwelcome companion.

Yet beneath it all, I discovered something unexpected: gratitude.

Gratitude that the damage had been contained to mere hours of broken builds rather than something far worse. Gratitude for the grace extended by my fellow developers. And gratitude for a lesson that would remain etched in my memory for all my days:

In the realm of open-source, your deepest midnight is another's bright morning. And a single misplaced click can summon you from slumber more swiftly than any alarm ever devised.

Years hence, when JCenter announced its sunsetting in 2021, my thoughts returned to that fateful night. By then, I had long since migrated my works to Maven Central, that ancient fortress where artifacts are immutable, eternal, and blessedly beyond the reach of accidental destruction.

The path to mastery winds through a thousand stumbles. It is not the absence of failure that shapes the craftsman, but the courage to rise and the will to continue.

That sleepless night taught me something profound about the nature of open-source stewardship.

When we publish a library, we are not merely sharing code, we are making a promise. Thousands of applications may come to depend upon our work, woven into the fabric of systems we will never see. A hospital's patient management app. A banking platform processing millions of transactions. A child's educational game. Our code becomes infrastructure, and infrastructure demands reliability.

This is why package preservation is not a mere convenience, it is a sacred responsibility.

In the open-source ecosystem, the disappearance of a single dependency can cascade into

chaos. Build pipelines collapse. Deployments halt. Teams scramble to find alternatives or pin to cached versions that may themselves vanish. What begins as one developer's mistake becomes a hundred teams' emergency. The interconnected nature of modern software means that our actions ripple far beyond our sight.

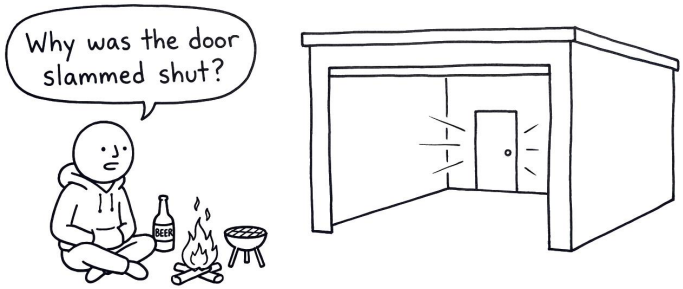
Security, too, demands our vigilance. A repository that permits easy deletion also permits easy replacement. In the wrong hands, such power becomes a vector for supply chain attacks, malicious actors who might replace a trusted package with a compromised version, or claim an abandoned namespace to distribute malware to unsuspecting dependents. The very convenience that made Bintray appealing also made it vulnerable.

This is the wisdom that Maven Central embodies.

Its guardians understood from the beginning that published artifacts must be immutable. Once released, a version cannot be altered or removed. This is not bureaucratic obstinacy, it is a fortress wall against both accident and malice. The GPG signatures that once seemed like arcane obstacles reveal themselves as shields, ensuring that the code you receive is the code the author intended to send.

To my fellow open-source developers, I offer this counsel: choose your repository as carefully as you choose your dependencies. Embrace the tools that protect not only your work but the countless projects that will come to rely upon it. The extra effort required by Maven Central is not a burden, it is a covenant with every developer who will ever type your package name into their build file.

We are not merely writing code. We are building the foundations upon which others will build their dreams.



Cats Behind the Door

An Englishman (Ben), an Irishman (James), a Scot (Niall), and a German (Marc) arrived at a Welshman's (Ash) place for a BBQ on the last day of Droidcon. For some of us, it was the first time seeing the place, so we were shown around a bit while Ash explained some of the issues with the architecture, the curious pipe system, and the dodgy electrics. We were in the basement, more specifically in the billiard room, kitted out with a fridge, a very nice billiard table, etc. The BBQ was outside, and to get to the toilet from there, we had to walk through the garage and the billiard room, into the house.

This meant a lot of doors could be left be open. Alexandra took the time to carefully explain something to us, something very important: their cats, Kotti and Py, are very friendly but also curious. They love people and try to come to us when visitors are there. They are house cats, not allowed outside, and they are incredibly quick too, so it is very

important to make sure the billiard room door to the garage is closed before opening the other door to go to the... "You mean this door," as Marc said while opening that very door, at which point one of the cats shot through and made a bolt for the other door.

Ash, somehow already perceiving that this would happen, was already in place to close the other door very quickly. "Yes, Marc, that door." It turns out the Scot had warned the Welshman that the German might do something like this while it was being explained to him. The Irishman stood by watching and laughing while the Englishman was drinking a beer outside wondering why the door was slammed.

Courier Feedback at My Front Door

About six years ago, I worked at a startup with two apps: one for ordering food and one for riders delivering it. Our team built the rider app from scratch.

At that stage we were in pure speed mode: launch features fast, grow quickly, and worry about edge cases later.

One evening I ordered dinner through our own app and thought, if the assigned rider uses our rider app, I can ask for real feedback in person.

The rider arrived. I thanked him, took the food, and casually asked what he thought about the app - without telling him I was one of the engineers.

He gave sharp feedback immediately.

Not just "this is good" or "this is bad" feedback. He talked about SDK integrations, NFC/Bluetooth payments, and how money flow in the experience could be improved. He even argued some of it should be quick to implement.

I was surprised and asked how he knew so much.

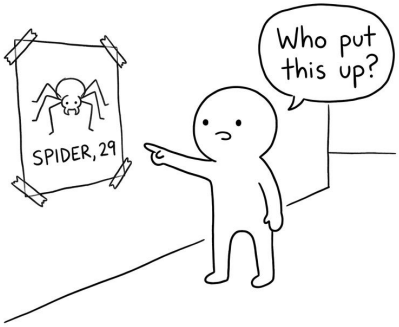
He said he had just covered similar topics in computer science class.

That moment was equal parts useful and humbling. Users can be technical, opinionated, and absolutely right about friction points.

I got strong product insight that night.

I also got colder food than planned.

User interviews are great, even when they arrive with your dinner.



Emma vs. Spider

Sometime after 1.0, a product manager decided that the team needed to internalize a sense of Android users, to better understand their motivations and better inform product and engineering decisions affecting those users. Ficus Kirkpatrick explained, “We were meant to be seeing this as a constant reminder of who we were supposed to be thinking about as we were building Android.”

So the PMs came up with a fake persona, named Emma, along with detailed facts about her, such as her age (21), her vocation (shop assistant), her interests, and the kinds of things that she used her smartphone for. They created a poster with this information, including a stock photo of someone representing this persona, and hung it on several walls in the Android building.

A few days later, Emma's poster was joined by one for another persona, but this one was not condoned by PMs. Spider was 29, a drug dealer, and used several burner phones for transactions.

The Spider author remained a mystery to everyone for many years. It was only when I interviewed him for the Androids book that the secret came out. Well played, Brian Jones.

How a Bot Can Delete Your GitHub Repository

We all have a story like this one, or in my case, many, where a seemingly innocuous change, resulted in a catastrophic failure or destruction. Sometimes the result is isolated, and you can recover whilst being able to hide your shame, and sometimes the result is public, where you might need a certain amount of damage control.

This story, whilst having necessitated recovery, has not been so public, and more akin to having shot myself in the foot, and face simultaneously. But its cause deserves the shame of public visibility, if only for its comedic value, and learning opportunity.

A bit of background, perhaps: I maintain a playground project, where I like to play with various technologies to sate my lust for conference driven development. It does an excellent job of letting me find out what may or may not be a good idea for production projects. This project uses a lot of technologies that could be described as

over-engineering, but are mostly present for shits and giggles.

One of these technologies is Terraform, something quite familiar to people who have worked on projects with Infrastructure as Code (IaC). I can actually highly recommend codifying your infrastructure, since it allows you to discourage management from managing projects through online consoles or control panels.

It may also make sense to utilise Continuous Integration or Deployment (CI/CD) ; there exists a diverse ecosystem of tools and utilities that allow you to take full advantage of this – in my case, I use GitHub Actions.

In some circumstances, depending on the resources you are managing with Terraform, you might need elevated privileges – specifically, privileges that might not be permissible by the GitHub Actions workflow that your Terraform configuration might be working on. This can usually be provided by creating a GitHub App, which can be installed into your repository with an elevated and fine-grained permission model. The configuration of such an application is a little out of the scope of this article, but you can consider it as having the same level of authority as an organisation or a user, instead of a repository.

Now perhaps, you might be inclined should your level of sanity deviate from the nominal average, to include the repository itself as a resource that could be managed by Terraform, with elevated administration privileges. This might even be considered a rational decision: you might want to manage labels on a repo, or configure repository secrets based upon the result of a Workflow. This is actually fairly convenient when considering keyless authentication from GitHub Actions, with Workload Identity Providers[1].

You might see where this is going, but you could be forgiven for thinking that this configuration was harmless, having been used for years without any problems. It only comes with one little innocent refactor, and then everything goes to shit.

When I first added Terraform, I liked the idea of using modules for isolation of configuration. A little later I assumed that without the requirement of re-use, this was a little unnecessary and could more clearly be facilitated with a top-level Terraform file. A reasonable assumption, you might assume.

No worries! These learnings often happen in the process of software development. The ability to adapt and learn from your experience is fundamental to our lives and abilities as software developers. A simple small PR will move the project in the right direction, and with the bonus of removing more code than adding. Nothing is more cathartic than deleting code.

This[2], on its own, would not have been enough to cause the clusterfuck that would inevitably ensue if it had not been for another recent change. Given that I had recently been working on refactoring Terraform files, I wanted to allow myself the option to apply a Terraform configuration on a PR by explicitly applying a label. Ordinarily this was only limited to the main branch, allowing me to verify the Terraform plan as a PR comment before merging, a decision perhaps sufficiently deserving of derision alone.

That being said, this PR should not have enacted the Terraform Apply, the conditions for this to have been done did not apply, the PR had neither the necessary label nor was it executing on the main branch. Something was wrong…

I like using the GitHub CLI, so a PR was created with ease, the automatic PR actions triggered, and now I await feedback. Whilst waiting I frequently like to jump back to main, ensure I'm up-to-date, and perhaps merge other branches I might be working on in parallel.

```
Git pull: repository not found
```

Huh, I guess GitHub must be down again…

GitHub Status normal.

