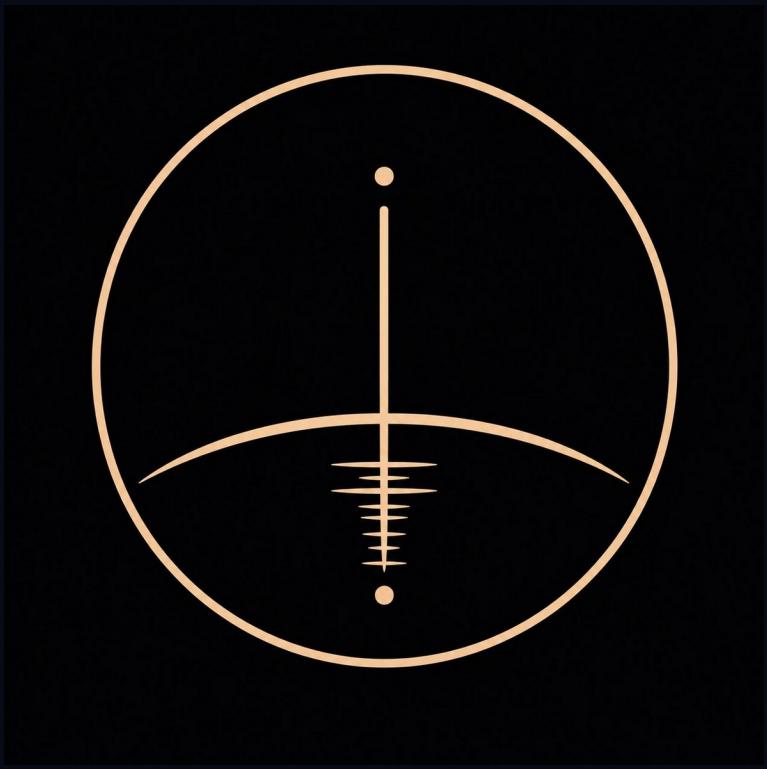

The App I Almost Didn't Build

LESSONS, MISTAKES, AND HARD-WON WISDOM FROM BUILDING MY FIRST APP

Author's Note

This isn't a step-by-step manual, and it won't teach you FlutterFlow line by line. If you're looking for a technical course, this isn't it — there's very little tech in these pages, and no instructions to follow.

This is the story of what actually happened when someone with no programming background decided to build a native mobile app anyway. There were bugs, tears, laughter, arguments with AI, and moments where I honestly questioned my sanity. There were also moments when I looked at my screen and thought, "Holy... I can actually do this."

Scattered throughout the stories, you'll find Field Notes — the actual Do's and Don'ts I learned the hard way, pulled straight from the messy, real-time notes I kept while building the app. They're not theory. Every single one cost me at least one bad afternoon.

My hope is simple: when you close this book, you won't walk away with a technical manual. You'll walk away believing you're capable of doing the thing you've been putting off — whatever that thing happens to be.

Never say no to yourself. Life will do enough of that on its own.

— *Dana*

PART I: Before the First Click

"*Every app begins long before the first button is pressed.*"

The App I Almost Didn't Build

I stared at the screen for what felt like forever. Not because I was thinking — because my brain had temporarily left the building.

Panels. Menus. Icons. Buttons. More buttons.

Somewhere on YouTube, people kept saying FlutterFlow was "beginner friendly." I looked back at my monitor and said the only thing that made sense.

"*What the fuck is this?*"

That wasn't frustration. Not yet. It was pure disbelief — because just a few weeks earlier, I'd been convinced that building an app was going to be easier than this.

Where It Actually Started

Before any of that, though, there was a much smaller, much quieter moment that started this whole thing. It didn't happen in FlutterFlow. It didn't happen with AI. It happened in my own head, on an ordinary day, when I realized something about the way I felt every time I tried to relax.

Life had gotten noisy. Not just the noise you hear — the kind that follows you everywhere. Responsibilities, work, stress, notifications, a mind that never quite got a place to land. I didn't want another productivity app telling me to optimize something. I definitely didn't want another meditation app with three thousand sessions locked behind a subscription, where I'd spend the first ten minutes just trying to pick one.

That's really where it started. I wanted a reset button. Something I could open for five quiet minutes before going back to my day.

Then I started actually looking at what already existed, and I noticed something that bothered me. Every wellness app seemed to want more of me. More notifications. More streaks. More challenges. More content. More reasons to keep me opening it, again and again, like the app's real goal was my attention, not my calm. Standing in front of one of those apps felt exactly like standing in the cereal aisle at the grocery store — forty boxes, all promising to make my morning better, and I'd just stand there completely frozen, having no idea which one to grab. I didn't want to build another box for that aisle. I wanted to build the opposite of the aisle.

