Ask Technically #3

Integrations, PM requirements for data models, and native apps vs. web apps

Justin
Sep 1, 2022

25

Hello dear paid subscribers, and welcome to the **third issue of Ask Technically**. A while back, I asked you all for burning questions about software, hardware, and everything in between and the response was overwhelming! I've got 50+ questions to work my way through, so we'll tackle 2-3 every issue to make sure everyone gets a quality and thorough answer.

Recall that Ask Technically *of course* does not replace the exclusive paid content (deep dives, company breakdowns) that you signed up for; it's just a new format I'm adding into the mix to make sure you're getting even more out of your subscription. If you missed the last paid post about frontends and backends, you can check it out here.

If you have any questions of your own, just reply to this email or send them to justin@technically.dev.

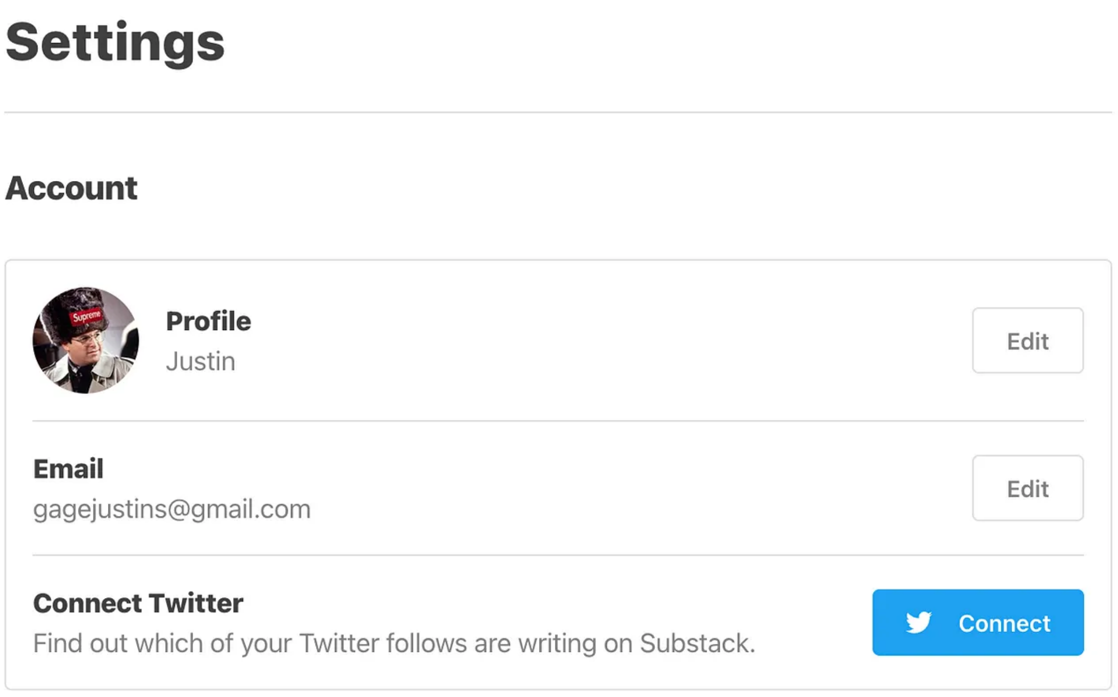
Ram asks…**what is an integration exactly?**

An integration between two tools at its core just means that they're sharing data between one another.

Recall that every piece of software you use has some sort of database that stores information about you and the product; Gmail stores your settings and your emails, Twitter stores your profile information and your tweets, etc. Integrations are how tools share this data between one another in ways that are useful for their users.

For a practical example, let's look at Substack's integration with Twitter. In my settings, I can connect my Twitter account to Substack, and as a consequence

Substack will be able to show me which of the people I'm following have their own Substack publications.



Settings

Account

Profile
Justin

Email
gagejustins@gmail.com

Connect Twitter
Find out which of your Twitter follows are writing on Substack.

So how does this work behind the scenes? It's essentially just **data sharing** between Substack and Twitter.

1. Twitter stores information about who I follow in their databases, and puts that information behind some combination of API endpoints. Let's say it's something like `twitter.com/api/following`.
2. When I click "connect" I'm redirected to authenticate with Twitter — this gives Substack permission to use my data, and Twitter will then "allow them in the castle" to use my information.
3. Substack sends a request to `twitter.com/api/following` to get the list of people I follow.
4. Substack takes that information and cross references it against the information in *their own* database about who writes which publications.