Repo missing from dashboard.

Suddenly, my heart rate rises significantly, a project that whilst not in production, that had documented a wealth of knowledge I had accumulated for nearly a decade, was gone. The feeling akin to having just shredded a vital set of documents, what had I done? How had this terrifying circumstance come to occur?

Then the realisation sets in. A vague recollection of documentation sitting near the periphery of my memory, how Terraform manages its lifecycle. Resources, if moved, can be destroyed, before being recreated. Without realising, with that minor refactor, I had moved a resource from a module to the top level declaration. It had a different identifier, therefore it was a whole new resource.

But no, surely a repository would have enough permissions to delete itself? Well, remember that elevated privilege configuration I mentioned earlier? Yep, it's gone.

Faced with the reality of my poorly considered decisions, I reflected upon what mistakes I had made, and what I could do to prevent this from happening again.

Firstly, it is always important to consider the principle of least privilege. The access token should only have access to what is absolutely necessary for the work to be done. Naturally, this would have been best practice, but sadly, in this circumstance, it had slipped my gaze. In this circumstance, I granted write permissions where read would have sufficed. This is a human error, and it can happen, which is why it's essential to perform regular evaluations of access control.

GitHub actually provides an action that can help with this, though it's more for job specific permission grants, rather than those that might have been given from a higher privilege level.

Some of you might think, if you are already familiar with Terraform, that surely there must be tools available for this. Something you might remember if you are fortunate enough to have the foresight, when having renamed a resource identifier, even inadvertently. Then you would be correct. Terraform provides two tools for this purpose: one that requires manual invocation, and one that can be added to the configuration.[3,4]

Whilst at first I thought it might have seemed like a good idea to attempt debugging Terraform scripts to improve the feedback cycle during the development process. The level of risk will never justify not being able to scrutinise the plan before applying it.

Terraform, will generate a plan, detailing its intention to apply, destroy, or update resources. This plan is meant for scrutiny, to verify something before blindly making changes to actual resources. This was on me. If you ever think it's a good idea to ignore the Terraform plan, let this story be a reminder not to do so. But more importantly, this serves as a reminder to myself in the future not to make this mistake again, because I'm fairly

certain this was my second attempt[5].

The first attempt was less harmful, as it disabled an implicit default behaviour.

`DRY_RUN=true` seems like it might be intuitive, but this is opt-in behaviour, since in most environments, a boolean will evaluate as false by default. Dry run might be fine for something like a Gradle build, where you have executed a command, but a Terraform workflow can be more destructive, and should require more safeguards.

In addition to the destructive behaviour, it's important when working with software to write code in a way that is easily understandable to humans – or, as is often described in software engineering, "reducing cognitive load". This simply means reducing obstacles to confusion, like not using double negatives, preferring truthful conditions, or in general, being positive.

`shouldContinueExecution` is much clearer than `shouldNotContinueIfStatementIsFalse`

By this logic, `DRY_RUN=true` requires an explicit declaration override, where you might imagine false to be the default. The default, being false, means that this is an opt-in rather than an opt-out.

After I posted this hilarious anecdote on ASG, a friend of mine wrote that this would be a fantastic example of something one could talk about in an interview, when asked about a failure, and how to recover from it.

Q: and how did you recover from this mistake?

A: I begged GitHub support to fix it for me.

Some recoveries aren't as elegant as you might like to imagine.

For most repositories, accidental deletion is easily reversible from your GitHub profile, but this doesn't apply to forked repositories, and this was a fork of one of my own former repositories.

Whilst I did have a local copy of this repository that I could feasibly push to a new remote origin; it would transpire, that due to having worked on different machines throughout the history of the repository, not all of the LFS remote objects had been synced.

This presented an issue, I could either rewrite my entire history, re-referencing LFS objects to newly created resources, though I would never be able to actually recreate them in their originality. Or, I could try to fudge the signatures in a very hacky way, something I'm not even sure would have worked out.

To my avail, I had heard back from GitHub support, they had been able to restore my repository, albeit with an odd fork subscription. The origin of the repository had been a fork of another of my repositories, forked by another person, then misattributed. Thankfully, disconnecting from a fork is fairly easy.

Sadly however, whilst being able to recover the repository, the GitHub Actions history, and secrets, had been purged. So no chance of being able to find the execution history of why this Terraform Apply had been made when I thought it should have been a dry run.

I had no choice but to allow the workflow to run again, with safeguards in place, in an attempt to diagnose the issue.

You could say that this might have all been avoided by getting this single bash statement correct in the first place. But ultimately, it was all the steps that led to this point that allowed the damage to be done. The bash statement, whilst a shameful display of my bash ineptitude, was only the match to the bonfire.

```
$([[ ! $has_terraform_label && ! $is_main ]] && echo true || echo false)
```

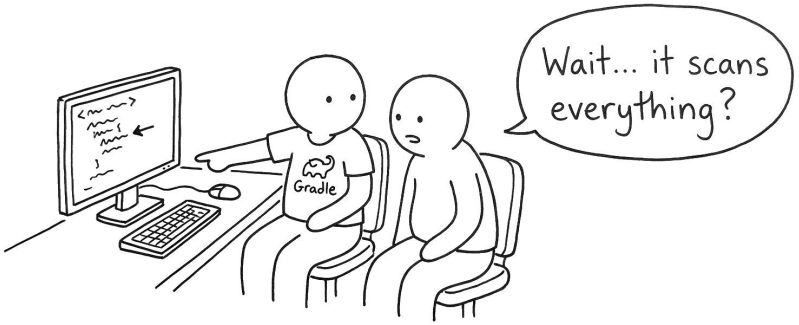
In this statement, false==true. It's enough that it's defined, and unless it's directly compared to `false`, it will resolve as true. So when either `has_terraform_label` or `is_main` are false, the statement will evaluate as false (due to the negation).

Dry run was still disabled.

But nonetheless, I persevered, attempting to disable the evaluation, and found that the Terraform Apply had again been executed. I had only been saved by having reviewed the access token, and revoking the write permission to repository administration.

Finally ending this saga, I decided to revert the whole idea of executing a Terraform Apply on a PR.

No PR should have this power.



How a Gradle T-Shirt Helped Fixing a ~300,000€ Bug

"Cache me if you can".

That's written in white text on a black t-shirt, accompanied by a cute elephant - the logo of the Gradle build tool used by millions of Android developers worldwide, and many more - be it backend, desktop, Kotlin or pure Java applications. Most t-shirts in my collection are tech t-shirts I received at events and tech conferences over many years of my development career, and they fill me with pride. They remind me of the things I learned, the events where I received them, and of the people I met in all these years - making many friends along the way! But this particular Gradle t-shirt has a special place in my heart. I've given a few live-coding talks at Droidcon conferences in the past years, and in many of them I swarm about Gradle and give tips on how to use it well - and how to improve those precious build times. After talking with the nice people of Gradle Inc. at their Droidcon

sponsor booth, and telling them that I love Gradle so much and that my talk contained lots of knowledge about it, they surprised me with this t-shirt.

Now I can already sense your "I call bullshit" tingle go off: How can Marc love Gradle? Many developers dread working with it, and I understand some of the reasoning. It's a complex tool, with a steep learning curve, and most of all: developers want to focus on writing their application code, and they usually have to touch Gradle when something does not work as expected or something breaks. Trust me, I was there. But after some time, I've started to build a mental model around Gradle, learn some of the pitfalls, and learned some tricks. And most of all: it allowed me to do the one thing I enjoy most: making developers happier! In one project I applied best practice after best practice (low-hanging fruit: activate the Gradle configuration cache if you don't have yet - it's awesome!), reducing the build time for a ~350 module Android-app from many minutes to under a minute on an M3 Pro MacBook.

Fast forward a few years, and I was working as a consultant on a complex Android product - with dozens of teams, probably hundreds of engineers, likely even thousands. The tasks of the team I was assigned to were extremely interesting - challenging and just above my skill set, so it fulfilled me with joy to combine my existing knowledge with something new I needed to learn, all to improve the developer experience of my team. I worked there for a few months, mostly remote. This time I traveled to the customer's office and stayed for a few days.

Then I got a coffee. Not just any coffee: the client had a nice portafilter machine in the kitchen area, so I could make myself a nice cappuccino - super fine Espresso with just the right crema, and I liked to do some latte art with it. It was after lunchtime, so many people

got the same idea, and I had to wait in a queue. A guy behind me noticed my Gradle t-shirt, and it got us talking. We chatted about Gradle, and I told him that I really enjoy working on this stuff, and that I find joy in making builds faster. He mentioned that their Gradle builds were quite slow, and that the cause could be that they were using some custom plugins made by another team. That got me curious - as soon as I hear some fellow developers complain about a bad developer experience, I just can not resist. Also, it widens my horizon: every problem I can analyse and find the root cause for deepens my experience for future debugging sessions. So I simply offered him: "How about we have a look? Let's take a few minutes and see where the problem is!" It didn't matter that he was on a different team than the one I was assigned to, or that the task was not on my agenda. These small pairing sessions often are incredibly valuable. I just didn't know it at the time how valuable it was going to be.

So we sat down at his desk. He described to me that every Gradle invocation was taking a long time. And sure enough: running a simple incremental build would hang at 0% for dozens of seconds, and then quickly complete within seconds after that. I wondered if that would be the same for any task, so I ran `./gradlew help` to run the built-in help task that comes with Gradle by default. And sure enough: even that simple task took 35 seconds, with the same progress: first 0% for a long time, and then immediately running to completion. So something had to be really slow during the initialisation or configuration phase. No problem, configuration cache to the rescue, because "cache me if you can"! I ran `./gradlew help --configuration-cache`, but the project was not compatible yet. Bummer. And upgrading is nothing that can be done in just a few minutes, because it would have affected multiple teams. So back to finding the root cause: what is happening in these 35 seconds? Yep, you could profile the build, or do a build scan, but often the most basic debugging

tricks will go a long way. Because Gradle runs code, you can simply *put print statements everywhere*! So I just added print statements at different spots in the `settings.gradle.kts` and in the root `build.gradle.kts` files. Just `println("AAA")`, then after the next code block `println("BBB")` - you get the gist. Then we ran the build again, and we noticed a big, oddly 35-second long gap between AAA and BBB! We looked at the code block between the two, and we found the root cause: just a few lines of code were scanning the full project folder - all files, recursively - for `build.gradle.kts` of the modules. But in this case, the project had millions of files - because the code scanned all files, including all build folders - only to find the same ~100 files every time! After finding the root cause, the fix was easy: simply hard-coding the includes directly! The 35 second waiting time vaporized, and the next help invocation took just a second.

