Technically Speaking

A Cheat-Sheet for Product Managers

🤔Full intro TBD…

This is quick and dirty, ***not comprehensive***. It's focused on the most foundational tech concepts that were important for my team of PMs (most of which did not have a technical background) who managed a set of e-commerce APIs (and the two dozen or so microservices that sat behind those UX-facing APIs).

There is more worth knowing, but haven't had time to try to capture and organize it clearly here. Maybe we can get there over time… 🤷

Introduction

Anything in this document is well covered on the web, so don't hesitate to search for more content on any given topic, to learn more. In addition to all the free content on resources like W3C's site, Mozilla's site, AWS and Azure's developer docs, Stackoverflow, free blog posts, Youtube videos, etc. — there are probably lots of good courses online (both free and paid) where you can dig deeper into specific topics as needed.

Also, since this document started, chatbots have become a fantastic resource for help understanding technical concepts. If it's not making sense, ask it to "explain it like you would to an 8th grader" or something like that; once you've got the basic concept clearly in your mind, then you can drill down to more detail, if useful.

Feedback Is Welcome

If you see topics missing, explanations that are unclear, or you have questions about technical topics coming up at work — feel free to let me know. If I think I can add value there, I'll address them in the doc when I get a chance. Corrections, suggestions, and questions are welcome any of the following ways:

- Comment on this doc:
    - Inline on the section in question
    - Wherever, for general feedback
- Email me:
    - dan@productintuition.com
- My blog:
    - Product Intuition
- DM or mention me on Reddit:

- u/danrxn
- https://www.reddit.com/user/danrxn

If you'd appreciate more content for PMs who want to get more confident working with engineers, have imposter syndrome about lack of technical knowledge/experience, or are curious about a "technical" PM role — please let me know. I can try to do more in this regard, if it would be valuable for folks.

On-the-Job Advice (FWIW)

The following was adapted into a blog post, Why the Smartest People Ask the Simplest Questions.
Lessons I've learned, typically the hard way, that may be useful for others:

1. ***Never pretend*** you understand something you don't. I've found that my smartest colleagues are the quickest to ask questions — even very basic-sounding questions — because they're not insecure about how they're perceived and just want to make sure they have the context needed to do their work (and like getting to learn new things). You don't need to stop every meeting about everything anyone says that you don't understand, but if you're being asked a question about something you don't understand, you should not try to fake an answer. "I don't know, but I'll figure it out and report back" is always an option, and it's radically more productive than just trying to guess or make things up. You can quickly google a term/concept you're wondering about, to figure out whether it's something you can read up on yourself or you'll need help from a team member to understand (e.g. some terms and concepts will be unique to your company, team, codebase, etc.) And it's totally fine to interject with something like, "I am not familiar with X [or don't understand X fully]. Is it easy to explain now — or should I read up on that or connect with someone later to learn about that?"
2. ***Never tune out*** during technical conversations. Even if something is over your head, ***try like hell*** to follow along, and over time your brain will do some level of synthesis based on the things you didn't understand. If you watch young children learning to speak, you may notice they'll sometimes repeat the last syllable of a word they didn't understand. They didn't even catch the whole word, but they're trying to engage and catch what they can. After hearing it enough times, they'll not only catch the whole word and eventually remember it, but they'll also have built up context around when and how others use that word. If you tune out, you're missing the chance to build vocabulary and context that will eventually start to become useful for you.
3. ***Keep a running list*** of things you're not sure about, so you can do your own research when you have time and ask teammates about the things you couldn't understand on your own.
4. ***Experts tend to enjoy sharing what they know.*** If someone is put off by you asking about a term or concept, either you are asking something that you could have gotten an answer to on the first page of the google results, or that person is somewhat of a jerk. Or they're a good teammate having a hard day, which happens to the best of us. But if someone's frustration about being asked turns out to be a pattern, just try to find someone else who seems to know things and is happier to help. As a kind engineer

(now a friend) once told me, when I apologized for asking a lot of questions, “No need to apologize. We ALL need to get up to speed!”