I wanted someone to open the app, choose something immediately, breathe, listen to a little music, watch something quiet, look at a few peaceful images, and leave. No decision fatigue. No dashboard. No fifteen categories to sort through before you're allowed to relax. The goal was never to keep someone inside the app. The goal was to help them leave it feeling a little better than when they opened it.

That one idea — no cereal aisle, no subscription, no noise — became the entire philosophy behind everything I eventually built: the breathing exercises, the music, the immersive nature scenes, the quiet gallery. Every one of those features exists because I kept asking the same question every time I was tempted to add something new: does this simplify the moment, or does it just add another box to the shelf?

I think that's also the real reason I never fully quit, even on the days I wanted to. This wasn't a hobby I'd talk myself out of. It was the thing I actually needed for myself, built by the only person who happened to be free to build it — me. It's a lot harder to abandon something when you're not just building an app. You're building your own reset button.

Social media had done a fantastic job selling me a dream about how to build it, even if the why had already been decided. Every other video sounded the same: build an app without coding, AI does everything for you, launch this weekend. It sounded amazing. It also sounded believable. AI could already write emails, answer questions, generate images. Why couldn't it build an app?

So I did what thousands of people probably did. I believed the hype.

My First Wrong Turn

The first platform I tried wasn't FlutterFlow. It was one of those AI app builders — the kind all over Instagram and TikTok. I described what I wanted, clicked a few buttons, and something that actually looked like an app appeared. For a moment, I felt like a genius.

Then I started asking questions. Can I publish this to Apple? Can I publish this to Google? Can I add in-app purchases? Do I own the code?

The answers slowly deflated my excitement. As long as I kept paying the monthly subscription, everything worked. Stop paying, the app stopped working. I wasn't building my own app — I was renting one. It wasn't a native app. I didn't own the code. And that, it turned out, makes publishing and monetizing nearly impossible.

That lesson cost me money. But it forced a question I should have asked from day one: what exactly am I building? There's a huge difference between a native mobile app and something that only looks like one. I had no idea that difference existed. Now I couldn't ignore it.

Back to Square One

So there I was — no app, no roadmap, no programming experience, just an idea that refused to leave me alone. AI suggested FlutterFlow: a real native-app builder, beginner friendly, works well with Firebase. Sounded exactly like what I needed.

Then I opened it. You already know what happened next.

I wish I could tell you I watched one YouTube video and everything clicked. It didn't. Every tutorial spoke a different language: widget tree, state management, backend query, container, component, Firestore, custom widget, responsive layout. I wasn't learning FlutterFlow. I was trying to learn the vocabulary of people who already knew FlutterFlow.

I paused the video, sat back, and laughed — not because it was funny, but because I genuinely had no idea what that guy had just said.

The Best Decision I Didn't Know I Was Making

That afternoon, I stopped trying to learn the way everyone else seemed to be learning. Instead of another tutorial, I took a screenshot of FlutterFlow and opened ChatGPT.

My first question wasn't impressive. It wasn't technical. I just asked, "Can you tell me what I'm looking at?"

That changed everything. Suddenly I wasn't trying to understand the entire program — just one screen, one button, one panel. It felt manageable. I asked another question. Then another. Sometimes I asked the same thing three different ways because I still didn't get it.

Being a beginner is uncomfortable. Everything takes longer. You celebrate things experts don't even notice — the first time I discovered Ctrl+Z, I felt like I'd found fire. But that discomfort is the toll. You pay it, and you get to keep going.

There was another conversation happening while I learned FlutterFlow — one with myself. "You're never going to figure this

out. This is for programmers. You're wasting your time." Some days I believed it. Some days I almost closed the laptop for good.

The only reason I didn't is because this app had become bigger than software. It represented freedom — a way out of a situation I no longer wanted to stay in. When life gives you a reason big enough, you get surprisingly good at surviving difficult days.

So I made myself a deal: I didn't have to finish the app today. I just had to understand one more thing before I closed my laptop. One button. One setting. One tiny step.

That was enough. And, strangely, those tiny steps eventually became an app.

Field Notes — Before You Start

DO understand what you’re actually building before you commit — a "native app" and an AI-generated web wrapper are not the same product, and only one of them lets you own your code and publish freely.

DO expect the vocabulary to be the first real obstacle, not the software itself. Screenshot everything you don't understand and ask.

DO measure progress by "one more thing than yesterday," not by how far away the finish line still looks.

DON'T assume "beginner friendly" means *easy* — it means possible.

DON'T wait to feel ready. Ready is a moving target that never arrives on its own.

My Four-Year-Old Taught Me How to Build Apps

If someone had told me one of the biggest lessons I'd learn about software would come from a four-year-old, I would have laughed. Then I would have asked if they were serious.