The key here is Substack's ability to get data from Twitter's servers and use it to their liking.

There are many different flavors of integrations. Sometimes, beyond just wanting access to read some data, tools will want to **kick off workflows** in other tools — in other words, *take action*. An example of an integration like this would be one between Hubspot and Salesforce: a popular use case is every time a new marketing lead fills out a form on your website (to, say, read a case study), a new account for them gets created in Salesforce. Under the hood, Hubspot is likely using one of Salesforce's API endpoints for creating a new account.

One more thing to note with respect to integrations is **authentication**. Most of the time, integrations need access to data that would otherwise be private to your account: your Twitter profile information, your email, etc. To get that access, the target (who you're integrating with) will usually ask the user to *authorize* the integration. You've probably seen this if you've used "Sign in with Google" before.

But no matter what type of integration it is, the logistics behind the scenes are the same: each tool is working with available API endpoints from other tools to make data sharing happen.

Samantha asks…**what is the contribution that a developer expects from a PM when building an API? What requirements should the PM provide?**

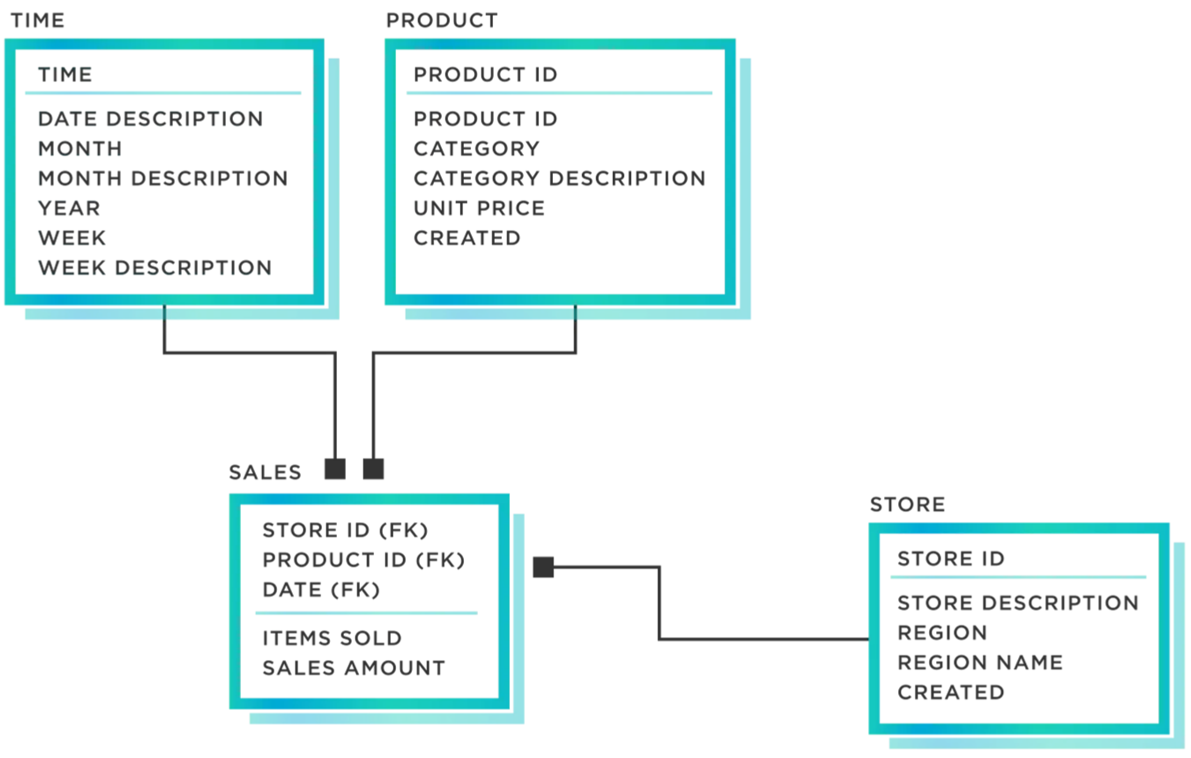
Ah, product management: for those who prefer to do all of the work and get none of the credit. I'm kidding but only sort of.