5. ***Some technical concepts are too much*** to pick up from a single conversation, blog post, youtube video, etc. In my experience, there are some technical concepts that can really only be understood by learning to program. But you probably don’t need to understand those, unless your product is a technical one (e.g. a product meant specifically for engineers to use). When I was managing a set of APIs, I spent about 6 months of evenings and weekends learning programming, and that was the only way I was ever going to be maximally effective in that context — but would arguably be overkill if I was working on some full-stack mobile or web app with a GUI.

Keep going! The most valuable learning always feels hard and takes time. Keep your head down, and as the months and years pass, you’ll look back and be amazed how much you’ve managed to learn (and your career results will reflect it).

Foundational Tech Concepts: An Incomplete List

Let me know what else belongs here, if you have unanswered questions on these or think something is missing (you’re probably right about it).

Hypertext Transfer Protocol (HTTP)

HTTP is the protocol that enables the Web — most of what we experience as "the Internet" today. HTTP is an agreed upon structure for sending messages between two computer applications over the Internet, and it is used for web pages and mobile apps to communicate with servers, servers to communicate with each other. HTTP is a specific, standardized way of using Internet Protocol (IP) to send and receive data over a computer network, like the Internet.

While every device and application that is on the internet communicates over Internet Protocol (IP), only certain devices and applications use HTTP (a higher-level protocol, which uses IP). This is similar to saying that all people can communicate via paper and ink, but only certain people can communicate by postal mail (a higher-level "protocol", which uses paper and ink).

Examples of applications which use HTTP:

- Web sites
- Mobile apps
- Complex systems composed of multiple servers

Examples of applications which use a different protocol (not HTTP) on top of IP:

- Email apps and servers (SMTP and IMAP protocols)
- Peer-to-peer file sharing (BitTorrent protocol)

HTTP's Request and Response Model

The entire Web and most native desktop and mobile applications are built around a very simple paradigm — ***request*** and ***response*** (via HTTP). If this isn't already obvious to you, then this is huge. Once you understand this is basically everything running over the air/wire for your product, things your engineers are saying will get much easier to understand!

REQUEST	RESPONSE	Examples
Ask a question...	…get an answer.	● Ask a server what file (e.g. an image) is at a specific address, and get back the image. ● Ask a server what the contents are of a specific page on a website, and get back everything needed to display the page in a browser. ● Ask a server what the price is for a specific product on an online store, and get back the price of that product.
Ask for something to be done...	…find out what happened.	● Ask a server to delete a file (e.g. an image) at a specific location, and get back confirmation that the file has been deleted. ● Ask a server to change the shipping address for a specific user, and get back confirmation that the server has updated the user's shipping address. ● Ask a server to create a new account for a user who is signing up at a website, and get back confirmation that the user's account has been created.

Clients (web browsers and other desktop or mobile applications) can send HTTP requests to servers (i.e client-to-server), and servers can also send HTTP requests to other servers (i.e. server-to-server).

While the request and response model is very simple in principle, it can be used to create very complex systems. Here are some examples of how so:

- A client may get back a lot of data in a single HTTP response
- Part of the response from one HTTP request (e.g. the location of a user's profile avatar image) may be used to make a different request, even to a different server
- A user taking only one action (e.g. clicking a link or button on a webpage) could trigger multiple HTTP requests at once, to a variety of servers/services (or multiple "endpoints" on the same service), and the responses received for each may include data that the webpage will use to make more requests — based on the Javascript code running in the webpage, this sequence(s) of requests and responses will complete only once everything the user wanted to happen is done (e.g. next webpage is displayed, her account has been updated, etc.).
- This kind of complex sequence can happen after every interaction from a user.