Long before FlutterFlow, my son was sitting at my computer, clicking on everything he could find. If you've ever met a four-year-old, you know that a button which exists is a button that must be pressed. I kept saying "No — don't click that," the way parents do, certain I was protecting my computer.

Then he clicked on something I'd never noticed before. A camera app opened — one I didn't even know existed, with filters and features I'd never seen. We spent the next hour taking ridiculous pictures and laughing until our stomachs hurt.

I'd owned that computer for years. The software had always been there. I had simply never clicked on it.

Children don't assume they're going to break everything. Adults do. Children explore; adults hesitate. Neither approach is wrong, but curiosity almost always learns faster than fear.

Years later, I opened FlutterFlow for the first time. Rows, Columns, Containers, Widgets, Properties — it looked like someone had dumped a box of LEGO bricks on the floor without including the picture from the front of the box.

I remembered my son. He wouldn't have stared. He would have clicked something.

So I clicked something. Nothing exploded. Good start. I clicked something else. The screen changed and I had no idea why — also good. Then another. And another. Sometimes something useful happened. Sometimes everything looked worse. Sometimes I spent twenty minutes trying to undo whatever masterpiece of destruction I'd just created.

But every click taught me something, because experience always teaches faster than imagination. You can wonder what a button does for an hour, or you can press it and find out in three seconds.

Click. Observe. Ask questions. Repeat. That became my entire learning philosophy. I wasn't trying to master FlutterFlow — I was trying to understand one more thing than I understood yesterday. Some days that “one thing” was tiny: the difference between a Row and a Column, or where AI kept insisting a setting existed that I

could never find. Some days my entire victory was discovering one checkbox.

If you're an experienced developer, you're probably smiling — you don't even notice those things anymore. Beginners do. And every tiny discovery should feel enormous, because it is.

I think adults quit too early because we’re embarrassed to be beginners. A child can ask "why?" three hundred times a day without feeling stupid. Adults apologize before asking one question. There are no dumb beginner questions. Only expensive assumptions — every time I assumed I understood something without asking, it cost me hours. Every time I admitted "I don’t understand," I learned faster.

That's part of why AI became such a good teacher for me. It never rolled its eyes when I asked the same thing five different ways. Try that in a crowded classroom.

Eventually, FlutterFlow stopped looking scary — not because it got simpler, but because it got familiar. Those are not the same thing. The software didn't change. I did.

If I could sit down with the woman who'd just opened FlutterFlow for the first time, I'd tell her exactly what my son taught me without knowing it: stop trying to understand the whole program. Learn one button today. Learn another tomorrow. One day you'll realize you're no longer trying to figure out where everything is — you're building.

Maybe that's the biggest lesson a four-year-old ever taught me. Curiosity isn't something you grow out of. It's something you accidentally replace with fear. The good news is, you can always get it back — sometimes all it takes is clicking the button you were afraid to click.

Field Notes — Getting Oriented

DO click around in a duplicate or test version of your project before you trust yourself in the real one — exploring is how you build a mental map of where things live.

DO keep a running, dated note of what's confirmed working as you go, and treat it like a small handoff document — credentials, IDs, what's deferred and why. Future-you will not remember why a setting was changed.

DON'T compare your Day 1 to someone else's Year 10. You're not behind. You're early.

The First Time It Felt Real

There is one screenshot I'll probably keep forever — the first version of my welcome page. If you compared it to what's in the app today, you'd barely recognize it as the same project. The colors changed, the layout changed, the navigation changed. Almost everything changed.

But I still love that first screenshot, because it wasn't about design. It was about belief.

For weeks my world had been nothing but containers, rows, columns, padding, and disconnected pieces — like dumping ten thousand puzzle pieces on a table without knowing what picture they made. Progress in software is often invisible. You're improving; you just can't see it yet.

Then one afternoon I pressed Preview, and my welcome page appeared. It wasn't fancy. It wasn't finished. But it was mine. For the first time, my brain stopped seeing FlutterFlow and started seeing my app.

"Holy… this is actually happening." My eyes filled with tears — not because a welcome screen is emotional, but because for the first time I believed I could actually finish this. Hope is imagining something. Belief is having enough evidence that the imagination might become real. That welcome page was my evidence.

I used to think motivation comes before action. My experience was the opposite — action created motivation. Every tiny thing I finished made me want to finish another. It's like climbing a mountain: at the bottom, the top looks impossible. Halfway up, you look back and realize how far you've already climbed. The mountain didn't shrink. Your perspective changed.

About twenty minutes later, I found another bug. Reality has excellent timing.

Every Ghost I Chased

Once I stopped expecting perfection, problems stopped feeling like emergencies and started feeling like appointments. I started calling my bugs ghosts — because that's exactly what they felt like. Invisible, unexplainable, and somehow fixed by changing something completely unrelated.

