Jimmy Miller

"We ran out of columns" - The best, worst codebase

> Oh the merchants2 table? Yeah, we ran out of columns on merchants, so we made merchants2

When I started programming as a kid, I didn't know people were paid to program. Even as I graduated high school, I assumed that the world of "professional development" looked quite different from the code I wrote in my spare time. When I lucked my way into my first software job, I quickly learned just how wrong and how right I had been. My first job was a trial by fire, to this day, that codebase remains the worst and the best codebase I ever had the pleasure of working in. While the codebase will forever remain locked by proprietary walls of that particular company, I hope I can share with you some of its most fun and scary stories.

The database lives forever

In a large legacy system, the database is more than a place to store data, it is the culture maker. The database sets the constraints for how the system as a whole operates. It is the point where all code meets. The database is the watering hole. In our case, that watering hole had quite a bit of pollution.

Did you know that SQL Server has a limit on the number of columns you can have in a table? Me neither. At the time it was 1024, today it appears to be 4096. Needless to say, most people don't need to know this. We did. The reason, Merchants (our table to store customer information) ran out of columns a long time ago. Merchants2 was the solution. A table with (if I remember correctly) some 500+ columns itself.

Merchants (and its best friend Merchants2) were the lifeblood of the system. Everything made its way back to Merchants somehow or other. But it wasn't as if Merchants was the solo (or duo) table. There were plenty of properly normalized tables, all with foreign keys to Merchants. But one will always hold a special place in my heart, SequenceKey.

SequenceKey

SequenceKey
1251238

For ease of understanding, I have recreated the whole of the SequenceKey table above. Yes. You read that correctly, this is the whole table. A table with a single key and a single value. If simplicity is a virtue, then one might declare SequenceKey to be the perfect table. What could be simpler?

But you may be asking yourself, what possible use could you have for a table with one column and row? Generating ids. Now the story I heard at the time was that once upon a time SQL Server didn't support auto-incrementing ids. This was the accepted, correct answer. My search to figure out if this is true was inconclusive. But in practice, it served as much more than that.

SequenceKey was the glue. In every stored procedure that created new entities, you'd first grab a key from SequenceKey, increment it. And then insert that as your ID for N different tables. You now had an implicit join between all these entity tables. If you saw an ID in the system, there was a good chance related tables would have a row with the exact same ID. Honestly kind of clever.

The Calendar

A database may live forever, but our login system was limited by the calendar. I don't mean an actual calendar. I mean a database table called calendar. What did it contain? A manually filled-out calendar. When asking our resident shaman (who went by the name Munch), he informed me that when the calendar runs out we can't login to the system. This happened a few years ago. So they had an intern fill out 5 more years to make sure it didn't happen anytime soon. What system used this calendar? No one knew.

Employees

Every morning at 7:15 the employees table was dropped. All the data completely gone. Then a csv from adp was uploaded into the table. During this time you couldn't login to the system. Sometimes this process failed. But this wasn't the end of the process. The data needed to be replicated to headquarters. So an email was sent to a man, who every day would push a button to copy the data.

The replacement database

You might be thinking to yourself, couldn't someone clean up this database? Make it nicer to work with? Well, the company was way ahead of you. There was a copy of the database. Data in this copy was about 10 minutes out of date. Sync only went one way. But this database was normalized. How normalized? To go from merchants to a phone number was 7 joins.

The Sales Numbers

Every salesperson had a quota they needed to meet every month called a "win". The tables that kept this data (not the financial keepings but a sales-specific way of accounting for this), were incredibly complicated. Every day, a job would figure out which rows had been added and updated and sync them with some system at headquarters. This wasn't really a problem until one salesperson figured out they could ask for those records to be manually changed.

This salesperson had already got their win and landed another big sale that month. They wanted it to be moved to next month. An intern was tasked with doing so. Word got out and over the next three years, requests would grow exponentially. At one point we had 3 interns whose full-time job was writing these SQL statements. Writing an application to do this was deemed too difficult. Before I left though, I made sure to help those interns build their own. No idea if it ever took off though.

The codebase

But what is a database without a codebase. And what a magnificent codebase it was. When I joined everything was in Team Foundation Server. If you aren't familiar, this was a Microsoft-made centralized source control system. The main codebase I worked in was half VB, half C#. It ran on IIS and used session state for everything. What did this mean in practice? If you navigated to a page via Path A or Path B you'd see very different things on that page.

But to describe this codebase as merely half VB, half C# would be to do it a disservice. Every javascript framework that existed at the time was checked into this repository. Typically, with some custom changes the author believed needed to be made. Most notably, knockout, backbone, and marionette. But of course, there was a smattering of jquery and jquery plugins.