HTTP Concepts & Terminology

HTTP requires every request to include several parts, which will give the server the information required to give an appropriate response:

- The address of the server (i.e. host) — Where is the server you're making a request to?
- The "verb" of the request (i.e. HTTP method) — What action do you want to happen?
- The "noun" of the request (i.e. path) — What is the resource on the server, to which you want the "verb" to happen?
- The "details" of the request (i.e. body) — How exactly do you want the "verb" to happen to the "noun"? (This may be blank, when no details are required.)
- Other fiddly bits that aren't critical to understand

Host Address

May be either an IP address (e.g. **`165.225.50.109`**) or a name (e.g. **`www.mywebsite.com`**).

HTTP Methods

HTTP methods are the “verbs” of your request. What do you want to do exactly, for the given resource you’re addressing? There are many, but you can quickly get familiar with the common ones:

- GET
- POST
- PUT
- Etc.

HTTP methods more or less support the typical “CRUD” lifecycle for any given data record, object, or resource. CRUD: Create, Read, Update, Delete. That’s essentially all you can do to/with a given record.

As an aside, CRUD is also a helpful framework when reasoning about what your software needs to be able to do. Whatever kind of data your app deals with, your users may need to do some or all of these operations, sometimes in bulk, so it’s a useful way to think through the basic flows.

HTTP Response Status Codes

Response status codes are well documented at MDN's page, HTTP response status codes.

- 20X
    - 200 OK
    - 201 Created
    - 202 Accepted
    - Etc.

- 40X
- 50X
- Etc.

HTTP Headers

- Some are standard
- Others may be bespoke for a specific endpoint(s)

HTTP Body

Contents can vary widely — any of the following (not an exhaustive list):

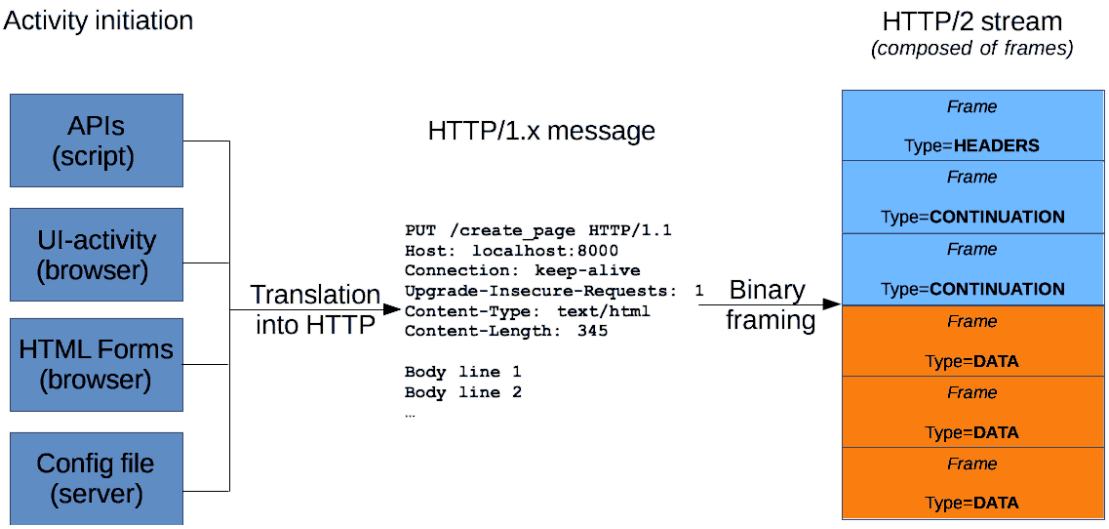
- HTML document
- CSS document
- Structured data (e.g. JSON, XML, etc.)

HTTP Is Just Text

HTTP requests and responses are just little text messages, but they're formatted in a standard way (following the HTTP specification), so that the software on both ends of the connection can understand each other's messages.

Anatomy of HTTP Request and Response (Just Text!)