We were both really happy that we found the issue, and it took us only 15 minutes to find it! I wondered how much of time we saved for the team. We checked in the Git history and found that the code has been sitting there for close to 2 years. If every developer had run 100 Gradle invocations each day, 35 second of waiting time each, on most working days a year, multiplied with the amount of developers on the team, this small issue cost the team 300 working days over the last 2 years, or 300.000€ when one developer costs 1000€ a day! Of course this is just a very rough estimate, but it's easy to say that the 15 minutes of pairing session was a really great investment of our time! Later on, it turned out that the build times on the CI server went down drastically as well, which we didn't even consider for our estimate. So much productivity gained!

In the last years I experienced myself or heard about many of these stories: developers

digging around, finding and fixing issues that no one even knew existed, or everyone already had accepted as the way it always has been, or that got labeled as someone else's problem. The flaky test suite that drove developers crazy, but didn't get fixed properly for years. The DevOps engineer that found that autoscaled CI-agents were not currently shut down, leading to millions of avoidable server costs. The Android developer that digged in the iOS build pipeline, and found that the Rosetta mode (the technology to run Intel-based apps on Arm64 chips) was still enabled, which meant that the builds were almost twice as slow as without it.

All it takes is people that care, even if it's not their main job or expertise. People that collaborate. That take the time to find the root cause, and are not satisfied with assumptions. Companies can help by reducing busyness and by bringing people together - a nice coffee machine can work wonders! And sometimes, a simple tech t-shirt can help sparking a conversation that will lead to big productivity gains, happier developers, and ultimately: a win for everyone!

How Julien Salvi Saved My Life in Paris

So this is a story about the time I almost caused an international incident… over pizza.

Every year at Android Makers, we organize an after-party at the Café Oz Denfert. We rent indoor-outdoor seating for all the attendees. It's one of those cool Paris places with dim lighting, exposed brick, small tables and mostly locals. Not a lot of tourists.

We had planned everything. There were drink tokens for everyone, there was free food for the attendees, and we had tables of finger food all around the room. You know the kind — little sliders, pizza slices, things served on wooden boards that make you feel fancy even though you’re just eating a very small burger.

And I felt very relaxed because I had ordered all the food. I knew what we ordered. I had seen the menu. I had looked at the invoices. I basically felt like the food authority of the evening.

If someone asked me what food was at the party, I could tell them.

So I'm walking around the venue, talking to developers, checking that everything is going smoothly… and then I see something.

Across the room, there's a table with food on it that looks… significantly better than the food we ordered.

I mean really good.

There's pizza. Real pizza. Big slices. Melty cheese. Beautiful crust. And a bunch of other amazing finger foods I definitely do not remember approving on the invoice.

And I stop and think to myself…

"That's weird."

Because I know what we ordered.

But then another thought comes into my head.

"Well… we rented the place and I paid for all the food."

So logically… that must also be our food.

This is the moment where my brain could have asked a few very simple question.s

Like…

"Hey, is this for our event?" or "Wait…did we actually rent out the entire place?"

But instead my brain did what any tired conference organizer's brain would do.

It said:

"Free pizza."

So I walk over confidently. Like I belong there. Like I am the CEO of pizza consumption.

There are a few people sitting at the table. I don't recognize them, but that's normal — there are hundreds of developers at this event.

So I do the most natural thing in the world.

I reach over.

Grab a slice of pizza.

And start eating it.

And I'm standing there, chewing, thinking:

"Wow. This is way better than the food we ordered."

And that's when I notice the two guys sitting at the table staring at me.

Now when I say these were big guys… I mean big guys. The kind of guys who look like they do CrossFit recreationally and intimidate people professionally for a living.

And they start talking to me.

In French.

Very fast French.

Very loud French.

Now here's the problem.

I do not speak French…anymore. I took 5 years of high school and college French. In no way did that prepare me for the verbal tongue-lashing that I was receiving. I can maybe order a ham sandwich in French…on a good day.

But that seemed decidedly unhelpful in the moment.

One of them is pointing at the pizza.

One of them is pointing at me.

And at this moment my brain slowly starts assembling the pieces of the puzzle.

Piece number one:

I do not recognize these people.

Piece number two:

This food is not on our menu.

Piece number three:

These men look like they could fold me into a small backpack.

And then the final realization hits me.

We didn't rent the whole restaurant.

We rented… part of the restaurant.

Which means I had just walked over to a completely different group of Parisians…

Stood by their table…

And started eating their pizza.

Without saying a single word.

Not even "bonjour."

Just full food piracy. And I'm standing there holding a slice of pizza like the world's dumbest criminal.

At this point they are very angry. Very loud. And one of the guys grabs the pizza out of my hand, throws it on the floor and is getting more and more animated about the whole international incident.

Luckily (and this is the only reason I'm alive today) one of the people from the local Android developer community sees what's happening. My hero of the evening: Julien Salvi.

He runs over.

And he starts talking to them in rapid French.

And you can see the whole story unfold in real time.

First they're angry.

Then they're confused.

Then they look at me.

Then they look at the pizza.

Then they look back at me.

They are still not happy. But an uneasy truce has been formed. It seems my life will be spared.

I'm fairly sure Julien said something like:

"No, no, he's not stealing your food… he's just an idiot."

Which, honestly, in that moment he was really just telling the truth.

Julien does the smart thing and quickly whisks me away from the very large men and I retreat safely back to our side of the party.

Where the food is not quite as good, but significantly safer.

And I learned two very important lessons that night.

Lesson number one:

Always confirm exactly what part of the restaurant you rented.

And lesson number two:

If you’re in Paris… and two large French men are yelling at you…

Put the pizza down.

Slowly.

And wait for someone who speaks French to save your life.

Thanks again Julien. I owe you one!

I Deleted the LDAP Server

Once upon a time - this is how every story starts, right? However, I just want to speak of 1 small anecdote, while giving you the punch line upfront. One of my former bosses was saying: Everybody has one wildcard to mess up pretty well, but some people need one each day. Well, the latter feels like me, of course.

Once I worked at a university as a working student. We had been a small team of like 6 plus 2 full employees, and our main duty was to administer the pc and server pool of the faculty. I was mainly responsible for changing/repairing/maintaining computers and some minor tasks like cleaning up the virtual machines or reinstalling the system on computer xy, while the other worked on Ansible, LDAP, or any other fancy things.

One day, I came to work as normal sit down, and started my shift after a few days off. I had yet another cleanup task assigned to me - since the others had some work finished on

some virtual machines, which I was now supposed to dispose. I scrolled through the machines very enthusiastically and deleted one after another. One of them was called something like - Virtual LDAP server. What I did not know was that during my absences, they exchanged the physical LDAP server with a fresh new virtual one, which also had a new LDAP tree and everything. Not knowing and with the full confidence of my boredom, I of course deleted the virtual server. 2 minutes later, the telephone rang, and my boss was in panic, asking if the LDAP server was down - upsi. I just blacked out the entire faculty for almost a week.

I Don't Use Emails

A couple of years ago I was working as a student in a big German company that has a factory. My task was to create a small PC-App that can help their factory workers calculate some values for production. It was an app that opened a window and you could click on buttons and some values were recalculated on the fly.

At some point I was presenting the app in front of our "very big" boss and some other "big" people. They really liked the App, all that "clicking" on buttons and how the value was calculated straight away.

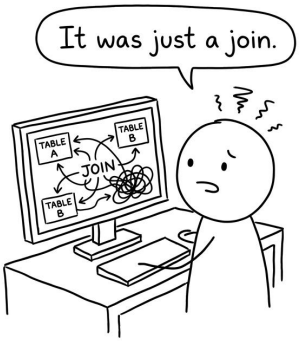
At the end of the presentation the main boss said:

"oh, that was all very interesting. How can I learn more about this app?"

"I could send you a zip per email if you want, then you can play with it a bit"

(it was a really small thing, so it could fit in an email). The response (in front of the whole audience) (As Germany is well known for its digitalisation):

"oh, I don't use emails.. could you please print your app and put it on my desk? Thank you"



It's Just a Join

2015, I had just gotten my first gig at a company my dad worked at. I got hired because I was an honest, upstanding guy. It was, indeed, the past.

It was one of those jobs where you touched everything nobody wanted: VBASIC, Microsoft Access, and some cursed drag-and-drop tool that made SAP look like sci-fi.

One day I was at a customer: Odebrecht. Yeah, *that* Odebrecht. Think Enron but global.

My boss showed up: big dude, striped shirt, tie pulled tight, straight out of *Wolf of Wall Street*, arms full of papers. He dropped a task on me: “I need you to merge the sheets of all employees worldwide. They’re split between Brazil and the rest of the world. Combine them.”

“I barely knew SQL,” I admitted.

“You had time. SQL wasn’t that complicated. This should be a simple join.”

We stared at each other. I realized there was no universe where I convinced this guy this was above my pay grade.

He walked away. I sat down at this random computer buried on the fifth floor, stared at the tables for a few seconds, then started Googling. Yes, we still used Google in 2015, in case you were reading this from the neural-implant future.

The tables were huge. Columns everywhere. Attributes I barely understood. But my untrained eyes kept seeing the same thing: the attributes in both tables looked identical.

So yeah, it really *did* look like a simple join.

I wrote the query. I ran it. I went to get a coffee.

When I came back, I started calmly checking the data. I had some names jotted on paper, a few users I wanted to sanity-check. And then I got that movie moment, the invisible shot where the coffee cup slipped from the guy’s hand in slow motion as he watched the disaster.

The join failed. Not with an error. Worse: it “worked,” but wrong.

My stomach dropped. I ran back to my boss and explained what happened.

“How could I rewind?” I asked. “Like… restore the old state?”