But this codebase didn't stand alone. Next to it were a dozen or so soap services and a handful of native Windows applications. Most notable was the shipping manager. Fable has it the entire application was built in a weekend by a solo developer. Let's call him Gilfoyle. Gilfoyle was by all accounts an incredibly fast programmer. I never met him, but I felt I knew him, not just through his code in the repos, but also through all the code remaining on his hard drives.

Gilfoyle's Hard Drives

Munch (yes this was the name he really went by) kept Gilfoyle's hard drive in RAID configuration on his desk years after Gilfoyle had left the company. Why? Because Gilfoyle was known for not checking in code. Not only that, but for building a random one-off windows application for a single user . So it wasn't uncommon to have a user come to us with a bug report for an application that only existed on Gilfoyle's hard drive.

The Shipping Bug

Most of my job was tracking down bugs that teams didn't want to dedicate work to. One particularly nasty bug would pop up once every few months. After we shipped things, the shipping queue would have stuck orders in them, that claimed to both be already shipped and not shipped. I went through a series of workarounds (SQL script, windows application, etc) to try and get us out of the broken state. I was advised not to try and track down the root cause. But I couldn't help myself.

Along the way, I learned how Gilfoyle thought. The shipping app pulled down the entire database and then filtered by date, keeping all orders past the go-live date of the application. The app relied on a SOAP service, not to do any servicey things. No, the service was a pure function. It was the client that did all the side effects. In that client, I discovered a massive class hierarchy. 120 classes each with various methods, inheritance going 10 levels deep. The only problem? **ALL THE METHODS WERE EMPTY.** I do not exaggerate here. Not mostly empty. Empty.

That one stumped me for a while. Eventually, I learned this was in service of building a structure he could then use reflection on. That reflection would let him create a pipe-delimited string (whose structure was completely database-driven, but entirely static) that he would send over a socket. Turns out this was all eventually sent to Kewill, the service that talked to shipping carriers. Why did this bug happen? Kewill reused 9-digit long numbers every month, someone had disabled the cron job that deleted the old orders.

The Beautiful Mess

There are so many more things to tell from this code base. Like the team of Super Senior developers who were rewriting the whole thing without shipping any code for 5 years. Or the red hat consultants building the one database to rule them all. There were so many crazy corners of this code base. So many reasons why there were whole teams dedicated to starting from scratch on just one bit of its functionality.

But I think the most important story to tell is about Justin's improvement of the Merchants Search page. The Merchants Search page was the entry point into the entire application. Every customer service rep would get on the phone with a merchant and type either their id or name to find their information. That would land you on a massive page with all their information. The page was information-dense in the best way, full of any information you could need and any links you could want to visit. But it was dog slow.

Justin was the sole senior developer in my group. He was bright, snarky, and couldn't care less about the business. He told it like it was, didn't pull punches, and could always solve problems by

himself faster than teams around him. One day Justin got tired of hearing about how slow the merchant search page was and went and fixed it. Every box on that screen became its own endpoint. On load, everything above the fold would start fetching, and as one loaded-in, more requests would come in. Took page load time from minutes to sub-second.

Two ways to decouple

Why was Justin able to do this? Because this codebase had no master plan. There was no overarching design the system had to fit into. No expected format for APIs. No documented design system. No architectural review board making sure things were coherent. The app was a complete and utter mess. No one could ever fix it, so no one tried to. What did we do instead? We carved out our own little world of sanity.

This monolithic app, due to sheer necessity, had grown to be a microcosm of nice, small apps around its edges. Each person, when tasked with improving some part of that app, would inevitably give up untangling that web, and find some nice little corner to build new things. And then slowly update links to point to their nice new stuff, orphaning the old.

This may sound like a mess to you. But it was remarkably enjoyable to work in. Gone were the concerns of code duplication. Gone were the concerns of consistency. Gone were the concerns of extensibility. Code was written to serve a use, to touch as little of the area around it as possible, and to be easily replaceable. Our code was decoupled, because coupling it was simply harder.

After

In my career since, I've never had the privilege of working in such a wonderfully ugly codebase. Every ugly codebase I've encountered since has never transcended its need for consistency. Perhaps it was because the codebase had been abandoned by "serious" developers long before. All that remained were ragtag interns and junior developers. Or perhaps it was because there was no layer between those developers and the users, no translations, no requirements gathering, no cards. Just you standing at the desk of the customer service rep, asking them how you could make their life better.

I miss that direct connection. The fast feedback. The lack of making grand plans. The simple problem and code connection. Perhaps it's simply a naive nostalgia. But just as I find myself laying on a couch longing to go back to some of the worst years of my childhood; when faced with yet another "enterprise design pattern", my mind flashes back to that beautiful, horrible codebase.

> **ⓘ Note**
> If you enjoyed this, you can hear more on the interview I did on the Changelog Podcast. I talk more about the crazy code here and about the time the Secret Service busted in my door for "Hacking".