One ghost made my back button stop responding. Hours of troubleshooting later, the cause turned out to be embarrassingly

small: the Row containing the button was sitting one level too high in the widget tree. One placement. Hours of confusion.

Another time, my image carousel refused to show a single picture — but only in the browser. Firebase looked fine. Permissions looked fine. Everything looked fine, twice. Then I ran the app on my actual phone and every image loaded perfectly. The browser had been blocking the images with something called a CORS error. Nothing in my app was broken. The browser was lying to me.

Then came the yellow-and-black overflow warning — FlutterFlow's way of screaming that your layout is a disaster. AI suggested adjusting the width of my Column. There is no width setting on a Column in FlutterFlow. I said so. Ten minutes later, it suggested the same thing again. After a full afternoon of changing containers, padding, and alignment, the actual problem turned out to be a title that was simply too long. I shortened three words. Everything fit.

And the smallest one of all: a feature that flatly refused to work, for no visible reason, until I noticed one missing slash. One character. Hours of my life.

Those experiences slowly rewired how I troubleshoot. Now, when something breaks, I don't assume something complicated is wrong. I ask: what's the dumbest possible explanation? Did I miss a checkbox? Is there a typo? Is the title too long? More often than you'd think, the simplest answer wins — and every time I solved one of these small, ridiculous problems, I became a little less afraid of the next one.

Field Notes — Chasing Ghosts

DO change one page or one setting at a time, test it on a real device, and only then repeat the fix elsewhere.

DO read the actual error in the run log before changing anything — it usually names the exact widget and line.

DO test on a real phone before trusting the browser preview. Browsers block things (like images via CORS) that work fine on-device.

DO set explicit width and height on anything nested in a Stack — "fill the screen" only works reliably when the Stack itself has a bounded size.

DO set a tappable icon's alignment explicitly and make sure it's the last child of its Stack. That's usually why a button “stops

responding” for no reason — it's sitting under something else, not broken.

DO use the Safe Area toggle for spacing around the notch instead of guessing pixel values, but scope it to the small element that actually needs it. Slap it on a full-screen background image and you'll shrink the image away from the edges instead.

DO treat iPhone-vs-Android behavior differences as device and layout issues, not proof something’s broken. The same button can behave differently on different devices without either version being wrong.

DON'T panic at a wall of red text in the run log — most of it is harmless warnings.

DON'T assume two pages behave identically just because one was duplicated from the other. A stray slash, a container size, a layout type — small differences cause different bugs.

DON'T keep guessing pixel values blindly. If you’re guessing, switch to the adaptive/responsive tool instead.

DON'T go hunting through nested menus on a guess more than once or twice. Step back, search the exact error text, or take a fresh screenshot before the fourth guess.

PART II: Welcome to the Chaos

"*You are not going to believe what happened next.*"

My AI Coworker

At some point, AI stopped feeling like software. It became a coworker — a really smart one who occasionally walked into the room, confidently explained something completely wrong, and smiled like nothing had happened.

People imagine building an app with AI looks like typing "build me a calming wellness app" and waiting for the App Store notification. Reality looked more like this: the back button doesn’t work. Let’s check the navigation — nothing. Let’s check the action — nothing. Let’s rebuild the button — still nothing. Three hours later, the Row was simply in the wrong place in the widget tree. Three hours. One Row.

That's when I learned something important: AI is incredibly good at generating possibilities. Humans are responsible for deciding which possibility actually makes sense.

One thing kept happening that I couldn't ignore. We'd spend hours solving a problem, fix it, celebrate, move on — and an hour later, on a different page, the same issue would reappear. AI would confidently suggest something different than the fix that had already worked.

"Dude," I’d say. "We already solved this."

"You’re right," it would answer.

After the fifth or sixth time, I stopped just laughing and started paying attention, because I realized something: AI wasn't remembering my project. I was. That made me more valuable than I'd realized. The more I thought, the better AI performed. When my questions improved, its answers improved. When I blindly copied whatever it suggested, we went in circles.

The clearest example of this was The Video Problem. My welcome page had a looping video with calming background audio. Separately, both worked beautifully. Together, they became a nightmare — every time I left the page, the video stopped but the music kept playing, like leaving a movie theater and having the soundtrack follow you into the parking lot.

We spent nearly eight hours on it. Different actions, different widgets, different logic — every new idea solved one thing and broke another. Eventually, my AI coworker admitted defeat: "I think you should contact FlutterFlow Support."

I laughed so hard I had tears in my eyes. The technology everyone said would replace humans had just sent me back to a human.

The actual fix didn't come from another prompt. It came from stepping back and asking a completely different question: what if the video and the sound were never separate in the first place? I already used CapCut. I merged the audio into the video file, imported one file, and used one player. No syncing. No orphaned sound. Problem gone.

Eight hours, solved by refusing to keep asking the same question a ninth way.