He squinted. “Rewind what? This was production. There was no old state. Don’t worry, I was sure you’d find a way to fix it.”

We stared at each other like we lived in two different realities. He turned back to his work like nothing was on fire. I walked back knowing I’d just broken the thing that decided

whether thousands of people got paid correctly that month.

He knew something I didn't: if you gave a guy with ADHD a bucket and a burning building, he'd find some way to put it out. I didn't know that. I just knew I'd screwed up a lot of people's payment sheets.

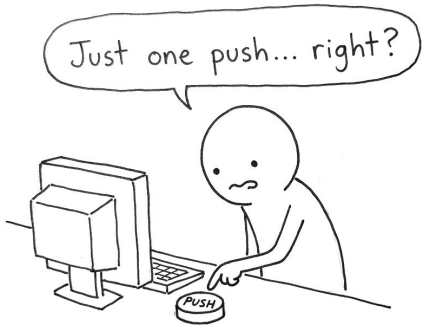
So I sat down, clenched, and started really analyzing the data in my notes and on screen. Somewhere in that mess, there had to be a way to untangle it.

If this could be merged, it could be unmerged, I told myself. Naive? Yeah. But it gave me something to hold on to.

My brain recruited every neuron it had. You know that gif of the guy with numbers floating around his head, seeing math in the air? That was me, except instead of blackjack probabilities it was employee IDs and foreign keys.

In the end, I not only undid the mess, I also finally understood why my join didn't work and how to do it right.

The best part? I did all that heroic data surgery on what was probably criminal spreadsheets.



Just Another Push

New developer, new country, new company and a freshly rebuilt Android team. That meant a large legacy app, many users and a gnawing fear of touching anything. Not our new developer though, they know what they can do.

On one warm afternoon, right at the start of summer break, the product manager had a great idea: let's promote our new discounts with push notifications!

"Great!" the new developer said. They had a lot of experience with those, it should be a piece of cake. And so they went ahead to sit with the PM and walk him through what they were doing.

Most of it was muscle memory by that time. Even if they had never sent push notifications for this app, it was just another push: select the text, select the user group, select the

actions and push them out. And so they did.

SUCCESS! The notifications were arriving perfectly in QA.

"Let's send them out to all our 10k users now!" said the PM.

Not a biggie, thought the new dev. They had sent push notifications to more than a million users before. These are rookie numbers. Still, just to be conservative, let's send them by user groups. And so they started.

Everything was going smoothly, the new dev was raking cool points with all the team. None of them were brave enough to handle this task, so they smiled and decided to ping their backend friend to brag and get dinner together after work. Their backend friend, in turn, shared that her day had turned sour: she was the BE on-call person and in the last five minutes of her turn everything had gone to shit and nobody knew why. Not a super strange occurrence. Backend folks had to handle a lot of outdated slow services, and those constantly required tuneups. Without asking much details to let their friend focus, the new dev sent their regards and continued with the next push notification.

As they pressed send for the sixth time, they saw a new message from their friend:

"OMG, we're being attacked! Our services are being flooded with login requests!"

"Wait…" the gnawing fear crept in, "wasn't there a silly login refresh on app startup?"

"Oh…" thought the new dev, as they finally opened their analytics dashboard.

Everything was red. Customer support was drowning in complaints.

"Hey, we're missing two more groups. Why are you not sending the next push?" the PM asked.

"No, we can't." they said as the DDoS attack kept going.

"Just another push"

LOG.WTF()

There is a class in the Android API called Log, which is used by developers to write messages out to the system log. All of the methods in this class denote the type of event being logged, such as “d” for *debug*, “i” for *informational*, and “w” for *warning*. But one method is a slightly longer: “wtf”. The method’s meaning, according to the docs[1], is “What a Terrible Failure”. But anyone familiar with current acronyms probably assumes a different meaning when reading the letters *wtf*.

Dan Egnor added this method. Dan worked on the OTA update mechanism and the check-in server, and was passionate about tracking problems happening on the fleet of devices. The other Log APIs were mostly useful for developers looking at the log output on a development device. But Dan wanted something that could track and log serious problems in the field and send that information to the Android team. So he added Log.wtf,

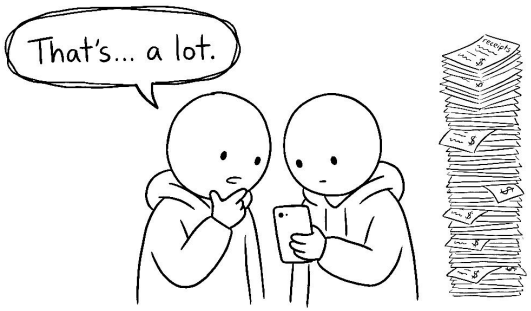
which recorded detailed information about the problem and uploaded it to Android.

A couple of years later, someone within Google emailed Andy Rubin to complain, saying how unprofessional it was to have this method name in the code and in the public API, and that it should be removed.

Andy responded by immediately, sending out the following email and copying the entire Android engineering team:

```
From: Andy Rubin

Date: Mon, May 24, 2010 at 10:12 AM

Subject: Re: "What a terrible failure?"

I apologize on behalf of the team if we offended you.  I'm
certain it wasn't intentional.

I've cc'd the Android Engineering team in hopes that the
person responsible comes forward and explains how the fuck
this happened.
```

What's particularly funny about this response is its deadpan approach. For anyone that didn't know Andy, they might think that he was truly upset at Dan. In fact, Dan was worried that he was in trouble when he first saw this: "I remember being kind of terrified and ashamed, because I thought Andy was serious." But colleagues convinced him that it was Andy humor, given the language in the email. Andy got the original joke, and intent, of the API, and approved. He was effectively telling everyone to ignore noise like this and to get back to work.



My $32,000 Bug

A few months after joining Square in 2012, Bob Lee asked Jesse Wilson and me to bolster the Android open source libraries they had started to build. This was the start of something truly special! And also the start of the worst bug I've ever been a part of…

Quick–let's do a crash course in Square libraries and payment processing.

The main Square library at the time was Retrofit. Not as you might know it today, but as a bag of goodies covering HTTP, serialization, persistence, shake detection, and more. The first task was to split it up. Retrofit kept the HTTP bits, persistence became Tape, and shake detection became Seismic. Each did one thing and did it well.

Now the Square app of the time only did card payments which is a two-step process. The "auth" does a quick check to see if the funds are in the bank, and the "capture" goes and

moves them. Maybe you've seen a pending transaction show up on your bank account only for it to disappear later. That's someone doing an auth to verify your funds without doing the capture.

Our tale involves Tape, which was an on-disk task queue. You append items, it puts them on disk, and you pop them off as they're processed. Tape actually offered the same "ACID" guarantees of a full database, but without the overhead. Remember, this was 2012 and our minimum-supported API level was like… 5. When dealing with other people's real money on cheap Android hardware you can't afford to lose data.

Each payment in the Square app triggers an auth, persists a capture task to a Tape queue, and then shows the success screen. Sometime later, the queue is processed to actually perform the capture. This all worked swimmingly.

But Tape was not perfect! Over time I found small improvements that could be made, and even some bugs to be fixed. One innocent change of mine removed a dedicated callback in favor of an existing, generic listener. Tests still passed. All good, right? (Cue ominous music!)

Weeks later a Square app update goes out on a Thursday. Back then there were no staged rollouts and no auto updates. Some people updated early Friday and started taking payments. Late in the day we noticed captures were not being processed. Yikes! That's money not moving.

By Saturday, every hour added one million dollars to the pile of money stuck waiting to capture in Tape. The Android heavy hitters of the time, Eric Burke and Ray Ryan, identified the problem, fixed it, and shipped an app update that evening. The old version

was denylisted to force an update. For those who didn't open the app, failsafes were triggered to upload the entire Tape queue for server-side processing.

The bleeding had been stopped. After a couple days something like $20,000,000 of stuck funds were successfully captured and paid out to merchants. Only about $32,000 dollars was unable to be captured, and Square just ate that cost and paid it out anyway.

So was my "innocent change" in Tape to blame? Yes and no. Long ago the app made an assumption about a parameter of the listener, and my patch changed it slightly. This change was valid in isolation (tests all passed!). By breaking the assumption, though, Tape inadvertently prevented the app from seeing the queue had items to process.

I have caused a lot of bugs in my life, and I will cause (and fix) many more in the future. But I'm not sure any will actually cost more than $32,000 of someone else's money.

One Hundred Engineers

One story that always sticks with me is from Droidcon london about ten years ago, back when the first day was still a barcamp and you'd pitch your talk that same morning. There was a guy from Facebook pitching a session on engineering challenges at scale.

Someone in the audience shouted, "How many developers do you have working on the app?"

He goes, "About a hundred."

You could feel the collective gasp. Everyone was stunned that a hundred engineers were working on an app that slow and clunky. It became the unspoken meme of the day.

Party Like It's 1995

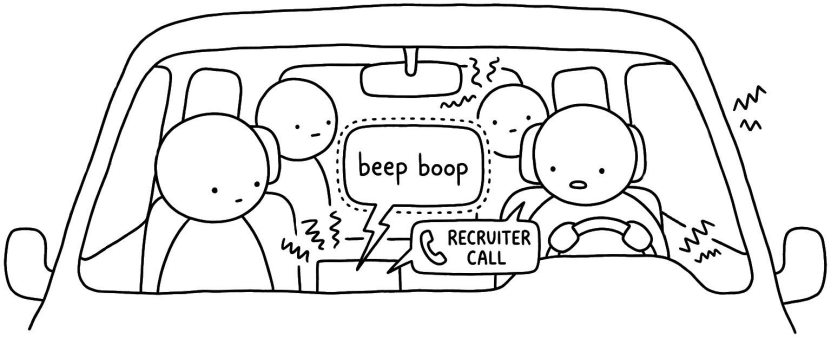
Android has a long tradition of adding easter eggs in its APIs. For instance I remember when Mathias, our graphics engineer at the time who also worked on the early sensor APIs, was doing the math to figure out the *pull of gravity at the surface of the first Death Star*[1]. My own personal early contribution was to add an undocumented tag to the parser handling View XML layouts: you could (and still can) add a `<blink>` tag to your layout to make its content blink, just like in early HTML. In the source code, the constant for this tag is appropriately named `TAG_1995`. Why 1995? No idea, probably because I was writing websites around that time and the date felt appropriate for when this silly tag was popular.

