Product for Software Engineers

Sample chapter

This is a sample chapter from the book *Product for Software Engineers* which is available for purchase at productforsoftwareengineers.com.

This chapter is in part two of the book where the spotlight is on how we move quickly to facilitate a strong customer feedback loop of shipping and validating. An important aspect of doing so is to avoid patterns in application development that cause us to backtrack or stumble forward. *Facts* presents two patterns to avoid and recommends alternatives.

Facts

Especially in the early stages of any product or feature, we have to expect very core things to change frequently.

In some sense we have to accept that making changes to code is easier than changing data modeling. Authoring more lines of code will be cheaper than orchestrating a data migration more often than not. I've not seen data migration tools as seamless or less risky to employ than merging a pull request to pepper over the shortcomings of a data model.

With the product evolving quickly and data modeling being harder and harder to improve and keep pace with the code, there is one trick that will help you model your data in a more foolproof and robust way that will endure iterations.

The trick: record facts. Immutable point-in-time records of what happened. You might be thinking "events" but I'm not referring to event sourcing patterns. I am recommending:

- Prefer smaller records (rows, documents) over large records.
- Prefer immutable records as opposed to mutable documents that get tattered with additional fields over time.
- Prefer simplicity in how a table / collection maps to the real world. For example, "emails" over "communication events." There will be time in the future to build materialized views over various collections if it proves necessary.

Designing systems which map to their domain as best as possible as opposed to fanciful abstractions will save you a lot of misjudgment and rewriting. Systems which encapsulate their domains *accurately* are more stable and thus easier to compose into more tumultuous systems nearest to the user.

As areas of a product mature and slow down, there will be time to consider how to reduce or coalesce reads and writes to best suit the access patterns you've found necessary.

It can be tempting, especially with a lot of experience in building the optimized versions of an existing product, to assume you know the right data modeling from the start. I've rarely seen that be the case.

If you'd gone down the path of mutable records, you couldn't as easily grok the activities domain objects experience and even worse you could be missing data that was clobbered by mutation. With facts we can reconstruct usage in much more useful ways to inform optimizations.

Early read/write optimizations tend to not be necessary because the odds that the particular system will benefit from being ready for the scale you're envisioning are slim. If those optimizations do turn out to be necessary, that later stage will be the better time to develop them. There will be more information about real-world usage to inform decisions.

If you guess wrong on early optimizations, you will need to churn and toil through many data migrations.

Keep things simple and stick to the facts. They can take a product's data model pretty far.

State machines

When I was interviewing at Stripe, I recall one of my interviewers proudly touting the fact that Stripe modeled many things using state machines. It was the only time in my 5 hours of interviews I thought, "geez maybe this place isn't for me."

State machines can model a lot of processes, however they aren't flexible in the same way facts are. In an academic sense state machines are cool, but in the field they carry a heavy burden.

Most state machine implementations lean on the idea that an object or resource can only exist in one state at a time. For each of these objects we would keep track of what state it is in, often with a `status` field. For example, an invoice could have a status of `pending`, `paid`, `refunded`, or `cancelled` over its lifetime. In choosing these allowed states we take a snapshot of our understanding of the world.

We get new requirements: invoices have multiple line items, and we need to allow refunding only some of them. Ugh, what do we do? We could keep the status as `paid` but then someone wouldn't notice the partial refunds. If we marked it `refunded` someone would assume the entire invoice was refunded. I guess we need a new state `partially_refunded`? Then we'd have to update a bunch of switch statements...

These requirements keep coming and coming, and because we've locked ourselves into a state machine, we need to keep making judgement calls about how to balance all the complexity and fit it into a single `status` representation.

The other knock against state machines is how the transitions between states are performed. While updating the `status` field may be quick in itself, juggling all the work that needs to be done before and after that state change can be difficult to manage. Especially when we need to account for the state we're transitioning from, not just the one we're moving to.

Most real world systems struggle with transitions. There's a lot of asynchronous work, network calls, and side-effects that need to participate in what otherwise should seem like an atomic operation to the observer of the object's status. The "solution" to this problem is locking, mutexes, and distributed leases over each of these objects as they transition.

In a distributed system where we reach for locks (really they are time-based leases to avoid deadlocks when computers die), we need a background correction job to go around poking things to make sure that they have actually transitioned.

Now we're scanning and scanning for objects, loading them up, reconstituting where they *ought* to be, and transitioning them.

The locks were already slowing simple operations to a crawl, now the correcting background job is falling behind. It needs parallelization to keep up and not lag so far behind, and it always needs to be running. When it does lag behind too much, we may need to advance each object through many states to get it caught up. If there are cyclic state transitions this all can look weird to an observer (e.g. receiving published events or web hooks) as they could receive events out of order or miss instantaneous transitions. Whoops, the correction job failed after a bunch of retries so now we've unlocked a new state: stuck.

This is how state machines spiral out of control in non-toy examples. It seems so simple in the beginning to model things that way, but once we got into building with I/O involved we had to live in eventual consistency and be beholden to our initial wrong choices about that `status` field. We had to reach for locks and other sophisticated tools that have a high mental overhead and take constant babysitting.

Storing facts has none of these problems. We can just keep adding new information as we go, not dancing around this `status` knot in all of our code.

This isn't to say a `status` isn't a handy user-facing construct. But in our implementation we don't want it to be the *driver*. Since we have all the facts recorded, we can still *compute* a handy status on the fly to show if we want.

As for Stripe, over the years the use of state machines has fortunately died out in new features and systems. The accreting, fact-based systems eventually won out as more and more engineers got burned by the high operational toil of state machines.

I'd already learned my lesson about state machines from web development, where managing state and side-effects is practically the whole job. Back-end state machines, as opposed to those short-lived frontend state machines, are even worse because of the persistent nature of the databases and data migrations involved to evolve the system. I guess we can call state machines a full-stack bad idea!

Don't use state machines, stick to the facts.