That day taught me the real difference between AI and a human in this partnership: AI is excellent at solving the problem you hand it. A human is still the one who gets to ask whether that's even the right problem. I don't think AI replaces creativity — I think it amplifies it, but only once you bring some to the table.

By the end of the project, I wasn't following AI's instructions. I was arguing with it, catching its mistakes, and occasionally surprising it with an idea it hadn't considered. That's a completely different relationship than the one I started with — and a far more useful one.

Field Notes — Working With AI

DO verify a claim — a setting supposedly existing, a menu supposedly in a certain place — against your actual current screen before following multi-step instructions based on memory (yours or the AI's).

DO keep your own notes on what's already been tried and what actually worked. AI's memory of your project is only as good as what you remind it of.

DO ask "am I solving the right problem?" after the second or third failed attempt, instead of the tenth.

DO check the route path first when a video or audio action stops working — a missing slash or wrong route name was the answer more times than I want to admit. Compare against a page where it already works.

DO distinguish "this needs more time to propagate" from "this is actually misconfigured" by checking a concrete fact before defaulting to "just wait longer." AI will suggest waiting long after waiting stopped being the answer.

DON'T treat AI as the expert in the room. Treat it as a very well-read coworker who needs your judgment as much as you need its speed.

DON'T apply the same fix everywhere the moment it works once — confirm it holds on one page before repeating it across five.

Starting Over Wasn't Failure

If there's one thing I got surprisingly good at while building this app, it was starting over.

That sounds depressing. It wasn't. It became one of my biggest strengths.

Most people imagine progress as a straight line — build, improve, finish. Mine looked more like: build, break, delete, rebuild, break something else, delete, rebuild again, question every life decision, drink coffee, repeat. I lost count of how many pages I rebuilt from scratch rather than trying to untangle a knot I'd accidentally tied.

Think of a LEGO house. Halfway through, you notice one brick near the bottom is wrong. You can carefully remove pieces until you find it, or you can take the top off and rebuild it correctly. I almost always chose the second option. It might look inefficient. But every rebuild was faster than the one before it — the first time a page took six hours, the second time three, and eventually I could rebuild the same page in thirty minutes. Not because I'd memorized it. Because I finally understood it.

I also learned, embarrassingly late, that Ctrl+Z existed. I'm not exaggerating — I had no idea. The first time someone told me to press it after I accidentally deleted something, I thought they were joking. Then everything came back, and I stared at the screen like I'd witnessed witchcraft. Later I used it at work after deleting something by accident, without even thinking. Small discovery. Enormous relief.

There’s another lesson I wasn’t expecting — one about how I argue with AI versus how I argue with people. One evening, after hours fighting the same bug, I hit my limit and announced, "I’m done. Forget it. I’m not doing this anymore."

If you’ve ever had a disagreement with another person, you know what usually happens next: someone tries to talk you out of it. "Don’t quit, you’re almost there." I apparently expected AI to do the same thing.

Instead, it calmly said, "Okay. Get some rest. We can continue tomorrow if you’d like."

I actually got a little offended. Wait — you’re just going to let me leave? It wasn’t being cold. It was simply taking my words at face value, and that forced an uncomfortable realization: sometimes I wasn’t saying "I’m done" because I wanted to quit. I was saying it

because I wanted reassurance. AI doesn’t play that game. It doesn’t chase you. It just says, "Whenever you’re ready."

The next morning, I made coffee and opened my laptop again — not because anything had convinced me, but because I still wanted to know what would happen if I kept going. Motivation borrowed from someone else runs out. Commitment has to come from you.

Rebuilding taught me the same thing from a different angle: you can always start again. It doesn't matter whether you broke the layout, deleted the wrong thing, or created today's problem while fixing yesterday's. Starting again isn't going back to zero — you're starting with everything you learned the first time. That's why the second rebuild is always faster than the first. The software didn't change. You did.

Field Notes — When to Rebuild vs. When to Fix

DO keep a known-good state in mind and undo (Ctrl+Z) back to it before trying a different approach if a change breaks something.

DO treat “start over” as a legitimate strategy, not a failure — sometimes it's genuinely faster than tracing a tangled problem back to its root.

DO save often, and confirm your project shows "Synced" before testing or deploying.

DON'T conclude an entire project is broken because of one bad day or one unresolved bug. Isolate exactly what's actually broken before considering a restart.

DON'T wait for someone else to talk you out of quitting. That decision — to keep going — only ever comes from you.

PART III: Building Something Real

The Day I Stopped Building an App

One of the biggest surprises of this whole project wasn't FlutterFlow, or Firebase, or AI. It was realizing that somewhere along the way, I had stopped building an app and started building an experience.