Pubic Class and the Word-Powered App Factory

When I interviewed for my first Android job I had to write a solution to a coding problem on a whiteboard and the first class I wrote down had `pubic class Foo { }` instead of `PUBLIC class…` I didn't notice until the end as I was so nervous but I did think it was odd that the guys interviewing me were suppressing a few sniggers.

The second story was at the app agency where I finally got the job. A few years in, some journalists were visiting our company to see how we developed apps and had photographers taking pictures. I had some pseudo-code written in Microsoft Word for an app I was working on and it must have been caught in one of the photos.

A week later someone noticed the picture in the article, zoomed in, and tweeted that our company was using Word to develop apps when they saw Microsoft Word with code on a developer’s screen.



Recruiters

Last day of Droidcon Berlin. Ash invited a few of us to his place for a BBQ.

I was driving, so the van lineup was me, Ash, Ben, Marc, and Niall. The Android chaos squad.

We had just set off when my phone rang: recruiter.

As a freelancer, you don't ignore those calls, even when the timing is awful. So I answered, kept it polite, and said I'd call back properly at the first chance.

While I was on the call, the van went weirdly quiet. Too quiet.

Then right as I was wrapping up, Niall shouted from the back:

"F***ing recruiters!"

There was a two-second silence that felt much longer.

Then from my speaker: beep boop.

Call ended.

Everyone turned to Niall. Ash, very calmly, reminded him that my wife Betty is also a recruiter.

That was it. Van-wide laughter. No survivors.

If your phone is on speaker, your friends are now part of your call script.

Skipping JIRA to Build Burst

It was fall of 2024. My dear friend Jake and I had been building a Kotlin multiplatform thing at a fintech company.

It was challenging and exciting. I was focusing on performance, which often required replacing small simple functions with fast complex ones.

Writing tricky code is fun, if it can be tested. But in multiplatform we didn't have the niceties of modern Android testing:

- Dependency injection
- Paparazzi
- TestParameterInjector
- JUnit rules

We were chatting about this at our company's annual full-mobile-team summit in San Francisco. The summit had a packed agenda with sessions on all the usual stuff like observability, design systems, and oncall.

I looked at the agenda, and I thought about the library we wanted to write, and I looked at Jake.

'Could we skip the next session on JIRA workflows and set up a Kotlin compiler plugin for a new testing library?'

'sure.'

So we did.

It felt subversive to hack on Burst with Jake. It was like I was in high school again, skipping math class to work on a science fair project.

Another session that morning looked inessential. 'Wanna skip this one on onboarding? We can finish the Gradle integration.' We skipped it, and some other sessions. We wrote a lot of code. It's pretty amazing what two programmers can do in a few days when they're sitting together and focused.

We pushed Burst 0.x to Maven Central during the summit. We got to 1.0 by the end of the month.

I'm proud of Burst. The project is simple and useful. But I love how it was built.

Staging Data in Production

In 2022, I shipped a production build pointing to the staging API. Yes, really.

Thankfully it only reached a closed beta, but that was still enough to expose the funniest and worst part: our staging data hygiene.

We had fake records with chaotic names and frustrated test entries written by tired developers blowing off steam. Internally, that seemed harmless.

Then those entries showed up in a production build, and suddenly our "nobody will ever see this" jokes were on real user screens.

The technical blast radius was limited. The embarrassment was not.

We fixed it, but the lesson was permanent.

The 2 A.M. Hallway WiFi Test

Early in my career, I worked with LAMP/JS in what would now be called "full stack" (though I hate the term). One of my early projects was building captive portals for hotels and hospitality WiFi (I'm so, so sorry). Well the Director of the company had this ethos that everybody in the company, no matter their role, should have some involvement with customer support as a means to better understand the end-users.

This meant providing second line tech support all the way through the night, which as a software engineer not an installation technician or something else, meant I could do little but record any hardware failures, apologise to the customer, and occasionally inform hotel staff if the customer needed to be moved to a new room. One fateful evening around 2am, I received a call from a customer, they couldn't access the internet, this is about 95% of all complaints. So I run through the usual diagnostic steps, eventually asking the customer if

they wouldn't mind attempting to use the access point outside their room.

Me: I'm sorry to hear that X, in order to identify where the fault is, it would be really helpful for me, if you could try the same device in the hallway, or closer to reception.

Customer: I can't do that.

Me: Uhm, I'm sorry, why might that be?

Customer: I'm not wearing any clothes.

The 50Mb Cheeseburger

Once upon a time when I worked at Uber, it was the Friday before Valentine's Day. I was about to head out and my colleague comes over and asks if I have a minute before I leave. He was oncall and had been paged about OOMs spiking in the Uber EATS app, specifically in Miami. OOMs are often media and EATS is an image-heavy app with an image-heavy main feed, so we decided to start there to see if we could find anything obvious. We sit down and get a JSON dump of the EATS feed in Miami, cities have local feeds curated by city operations folks to give a tailored experience and recommendations for local restaurants. One by one we start opening up all the image URLs in the payload to see if anything looks off.

Then, we found it: A 50Mb image of a cheeseburger.

It was incredible and looked very tasty. But too big of a bite for the EATS app. You see,

Uber did not use a CDN, and the custom CMS Uber has internally for city ops is basically a direct line between upload to download in the app feeds. So some unsuspecting city ops person had taken what was probably the raw, extremely high quality image for a promotion they wanted to run. And it took down the app for all of EATS Miami.

The solution was in two parts:

1. Nip that image from the feed to immediately mitigate.
2. Prevent this from happening in the future. Normally, a company might solve this with a CDN, but that makes too much sense. So instead, the image loader in the android apps now has an internal network interceptor that peeks at the content-length header before it tries to download. If it's over a certain size, it just drops the response on the floor and refuses to try to load it. Probably shows a generic error placeholder in the image slot of the UI.

That patch is probably still there today in the other apps due to shared infra.

The Android of Usher

Gather close, for the shadows lengthen and the screen-glow grows dim. I have a tale that will turn your hair as white as a fresh #FFFFFF background. It is a story of hubris, of a cursed contract, and a descent into a digital madness from which few return with their sanity intact.

I was no stranger to the grand architecture of mobile development. I had wandered through many a digital estate, tending to the "Android Annexes" of the world with a steady hand. More often than not, these Annexes and their iOS Manors were twin siblings,different languages, perhaps, but the same noble blood. They shared a common decency; if a staircase led to a balcony in the West Wing of Swift, one could reasonably expect a similar ascent in the halls of Kotlin or Java. I had grown complacent, lulled into a false sense of security by years of orderly logic and predictable patterns.

Then came the Master of this specific Estate. He approached me with a smile that did not reach his eyes and a request that seemed, at the time, quite trivial.

> "My good fellow," he began, "you have tended the iOS gardens with such grace. Surely, you could step into our Android Annex? A mere trifle of a task. A small adjustment to the interface. A three-hour excursion at most."

I, blinded by my own pride and the promise of gold, signed the quote without a second thought. I had accepted the cursed contract.

I shall never forget the chill that ran down my spine when I first attempted to "Sync" the project. It was not a mere click of a button; it was a siege. The IDE let out a mechanical wail as it encountered the build.gradle file, a scroll of ancient, conflicting incantations.

I spent four hours in a feverish battle just to make the machine breathe. Every time I cleared one error, three more rose from the depths like a many-headed hydra. Dependencies from 2010, long since abandoned by their gods, fought for dominance over libraries that should never have met. I was an alchemist trying to turn lead into gold, but all I had was rusted iron and the smell of ozone. When the "Build Successful" banner finally appeared, it did not feel like a victory. It felt like a warning.

I breached the inner sanctum of the source directory, and my lantern flickered. There were no packages. No folders to separate the logic from the layout. Just a flat, featureless plain of Java files, sixty or more, huddled together like survivors of a shipwreck.

I opened the MainActivity.java. It was a monolith, six thousand lines of scrolling terror. But as I read, the true horror dawned on me. This was not the work of a decompiler or a

machine. No... this was the work of a **Patchwork Man**.

The "developer" before me had been a creature of profound, calculated lethality in his laziness. He had scoured the dark corners of the internet, hacking limbs from StackOverflow answers and stitching them onto the torso of ancient tutorials. One method was indented with tabs; the next was a chaotic jumble of single spaces. Variable names changed mid-thought, what began as user_data would, through some unholy ritual of copy-pasting, suddenly become theThingThatWorks_2.

It was a Frankenstein’s monster of stolen code, held together by the digital equivalent of duct tape and prayers. Logic didn't flow; it convulsed. I found a 400-line if-else chain that checked the system time for no discernible reason other than the fact that the original author had likely copied a "Night Mode" tutorial and forgotten to remove the parts he didn't understand. To change a single button's color was to risk the entire Manor collapsing into the tarn.

I did not spend three hours on that UI change. I spent five hours merely documenting the rot so the Master would believe my report. I had to draw maps of the madness just to prove that one could not simply "move a button" when the button’s very existence was tied to a global variable named zzz_final_dont_touch.

The Master was bewildered. "But the iOS version was so elegant!" he cried. He could not grasp that while the exterior of his house looked fine, the foundation of the Android wing was built of rotting timber and discarded rags.

I finished the task, eventually. I patched the monster and fled that repository, never to return. But I left with a scar on my soul and a new commandment etched into my heart.

Now, when a client approaches with a "simple task" and a "three-hour quote," I hold up a warding hand and speak the Vow:

"I shall not sign, nor shall I quote, until my own eyes have looked upon the source."

The App Nobody Wanted to Delete

This was back around 2009/2010 and I was working for one of the major phone manufacturers. Back then, there was a category of apps that all manufacturers wanted to ship on their devices: the social media aggregator. At this time there were public APIs to read the social media feeds from services like Twitter or Facebook, and there were plenty of apps that implemented feeds from multiple services. We had big ambitions to implement this and make it one of the more prominent features of our devices. We implemented a rather sophisticated background sync of all the accounts the user added, which made it possible to view your social media stream even when the device was offline.