For those unfamiliar, product managers are responsible for bringing together engineers, designers, and business teams to build software products: creating new features, adjusting existing ones, fixing bugs, etc. Though the role isn't explicitly technical, it requires side by side work with engineering, and as such many lovely Technically readers work in product management.

Sometimes, product managers will work on a new feature that requires some backend work. For a simple example, let's imagine we want to allow our users to toggle whether the app is in light or dark mode. Since we want to persist their chosen setting for eternity – it would be annoying if you need to set it every time you log in – the data on which setting the user has chosen needs to be **stored** in our production database. But there are a million different ways to do that:

- Should it have its own table?
- Should it be stored with existing user data?
- Should it be stored as a boolean (yes/no), or text to allow for more future settings?
- How do we account for other future settings we might want to include?

All of these questions concern **the data model** for this particular feature. And this is a *very* simple example. Some features could require entire sets of tables, rearchitecting how they relate to other existing tables, and more.



A classic example of a data model (source)

So what's the role of the PM here? Sadly that's beyond the pale of this newsletter — we just explain the tech here. Obviously you should communicate with your engineering team and ask what exactly they're expecting from you. *If I had to guess*, it would be useful to your engineering team for you to think through what **impacts** this new feature might have on your product's data universe. Though getting in the weeds on exactly how the data should be stored is probably beyond your responsibility as a PM, having good answers to these questions is not:

- What data will this feature require to work?
- In what ways is this feature's data tied into data in the rest of the product?

Beyond that, as an engineer, I'd prefer to be doing the initial thinking on what the exact data model should look like with *input* from my PM, as opposed to them telling me what to do here. But that's just me!

Daniel asks…**what is a Native App vs. a Web App?**

We've covered this distinction a few times before, so it warrants some more detail.

### Native apps vs. web apps: basics

A **native app** is an application that can only run on a specific platform. An example is a desktop application for your Mac, a mobile app on your Android phone, or an app running on your Roku TV. Each one of these platforms has specific development kits that developers use to develop apps for them: MacOS uses SwiftUI, Android uses, well Android, and Roku TV has their own developer platform too.

A **web app** is an application that runs in the browser, and as such can run on many platforms. HTML, CSS, and JavaScript are the only languages that can run in the browser, so every web app you use is some combination of those on the frontend. When you load a website or app in your browser, whether you do so on your laptop, phone, or TV, you're accessing the same code.

### A portfolio of apps — frontend and backend

Generally, a company will start out with building for one platform and then expand from there. The most common example of this is starting with an app for iOS, e.g. imagine you're Snapchat. Ultimately developers will want to build a portfolio of apps that run across all platforms, e.g. you can access your Gmail through your browser, an iOS app, and Android app, etc.

Recall that almost all applications are composed of a frontend and a backend. The backend is the data and logic for the app — a database and some API endpoints. The frontend is the UI — what the app looks like and how users interact with it.

With all of these different ways of building apps, the one thing that usually stays constant is **the backend**. Whether you're using Gmail on your phone or laptop, you're accessing, reading, and manipulating the same data behind the scenes. So *in general and with some exceptions*, the discussion about native apps vs. web apps is mostly a discussion of frontends.

### The why: benefits of native apps vs. web apps

So why would you build a native app – and need to build and maintain several of them across multiple platforms – vs. just building one web app and being done with that? There's basically 2 answers here:

1. **Distribution** – if you build an iOS app, you can put it in the App Store, get featured by them, appear in search, etc. In theory this means you will grow more / faster. The same holds true for Android, Roku, Microsoft, etc.
2. **Improved user experience** – using a platform's developer kit will give you the most fine grained control over how your app looks, responds, and feels on that platform; on the web, you're just guessing.

Though *generally* the burden of maintaining several native apps across platforms is a pain in the ass, there are some exceptions where you can use a single framework across multiple of these platforms. A good example is Electron, which allows you to "port" your web code in JavaScript to other platforms like MacOS. On the reverse side, React Native Web allows you to port your mobile app code in React Native to the web. And while these tools aren't without their issues, they're moving in a nice direction.

What did you think of this issue of Ask Technically?

- Fantastic
- OK

- 😔 Didn’t like it

If you have any questions of your own, just reply to this email or send them to justin@technically.dev.

Comments

Write a comment...