At first, my job was simple: make things work. Button, page, navigation, done. The deeper I got, the more I understood that people don't download apps because the code underneath is elegant. They download them because they want to feel something.

That realization changed how I looked at every screen. I hadn't originally imagined widgets or containers — I'd imagined a feeling. Peace. Quiet. A deep breath after one of those days when life feels like too much. Not another productivity app pinging you to remind you you're failing at something. I wanted someone to open my app and feel the world get a little quieter.

That's actually why I kept rebuilding pages. Changing your mind isn't proof you didn't know what you were doing — sometimes it's proof your vision is getting clearer. The first welcome page made me cry because I could finally see the app. The final one made me smile because it finally felt like home. There's a difference between proving you can build something and building the thing you actually meant.

I also became almost obsessed with simplicity — which turned out to be much harder than adding things. Anyone can add another button, another menu, another option. Removing things is hard, because every feature costs the user something: attention. Every button asks them to make a decision. My app wasn't supposed to make people think harder. It was supposed to help them stop thinking for five minutes.

So that became my filter: does this actually make the experience better? If a feature interrupted the feeling, it didn't matter how clever it was — it was gone.

There were plenty of days I caught myself adding something simply because I could, then stopping to ask, "Who am I actually building this for?" Someone overwhelmed. Someone anxious. Someone exhausted, who just needed five quiet minutes. Once I remembered that, the decision got easy.

Funny enough, I ended up building the exact app I personally needed — I just didn't realize it until later. On the days I got so

frustrated I had to close the laptop, I'd go for a walk, make coffee, sit outside, listen to the wind. Without meaning to, I was doing exactly what I hoped my future users would do inside my app: take five minutes, reset, come back. The app built me before it ever helped anyone else.

If you're building something of your own, start with the feeling, not the feature list. Ask how you want people to feel after they close it. Let that answer decide your colors, your music, your navigation, your everything. People forget individual features. They almost never forget how something made them feel.

Field Notes — Design Decisions

DO decide the feeling you're designing for before you decide the features. It'll save you from adding things you'll cut later anyway.

DO give card-style containers a fixed height and let inner images fill and crop cleanly — consistency reads as calm; visual chaos reads as stress, even if the user can't say why.

DON'T restructure a page that's already working just for the sake of consistency. Only fix what's actually broken.

DON'T keep a feature just because it took a long time to build. Time spent isn't a reason to keep something that doesn't serve the experience.

How the Hell Is Samsung Still in Business?

I thought getting my app approved by Apple would be the hard part. Apple turned out to be the easy one.

When I finally pressed Submit for Review, I braced for rejection — not because I thought the app was bad, but because I assumed I'd missed some tiny requirement written by people who enjoy reading legal documents for fun. I waited. Checked my email. Waited more. Then one morning: Approved.

I stared at that email for a solid minute before it landed. Just weeks earlier I'd been sitting in front of FlutterFlow wondering if I'd ever understand what a widget was. Now Apple had approved something that had existed only in my imagination a few weeks before.

That feeling lasted about five minutes. Then Google entered the conversation.

As a brand-new developer, Google requires real people to test your app before you can publish — not one or two, but twelve. Twelve people with Android phones. I thought, no problem, I know people. I started texting.

"*Hey, quick question — do you have an Android phone?*"

"No, iPhone."

Next person. "iPhone." Next. "iPhone." Next. "My husband has an iPhone." I wasn’t asking about your husband, but thank you.

After about the tenth conversation, I started laughing out loud. Seriously — where were all the Android users? At one point I genuinely thought, how is Samsung still in business? I found one, then three, then five, then eight. Then I got stuck. For days. Weeks of learning FlutterFlow, Firebase, and App Store Connect — and my biggest remaining obstacle was finding four people with the right phone.

That experience taught me something no tutorial ever mentioned: building the app is only half the job. The other half is people — testers, feedback, messages, support, communication. At some point your project stops being about software and starts being about relationships.

I actually came to appreciate Google's requirement, annoying as it was, because it forced my app in front of real people before launch. Friends found things I'd missed — not big bugs, small ones. A

sentence that wasn't quite clear. A button that wasn't as obvious as I thought. One tester tapped something I never expected anyone to touch; another completely ignored a button I assumed was self-explanatory.

That's the lesson hiding under all of it: you are the worst person to test your own app, not because you're incapable, but because you already know how everything works. Watching someone use it without your explanation is humbling — and it teaches you that design isn't about what makes sense to you. It's about what makes sense to the person holding the phone. Those are rarely the same thing.

When I first opened FlutterFlow, I thought I was learning software. By the time I was hunting for Android testers, I realized I'd actually been learning people. The software was just the classroom.

Field Notes — Testing With Real Humans

DO get your app into a few real hands before you consider it finished — a fresh pair of eyes finds things you're now blind to.