I had designed a database scheme to handle this, and this schema was fairly normalized resulting in multiple tables that needed to be queried when we read the database. Each item in the feed only included one insert, but this table did reference a couple of other tables

using a foreign key. An interesting fact about SQLite is that it doesn't enforce any foreign key constraints by default. This is usually not a problem since we could make sure we always used the correct key in the application code. It is possible to enable foreign key constraints in SQLite, but it is highly discouraged as it adds a huge performance penalty on inserts and updates.

However, those of us that have studied computer science and taken the database course have usually been taught that foreign key constraints is a very important thing in relation databases, and you should always make sure your relationships between tables are enforced using proper foreign keys. This lets us do things like cascading deletes automatically and generally makes it easier to build large databases. This is, however, not how you should use SQLite. The lead developer for this app didn't know this. SQLite was new to them, but they had experience with other RDBMS, so they simply enabled this foreign key constraint and didn't think about it more.

This was part of the release of this app, and none of the testing had noticed any problems. However, we did notice that users complained that it didn't sync their social media feeds properly. Data was missing and when they forced the sync in the foreground it took a really long time to complete. I discovered that the SQLite database was opened with the following executed immediately on connection:

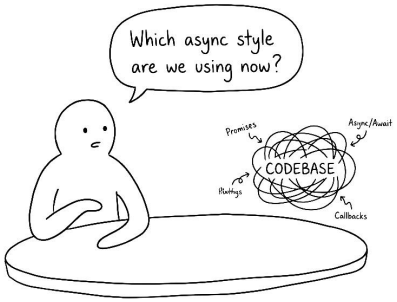
```
PRAGMA foreign_keys = ON;
```

This resulted in a huge performance hit when we synced the social media feeds. We might be inserting thousands of items, and each insert had to do a lookup on each foreign key. For some users this took so much time that the system had decided to kill the app since it

was running in the background, effectively aborting the whole database transaction and no items were inserted. Removing the foreign key constraint effectively eliminated this problem and the inserts were so fast that this problem never occurred. The engineer in charge was at first very skeptical to this solution, but after showing the number and describing how we to handle cascading deletes and such, we rolled out an update where we didn't enforce the constraint. The problem was solved and users could see their entire feed again.

The lesson here is that best practices in one domain (in this case, databases in backend system) does not necessarily translate to the world of mobile apps. We have many other examples of this in the world of Android, where enterprise patterns and solutions from Java EE was ported to Android (since we used Java), resulting in various implementations of event buses, reflection-based dependency injection, and many other solutions that worked really well in backend but was less suitable for Android.

Best practices are usually domain specific, and don't assume that you can easily apply them across domains.



The Async Library Gold Rush

Anyone who has worked on Android development for a long time know that one of the most popular things to build libraries around are anything related to asynchronous operations and concurrency in general. We know we can't do an I/O or other long-running operations on the main thread, so we've tried all kinds of ways to make sure that we do network request, file operations, or image manipulations on background thread. We have the, now deprecated, famous AsyncTask in the Android API as well as the entire Loader API (which I believe nobody except Ian Lake knows how it works). There are also a massive set of third-party libraries, and after some time the developer community sort of reached a consensus that reactive streams and RxJava was the solution to all of this. When RxJava started to gain traction, it was so popular that the conferences had the problem that so many submissions to their Call for Papers was about RxJava. It wasn't just on Android where reactive streams was popular, every other platform had their own implementation

and ways of using. RxJava was also very popular in Java backend projects, where it let you write reactive code similar to how Scala works, but without all the penalties that comes of using Scala.

When you build backend system that reacts to events, you can often end up in a situation where you receive events faster than you can process them. The first version of RxJava had a problem, it couldn't handle this very well. In the RxJava world, this is called backpressure and proper support for this was added in version 2 of the library. In the new version, you now had two base classes for implementing your reactive streams, Observable and Flowable. Flowable was added to support backpressure properly and it was one of the major new features with that version. Naturally, the developer community loves new things so there was plenty of blog posts about how it worked and why you need it. Although the official recommendation from the RxJava team was to use Observable (which didn't implement the backpressure mechanism), there was plenty of blog posts from Android developers that recommended you to use Flowable instead.

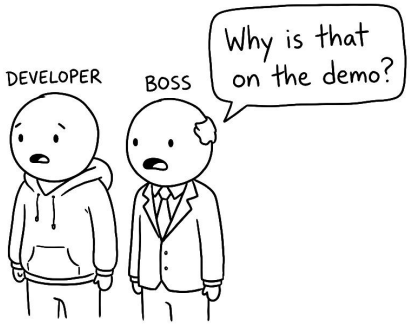
So why would this be a problem? In most cases when building an Android app, using a Flowable instead of an Observable wouldn't cause any problems. While there is a performance impact of using Flowable instead of Observable, it is usually not big enough to cause any problems. However, if you use reactive streams all over your code in many places, you might see that your app runs slower than necessary.

I joined as a contractor for a company that had an existing app. The lead developer had converted their old RxJava 1 code to RxJava 2, and the use of reactive streams was all over the app. UI events (button clicks, etc.) was translated to a reactive stream and passed downwards, and anything coming from backend was delivered as a reactive streams. The

app had a lot of network call that was aggregated until it ended up in the UI. In the upgrade to RxJava 2, all the Observables in the code had been converted to Flowable. The developer had learned about backpressure and felt it was necessary to handle this everywhere, regardless of what the source of the event was. That meant that event when there was a single network call, it was immediately converted to a Flowable. UI events was also converted into a stream with back pressure support, even though it was a stream that was connect to a single button that was disable after it was clicked.

The app worked as expected, but the Android app was much slower than the iOS version. A screen that loaded almost instantly on iOS could take a few seconds to load on Android, and nobody understood why. When I joined I didn't react on the use of Flowable everywhere, but as a developer you usually run the app in debug mode, and then this performance hit is even more noticeable. Nobody had done any real measurements to figure out why the Android app was slower, and it took a while before I figured out that a lot of time was spent emitting events into the Flowable.

By simply converting all network calls to return an Observable instead of a Flowable, almost all performance problems with screen loading was gone. This was, again, a case where a practice to solve a problem that exists in one domain (large enterprise systems) shouldn't be applied in a different domain (mobile apps). Fortunately we now have Kotlin Coroutines and Flow, so this is no longer a problem in the same way and we don't have to think about backpressure anymore.



The Bug That Wasn't

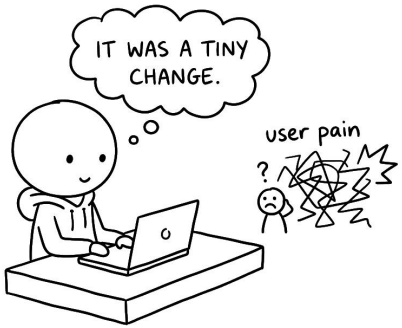
Back when I worked at an app house, I spent a frantic couple of weeks building a demo app for a client to present at a conference. It was a total rush job, but I managed to ship it just in time.

On the night of the event, I got a panicked call from my boss. "Did you do anything... weird... in the code?" I had no clue what he was talking about. He was so convinced I'd snapped that he pulled another dev to audit my work. Apparently, the app was displaying "big fat c**t" right in the middle of the showcase.

We spent a few panicked hours checking and double-checking every line. Nothing. We were all spiralling, wondering if we'd been hacked or if I'd developed a malicious subconscious typing habit.

Then the client called back, sounding incredibly sheepish. "Don't worry about the text," they muttered. "We found the source."

It turns out the demo was being run on a personal phone, and the person demoing it simply had a less than flattering nickname saved for one of their contacts.



The Christmas Release That Bit Back

This happened around Christmas, when our team usually avoids releases.

That time, the client pushed for one feature and management accepted the risk. We engineers pushed back on releasing before Christmas, but we were unheard. We did not really agree with the decision; it was forced on us.

The change looked small: update backend responses, make corresponding frontend adjustments, and ship.

But those responses were cached in the app.

Because it was pre-holiday, support capacity would be thin if anything failed. The plan was simple: if something went wrong, revert quickly.

I was one of the frontend developers on holiday watch.

Then something went wrong.

Impact was limited, and the practical fix was to clear the app cache. Users were not losing critical personal data. Still, it was a production miss, and I took it personally.

Our team had a strong quality record. I care a lot about building trust through software, so even a small failure felt bigger than the incident itself.

Looking back, two things initially seemed clear.

First, we got too relaxed because the change looked small.

Second, review accountability failed, including mine. A test existed, but it did not validate the behavior that actually mattered. I approved work I should have challenged harder.

At first we suspected a database migration test. It looked suspicious, almost as if part of it had been generated mechanically without enough human scrutiny.

But when we investigated more carefully, we discovered something different.

The migration test was actually correct. It verified upgrading the database from version Y to Z, which was the expected path.

The real problem appeared in another upgrade path: users updating from version X directly to Z. Because the two releases had happened very close to each other, that path became possible in the wild.

And we had no migration test for that scenario.

Not because someone forgot to write one, but because none of us had anticipated that upgrade combination. When we looked back, we all agreed we probably would not have

caught it anyway.

The deeper lesson was not about one missing test.

Quality is teamwork, not individual heroism.

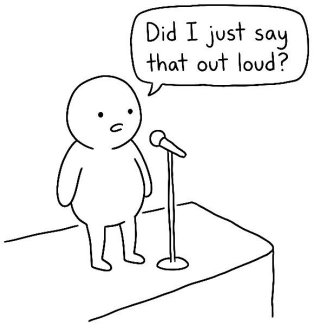
You build trust slowly, then risk losing it in one avoidable release.

We recovered, and we kept trust. We met afterward to discuss why it happened and how to avoid similar situations in the future.

Our next release, with features my team owns, went out with no bugs reported.

The reminder stuck: be accountable, review with intent, and never let calendar pressure lower your standards.

There is no such thing as a “tiny” release on the week before holidays.



The Clap That Became Crap

For the big announcement at Google I/O 2018, where we were revealing Google's official support for Kotlin, I had to prepare a presentation on the language and deliver it on the Amphitheatre in Mountain View as a keynote session in front of thousands of people in presence, and the many tens of thousands watching online.

I had decided to make it a live coding session just to add to the pressure! Anyways, the talk kicked off and I started writing code. Things were going well and the audience was really enjoying it. At some point they started clapping and I felt so relaxed. I decided to thank them by saying "I love it when people crap".

Slight slip that went down in history and for the next few years I was reminded of it quite frequently!

The Client Knows Best

A client once came to us for an app, got our estimate, decided we were too expensive, and disappeared.

Months later they came back in full panic mode. They had spent the budget elsewhere, the app they got was a disaster, and their stakeholder demo was less than a month away. It was a recipe app, and it could overheat good phones within seconds.

Four of us jumped in to patch it up.

Very quickly we realised what kind of month we were about to have. Delays were being handled with loops instead of timers. Buttons were not really buttons, just boxes with text on top. Every few hours someone would find a new horror, say it out loud, and the rest of us would laugh for five seconds before going back to damage control. For a while, it was basically Friday at 4 PM every day.

We somehow got it stable enough for the demo window, which should have been the happy ending.

It wasn't.

On the final showcase day, our CEO was with the client while I stayed back at the office. Then my phone rang. He was panicking.

"They're only seeing a black screen."

Two or three of us immediately dropped everything and started checking builds, servers, devices, config, all of it. For ten or fifteen minutes it was absolute chaos and a lot of "it works on our phone."

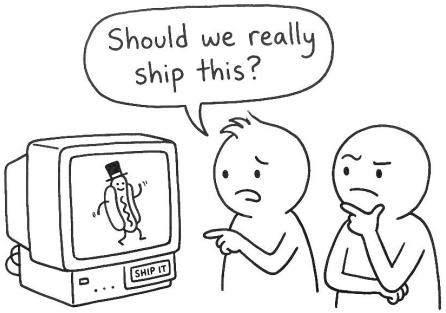
Then we found the problem.

The client had set both the text and the background to black on the production server. The app was fine. It was just black text on a black background, perfectly invisible.

That was the whole emergency.

I called the CEO, explained what happened, and later he came back to the office with a huge box of donuts to apologize for the tone of the panic call.

That project taught me something: the scariest production bug is sometimes your client.

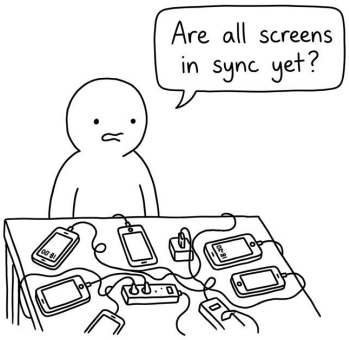


The Easter Egg That Wasn't

The Android dialer allows apps to register "secret codes" using the `Telephony.SECRET_CODE` action in their manifest. When a code is registered, dialing `*#*#xxx...#*#*` (where x's are numbers) will invoke your app. In the weeks leading up to Android 1.0, I figured I could use this to create an easter egg that would list all the folks who worked on the first release of Android. Although I forgot what the secret code was, the list of names was hidden in the PNG metadata of one of the launcher icons, and the code itself was in launcher and looked like a comment thanks to Java's unicode escape trick.

I showed this easter egg to a few folks, and it took on a life of its own. It eventually became an actual entry in the Settings application, with an animated scroll built using `WebView` (it was out of my hands at that point). But folks started worrying that some

names would be missed, so it was decided to kill the idea entirely. It was probably a good idea to do so, to ensure fairness for everyone who contributed to Android, but I kind of wish I had kept it a secret.



The Origins of Blinkendroid

This story takes us back to a time before the very first Droidcon. I was a student in Berlin and part of the Berlin Android Stammtisch, a growing developer group that predated GDGs.

When the first Droidcon was announced, we brainstormed ways to bring the community closer together. Since every Android developer likely had a phone or two, we wanted to leverage that collective hardware.

Our initial idea was ambitious: a fully functioning massage chair made of phones vibrating in patterns. It sounded great—phones vibrate, and with enough of them, we could pull it off. But the logistics of modifying a chair to hold all the devices, combined with the realization that no one would want strangers sitting on their expensive phones, led us to reconsider.

We pivoted to connectivity, imagining a matrix where each device acted as a pixel in a larger display. This concept was inspired by the Blinkenlights project, where building windows served as pixels. Naturally, we borrowed the name: Blinken-Droid[1].

Since Android was a new and niche operating system at the time, we had to experiment to discover its capabilities, specifically regarding low-latency connections. Early tests with speaker and microphone communication proved physically painful and were quickly abandoned. Relying solely on TCP crashed the WiFi and increased latency. Eventually, we settled on a hybrid infrastructure: a server/client model providing video via TCP, synchronized by a UDP heartbeat.

After publishing the first version and gaining TV attention during Droidcon, we convinced Google to bring us to Google Developer Day in Munich. We set our sights high: a Guinness World Record. However, reading the official rules sparked a moment of panic—we needed two independent expert witnesses from official institutions to verify the attempt. We managed to find the right people, arranged 72 phones on a table, and successfully displayed several synchronized videos to secure the record. We even seemed trustworthy enough that we ended the attempt with three extra phones. You can watch the Guinness World Record on YouTube[2].

The moral of the story? Go out and have fun. Build things with people you like. Get creative, challenge yourself, and you might just make some great friends along the way.

The Project With No Tickets

Working for an agency, circa mid-2010s, working on-site at the customer's offices (a UK-based retail bank) to be integrated directly with the dev team. Most people on the project were contractors, including the scrum master. There were two of us from the agency as devs on this project (me on Android, someone else on iOS) and we also provided a project manager

The first sign of trouble was that the project manager from our side was asked to stop attending team meetings.

I don't recall the reason, but our PM was not a trouble maker or a nuisance to them. It felt out-of-the-blue to us.

The next thing is we noticed the rest of the team piling in to the meeting room next to our

bank of desks yet we had no event invites nor did anybody prompt us to join them. And this kept happening every day.

No tickets were being assigned to us.

If we asked for a task, an excuse would be given as to why there were no tasks for us to do. Asked what we should do, it was met with a shrug.

But we were in a busy office environment. And working for an agency, the agency was conscious of its image to the rest of the business.

So we had to pretend to be busy, even though we had no actual work to do and shut out of any influence.

This went on for a few weeks, then the agency decided it was no worth it anymore, so we were removed from the project.

And why might this have happened? Well, we never knew for sure, but the hypothesis was that the contractors running the team were fearful of the agency coming for their lucrative roles.

The Tale of a Very Expensive Analytics Event

Many things can go wrong in an app built around a map solution like Google Maps or Mapbox. There are performance considerations, UX challenges, expensive APIs, and more. These apps can become quite complex relatively quickly, and the more features you squeeze in around the map, the more that complexity grows.

With all that complexity comes the need for observability — seeing what your (anonymised) users are doing in the app so you can make informed product decisions later. That's where analytics services come into play, along with the tracking events you send to them.

Of course, these services are rarely free, and they usually factor event volume into their pricing.

So it happened that a PR with a feature was created, reviewed (by me), approved, and

merged. It introduced a seemingly harmless use case class that was supposed to track when one of our main UI components — the entry point to most of our features — showed up as a result of the user clicking an object on the map. However, we needed the device's location to calculate the distance between the user and that object, to be included in the event. We had another use case class responsible for providing location data to other parts of the app, as one usually does. So we grabbed the location, ran some calculations to derive the data we needed, and logged the event. The feature shipped, and everyone was happy.

About a month later, a PM (if memory serves) tagged the Android team on Slack, asking about tens of millions of similar analytics events flooding our Amplitude pool — causing roughly €10,000 in extra costs over the past month.

After the initial panic in the team, we quickly blocked the event in Amplitude to prevent further costs, and began investigating what went wrong. Fortunately, we had a clear lead, the rogue event, so we could easily pinpoint the exact line in the exact commit.

It turned out that the naming of our location provider's functions was ambiguous:

`getUserLocation()` vs. `fetchUserLocation()`

One provided a continuous stream of location updates; the other, a single value fetched once.

Both returned RxJava types that we could chain our calculation logic onto the exact same way. The code in the PR used `getUserLocation()` — the stream — which meant the analytics event fired on every single location update from the moment the user opened the app's main screen and tapped on an object on the map until they closed the app. For every

user. Every few seconds. For a month.

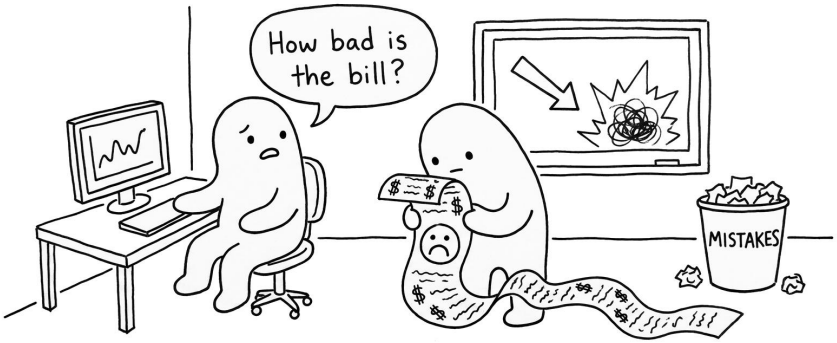
Even though the fix was straightforward, the harm was already done, and there were multiple hundred thousands of users on the app version with this bug. On the bright side, we already had force updates in place.

One line of code. Ten thousand euros. Multiple lessons learned.

- When subscribing to reactive streams, ask yourself: "How many times will this emit?"

- When you see reactive code, trace streams to their sources.

- Set up anomaly alerts for analytics, as catching one in a day is a bugfix, catching one after a month is a budget conversation.



The Tale of the Very Expensive CI Build

Once upon a time, at a bustling tech company in the heart of the UK, lived a fresh-faced Junior Android Developer named... well, let's just call him Dev-y.

Dev-y worked on a mobile app, and his team used a cranky old relic of a tool for building it, testing it & delivering it, called Jenkins. This Jenkins instance, for very questionable reasons, actually lived in a on-site "box" in the corner of the office and, for even more questionable reasons, was plugged right into the World Wide Web!