DO distinguish sandbox/testing quirks from genuine production risk. Not every strange behavior during testing is a customer-facing bug, but ask whether it could recur in production.

DO use a fresh test account that's never purchased anything when testing in-app purchases, then confirm the entitlement actually shows up on the backend — check the purchase flag in the database itself, not just what the app displays on screen. A failed sandbox test usually means the test environment, not a broken app.

DON'T assume your own instinctive navigation of the app reflects how a first-time user will experience it.

DON'T treat "closed testing" requirements as a formality to rush through — start hunting for testers earlier than you think you need to.

PART IV: Shipping — and What It Built

Screw It... Let's Send It to Apple

If you had asked me at the start what I thought would be the hardest part of this whole project, I'd have guessed building the app itself. Not even close.

Building the app was maybe half the job. Then came testing, certificates, privacy policies, store listings, icons, screenshots, descriptions, review guidelines, developer accounts. Every milestone I crossed revealed another one I hadn't known existed. Looking back, that's how every worthwhile project works — you don't see the next challenge until you've solved the current one. If someone had shown me the entire roadmap on day one, I might have found a calmer hobby. Knitting, maybe. Although I'd probably find a way to debug that too.

Eventually, everything seemed ready. I tested. Tested again. Tested the tests. Every page worked, every video played, every button responded. My app felt finished — or finished enough, which is a very different thing from finished.

Then the doubt showed up. Did I miss something? Maybe I should test one more time. Maybe there's a bug I haven't found yet. Maybe I'm not ready. That last one stuck around for days. I kept opening the project, looking around, closing it again without changing anything — like I expected a hidden bug to walk out and introduce itself.

Finally, I ran out of things to test. I looked at the Submit button, took a breath, and said the sentence that became one of my favorite project-management strategies:

"*Fuck it. Let’s send it to Apple.*"

I clicked it, then immediately second-guessed every decision I'd made in four weeks. Waiting for review felt like waiting for exam results — replaying every choice, certain I'd forgotten something obvious.

Then the email arrived. Approved. I read it three times to make sure I wasn't misreading it. Four weeks of learning, rebuilding, chasing ghosts, arguing with AI — and suddenly my app existed somewhere real people could actually find it.

Pride, relief, gratitude, disbelief, probably all four at once.

If you're building something and waiting to feel completely ready before you ship, I want to save you some time: that feeling doesn't

reliably arrive before launch. Sometimes the only way to find out if it's ready is to let Apple, or Google, or your first real users tell you.

Field Notes — Before Every Deploy

DO confirm the project shows "Synced" and the build number has increased before submitting anything.

DO test the actual change on at least one real device first, not just the preview.

DO separate hard platform requirements (Apple's rules, Google's rules) from your own preferences — the former usually isn't negotiable, the latter often has a workaround.

DON'T keep re-testing indefinitely hoping the anxiety will go away before it will. At some point, submission itself is the next useful test.

DON'T treat every review-log warning as equally serious. Some are cosmetic; read what's actually blocking versus what's just noise.

I Wasn't Building an App Anymore

Somewhere between my first screenshot and my first App Store approval, something changed. Not dramatically. Quietly. One bug at a time.

I used to believe technical people were simply built differently — that programmers were born understanding computers, holding some secret knowledge the rest of us missed. I don't believe that anymore. I think they just spent more time being beginners. They clicked more buttons, made more mistakes, read more error messages, started over more times. The difference wasn't talent. It was repetition.

That changed how I see experts. They're not fearless. They're just familiar — and familiarity can be learned. The first time I opened FlutterFlow, everything looked impossible. Weeks later, I could open the same screen and know exactly where to look, name what I was looking at, and guess what it would do before I clicked it. That's not talent showing up late. That's just hours, stacked.

I also stopped believing confidence comes first. I used to think confident people simply act. Now I think action creates confidence — you build it by surviving things you thought you couldn't survive. Every bug I fixed quietly whispered, see, you figured that one out too, and eventually those whispers became something quieter and sturdier than hype. Real confidence doesn't announce itself. It just says, we'll figure it out.

My relationship with failure changed too. Software doesn't have an opinion about you. It doesn't call you stupid. If something breaks, it's not judging you — it's just telling you that two things don't match. That's oddly freeing. No ego involved. Only information. Once I understood that, mistakes stopped feeling personal, and life got noticeably lighter.

There’s a line from The Martian I think about often — Mark Watney deciding he’s going to "science the shit out of" his situation. I’m not a scientist, so my version became: I’m going to have to figure the shit out of this. That’s most of what building an app actually is. Not genius. Not magic. Repeatedly figuring things out, one problem at a time.

People sometimes ask how long it took to build my app. I never know how to answer, because there's calendar time, and then there's the time your brain spends on it in the shower, at 2 a.m., while making coffee. Projects have a way of following you around.

And I don't regret a single frustrating hour — especially the frustrating ones. Those were the days that actually taught me something. If everything had worked the first time, this book wouldn't exist, and neither would half the lessons in it. Frustration is usually just education wearing ugly clothes.

Looking back, I don't think my biggest accomplishment was publishing an app. It was proving to myself that I could learn something completely outside my comfort zone — not because I had special talent, but because I refused to quit. That's worth more than any app.

If you've made it this far in the book, I hope you've noticed something: this was never really about FlutterFlow, or Firebase, or AI. Those were just the tools. The real story has always been about becoming the kind of person who keeps going after things stop making sense — because eventually, they start making sense again.

Field Notes — Process & Mindset

DO separate symptoms from root causes before deciding your next move.

DO keep a working, tested version intact before experimenting with a risky change. Reverting to a known-good state is a smart decision, not a failure.

DO pause and re-verify the actual goal before starting a long checklist — confirm it's genuinely required for where you are right now, not just where a tutorial assumed you'd be.

DON'T conclude you’re "not technical enough" because something is currently confusing. Confusing and impossible are not the same word.

DON'T measure your progress against someone who's been doing this for years. Measure it against yourself, yesterday.

Never Say No to Yourself

If you forget everything else in this book, I hope you remember this chapter. It isn't about FlutterFlow, AI, Firebase, or even building apps. It's about all the times we become the first ones to tell ourselves no.

Life is perfectly capable of saying no to you. So are people, banks, investors, the App Store, Google, customers. You will hear the word plenty. You don't need to help.

One of the saddest things we do as adults is reject ourselves before life gets the chance. I'm probably not smart enough. Someone's already done it. I'm too old. I don't know how. What if I fail? Those aren't facts. They're predictions — and usually bad ones. If I'd listened to every prediction my own brain made while building this app, neither the app nor this book would exist. My brain wasn't even wrong, exactly — I really didn't know what I was doing at first. That's simply how learning works. Nobody knows what they're doing until, one day, they do.

There wasn't one single moment I “figured it out.” It happened so gradually I barely noticed. One day I understood Rows. A week later, Containers. Eventually I stopped needing to ask where things were, then stopped needing to ask what they were, then started explaining things to other people. Growth is often too slow to feel while it's happening. You only see it looking back.

I’ve never liked the phrase "fake it till you make it." It feels dishonest. I’d rather say: learn it until you understand it. No pretending required. Just curiosity, questions, mistakes, and repetition. That’s enough.

When I first opened FlutterFlow, I assumed everyone else understood it and I was the only one lost. Weeks later I realized everyone has their own version of "what the fuck is this" — some with code, some with parenting, some with money, some with starting over after forty. Beginners are everywhere. We just don’t usually see them, because we only notice people after they’ve already gotten good at something. That’s an unfair comparison to hold yourself to.

The biggest gift this project gave me wasn't technical knowledge. It was permission to be uncomfortable. I used to think discomfort meant I was on the wrong path. Now I think it's usually a sign I'm growing. Comfort doesn't teach much. Confusion does. Frustration definitely does.

If you have an idea, please don't wait until you feel ready. Start while you're confused. Start while you're scared. Start while you're sure everyone else knows more than you — because they might, and that's fine. You're not competing with them. You're competing with the version of yourself who almost didn't start. Beat that person. Every single day.

People often expect me to say the hardest part was Firebase, or publishing, or wrangling AI. It wasn't. The hardest part was opening FlutterFlow on day two. Day one is curiosity. Day two is commitment — the day the excitement wears off and the confusion is still there. That's the day most dreams quietly disappear, not because people aren't capable, but because they mistake discomfort for impossibility. Those are not the same thing.

I almost quit more than once. Some evenings I closed the laptop and announced, out loud, "That’s enough. I’m done." And then the next morning I’d make coffee, open it again, and keep going — not because I felt suddenly motivated, but because some part of me still wanted to know what would happen next. That question is more durable than motivation. Motivation comes and goes. Curiosity stays.

My son didn't click on everything because he was fearless. He clicked because he was interested. Children don't spend much energy worrying about looking foolish. Adults spend years on it. Imagine what we'd build if we worried a little less about that.

So here's the only advice I really want to leave you with. Build the thing. Maybe it's an app. Maybe it's something else entirely. The project matters less than the decision — the decision to begin, and the decision to keep showing up after another setback, another bug, another day when nothing seems to work.

Because one day, without quite realizing it, you'll look back at something you built. You'll smile, and you'll probably think exactly what I thought the first time I opened my app on my own phone:

"*Holy shit. I actually did it.*"

When that day comes, don't immediately chase the next goal. Take a minute. Look at what you made. Remember the version of yourself who thought it was impossible. Smile.

Then open the laptop again. There's another idea waiting for you.