One chilly winter's afternoon, Dev-y was zipping along, coding like a champ! But in a moment of rush and without thought, committed the greatest of developer crimes: he checked into the codebase his very secret AWS key! This key, although constrained still had full access to Amazon's giant fleet of cloud computers (called EC2s, this will be important later)

The Jenkins box, doing its job, happily checked-out Dev-y's code locally, key and all & ran the pipe. The build passed, and Dev-y went home, happy with his day of work, he snuggled up warm and cosy, dreaming of green builds.

The next morning, Dev-y walked in, grabbed his coffee, and settled down ready for another full-packed day of tackling bugs & learning more about this wonderful world of code!

However his serene morning was very soon derailed as...WHOOSH!

Two serious-looking people from the IT security team rushed over. They looked stern & serious, filling Dev-y with fear.

"Come with us, young Dev-y," they whispered dramatically.

Dev-y's heart went ker-thump-ker-thump as he was whisked through the bowels of the building, areas of the company he had never seen before & never knew existed.

The head of IT security finally ushered Dev-y into a small eerie room that felt like a interrogation room & explained the terrifying news:

"Something has happened"

"Last night a bunch of EC2 instances were spun up to dig for sparkly digital treasure" (that's 'crypto' kids!)

"Resulting in an astronomical AWS bill that we have to pay"

"And key that performed these operations... is yours!"

"Do you know anything about this?!"

Dev-y was shocked! He honestly had no idea. He did not do this. He had no idea how to even do those things. How could his key have done these things?!

The investigation swiftly began and it did not take long at all for the security professionals to figure out the true wicked plot & unravel the mystery to verify Dev-y's alibi.

"Ok we found the problem, you checked in your AWS key & the Jenkin's version your team is using is like a chain-link fence made of spaghetti."

It turns out that during the night, evil hacker's of unknown origin that already knew of the Jenkin's box, found something new & very interesting delivered straight to their feet.

They slipped onto the public Jenkins dashboard & gained access, probably, with some admin credentials that were never changed from the default, they then saw the most recent 'checkout' of the app's codebase contained a very tasty AWS token…

The hackers raced against the clock, using their new found token to fire up every large computer they could find to MINE, MINE, MINE, causing an emergency mega-bill to pile before the security team could hit the big red panic button & lock out Dev-y's AWS account!

For just those few moments the evil hackers were able to accrue a 10,000 pound AWS bill for the company, and an unknown amount of (essentially free & untraceable) 'cryptocurrency' before the heist was abruptly halted.

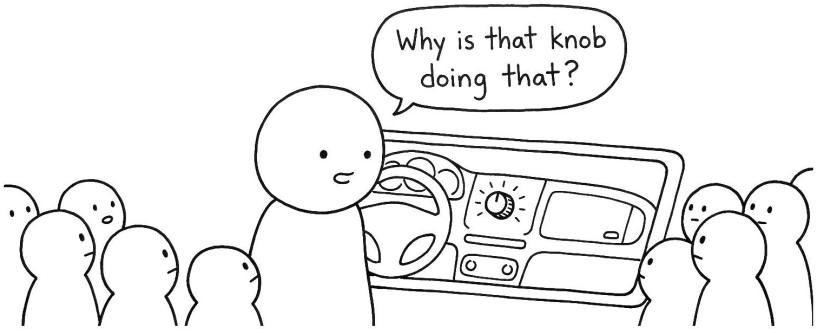
So, what happened to our hero, Dev-y? Was he fired? Forced to sell his his liver to pay the bill? or some other unthinkable punishment?

No! The company's wise elders knew that even the best developers make mistakes (especially learning juniors). Dev-y got a small, gentle "tut-tut" from his boss and a very long, very educational seminar about “The Dangers of Checking In Secrets”, even to private repos.

From that day forward, Dev-y became the most cautious developer in the land! He checked his code twice, then thrice, making sure he never, ever slipped a precious secret into the into any codebase ever again!

The moral of the story, my little coders, is this: always treat your secret keys like gold and your old build tools like, well, tools created by other people, that too themselves could make mistakes & make sure to keep said tools up-to-date!

But most of all never check in secrets!



The Tech Lead and the Embarrassing Volume Knob

Picture this: It's 2017, and I was the Tech Lead at German Autolabs. We'd poured over a year into building the Chris companion app, integrating it with our super complex, Automotive Voice Assistant (AVA) platform. This thing managed everything: navigation, phone calls, SMS, and WhatsApp. We needed perfect state awareness to keep drivers safe, and we were incredibly proud of the clean architecture we'd built.

Our big moment came during a routine test drive with our Product Manager. Everything should have been flawless.

Instead, the app entered a bizarre, catastrophic state. The PM would turn the volume knob, and the music would randomly go silent without a pause command! Then, if they tried to use a voice command, Chris was stuck, trying to pause the music that was already paused. The entire interaction flow crumbled. The feeling in that car was pure embarrassment. We

were the tech experts, leading the development of this advanced platform, yet we couldn't explain why a simple volume knob was breaking our whole system in certain cars.

After days of intense debugging, testing two different Volkswagen models side-by-side, we found the ridiculous, undocumented culprit. It wasn’t a flaw in our complex AI code; it was the car’s volume knob.

Here was the absurdity: One Volkswagen model behaved correctly when muted, simply silencing the audio. But the other silently sent an unprompted Bluetooth PAUSE command to the phone before it muted the audio.

This invisible, phantom command—triggered by the simple act of our PM turning the volume down—was corrupting the state of our entire AVA platform. A year of secure, complex development, defeated by an arbitrary engineering choice hidden deep inside a car’s infotainment Bluetooth firmware. We realized the embarrassment of the test drive was just the beginning of the hilarious absurdity of debugging hardware.

Twas the night before planning

Twas the night before planning, and all through the team
There were whispers of mystery about an upcoming scheme;
The coffee was brewing in the kitchen en masse,
With hopes that a rich client would soon part with their cash;
The meeting room was prepped, and the exec were excited,
For the secretive "special guests" that were cordially invited;

Suddenly silence took hold, and in with a flash,
Eight lots of legs entered the room in a dash;
The developers awaited their fate by their desks,
And the stillness was deafening, as they stood statuesque;
"What are we building? For how much? And for why?"
A weary iOS developer let out in a cry;

An age passed until the room's shadows animated,
And it was finally clear the agency's new fate was dictated;
Muffled congratulatory exchanges were observed,
And bursting out of the room the beaming exec did emerge;
They shook hands with the client, and showed them the door,
As they departed, hi-fives were exchanged while the CEO roared;
"Everyone, make your way to the meeting room at pace"
"We have news" he exclaimed with a wry smile upon his face;

A scampering team gathered, with intrigue in their hearts,
As a presentation by the exec was all ready to start;
The slides whizzed by but the team's joy slowly turned to frustration,
With unease growing about the strange content of the hurried narration;
Gasps were let out and jaws hit the floor,
Some left the meeting as they could take it no more;
The room's tone was muted, as by the end of the epilogue,
The team had just learned they were building a "Facebook for Dogs".

You Know Him, Right?

At one of my first Droidcon Berlin events, I was on my own, pretty naive, and not naturally social.

I was trying to meet people, but half the time I didn't even realize who I was talking to until much later.

At one booth, a confident, friendly guy started chatting with me about what I did and what tools I used. We spoke for several minutes, laughed, and had a genuinely good conversation.

At one point he asked whether I used JetBrains products.

I still didn't clock who he was. I just thought it was a sales pitch.

Later Ash asked, "Do you know who that was?"

I said no.

It was Hadi Hariri.

Fast-forward to the evening: beers, small groups, pub hopping, random conversations with people I'd just met.

Again, I was happily talking to someone for ages without realizing who he was.

Months later, during a job interview, the interviewer asked who I'd met at Droidcon. They were a huge fan of someone and asked if I'd met him.

Name didn't register, so I looked him up.

Then it hit me.

I had spent part of that night in the pub chatting with Chet Haase without realizing it.

The interviewer went silent.

I didn't get the job (not because of that), but the pattern was clear:

I'm apparently very good at networking with famous Android people, and very bad at identifying them while doing it.

Resources

An Open-Source Nightmare at Midnight (p. 10)

[1] github.com/skydoves/colorpickerview

How a Bot Can Delete Your GitHub Repository (p. 22)

[1] cloud.google.com/blog/products/identity-security/enabling-keyless-authentication-from-github-actions

[2] github.com/ashdavies/playground.ashdavies.dev/pull/1676/files

[3] developer.hashicorp.com/terraform/cli/commands/state/mv

[4] developer.hashicorp.com/terraform/language/moved

[5] github.com/ashdavies/playground.ashdavies.dev/pull/742

LOG.WTF() (p. 52)

[1] developer.android.com/reference/android/util/Log#wtf

Party Like It's 1995 (p. 58)

[1] developer.android.com/reference/android/hardware/SensorManager.html

The Origins of Blinkendroid (p. 89)

[1] de.wikipedia.org/wiki/Projekt_Blinkenlights

[2] youtube.com/watch?v=Ts8vIUML37Y

About the Authors

This book would not exist without the people behind it.

What started as a joke, somewhere between late-night conversations and conference hangouts, quickly turned into something real thanks to a group of developers who were willing to share their stories, the funny ones, the painful ones, and the ones that probably should not have happened in the first place.

Each story in this book comes from lived experience… or at least something close enough to it.

To keep things interesting (and to protect the innocent), the stories are not attributed. You are free to guess who wrote what.

A sincere thank you to everyone who contributed their time, creativity, and questionable production decisions to make this possible.

The Authors

Alexandra Davies

Andy Barber

Ash Davies

Ben Kadel

Ben Weiss

Chet Haase

Chris Ward

Cicero Hellman Ratti

Ed Holloway-George

Eliza Camber

Erik Hellman

Greg Fawson

Hadi Hariri

István Juhos

Jaewoong Eum

Jake Wharton

James Cullimore

Jesse Wilson

Lucas Villa Verde

Marc Reichelt

Mat Rollings

Matthias Geisler

Mohsen Mirhoseini

Muriel Kamgang Mabou

Niall Scott

Nicole Terc

Romain Guy

Scott Alexander-Bown

Sergio Sastre

Zac Sweers

Zachary Powell