The following section is quoted from MDN's [HTTP Messages] page:



The HTTP/2 binary framing mechanism has been designed to not require any alteration of the APIs or config files applied: it is broadly transparent to the user.

HTTP requests, and responses, share similar structure and are composed of:

1. *A start-line describing the requests to be implemented, or its status of whether successful or a failure. This start-line is always a single line.*
2. *An optional set of HTTP headers specifying the request, or describing the body included in the message.*
3. *A blank line indicating all meta-information for the request has been sent.*
4. *An optional body containing data associated with the request (like content of an HTML form), or the document associated with a response. The presence of the body and its size is specified by the start-line and HTTP headers.*

The start-line and HTTP headers of the HTTP message are collectively known as the head of the requests, whereas its payload is known as the body.

API Design

Lean on your engineers and architects for designing APIs, similar to how you'd lean on your designers for UI/UX design. But just like with UI/UX, it's super useful for you as a PM to have a sense of the concepts and best practices, so you can ask the right questions and be a good partner in the process — and appreciate when your partners are doing great design work.

REST APIs

A popular pattern for Web-based APIs is REST, and this is the pattern we use at Nike. REST is an abbreviation for ***REpresentational State Transfer***.

- REST is described in Microsoft Azure's Web API design doc for architecture best practices.
- If you're working on a REST API(s) product, it may be worth finding a course on this topic

If you're working with non-REST APIs, do some googling to get up to speed on the paradigm you'll be working with. Try to learn why the alternate pattern was chosen (probably some good, interesting reason) and how it works.

Best Practices

If you're at a big company, hopefully your org has well-documented standards for API designs, so interfaces across all services (regardless of the team who built/owns each) can be consistent in the patterns and conventions they use. Consistency is super important for the developers who need to integrate with multiple APIs to enable the functionality of their own systems or apps, just like consistent UI/UX patterns are important for users of a GUI app.

It's great to learn what best practices are in general and the principles at play. But often consistency will trump "best" practices. If you have an API Standards doc to follow, and it's adhered to across teams, then try to stick to that whenever possible. You doing something "better" than other APIs in your org's ecosystem may be worse for the teams that need to

integrate with your APIs and numerous others that are following the prescribed standards. You don't want your API consumers asking, "Why on Earth did you do it this way, when EVERY other API does it the prescribed way?!" Your answer of "IS BETTER!" may not be well received.

Data Models

⚠️ This is super-duper important! Unfortunately, it's a big topic and needs some good examples to illustrate various concepts. Hopefully I'll get to it eventually… 😬

Interaction Patterns & Call Sequences

⚠️ Still haven't gotten to this one either. Not as important as getting data models right IMO, but still pretty important. Hopefully your engineers will drive this for the most part though.

Structured Data

⚠️ I've not yet gotten around to fleshing this out, so it's more of a list of topics to research…

Formats:

- JSON
- YAML
- XML

About JSON — the most common format for transmitting structured data:

- Elements
- Primitive Data Types (string, number, boolean, object, array)
- Objects
- Arrays
- How UX clients will use JSON responses

Systems

⚠️ I've not yet gotten around to fleshing this out, so it's more of a list of topics to research…

- System Architectures
    - Monolith
    - Service Oriented Architecture (SOA)
    - Microservices.io website explains about the microservices architecture style (often contrasted with monolithic architecture)
    - Cloud Design Patterns, from Microsoft Azure's architecture best practices docs
- AWS Infrastructure (you can find the equivalent if your team uses Azure, Google Cloud, etc. for infrastructure):
    - EC2
    - Lambda
    - SQS

    - Dynamo
    - S3
- Visualizing & Communicating About Systems
    - Breadboards, with optional underlying data — Breadboarding section, from Ryan Singer's *Shape Up* book
    - Block diagrams
    - Sequence diagrams
    - Tables (e.g. "Applications" columns and "Requirements" rows)
    - Example requests/responses JSON payloads — full or snippets
    - Dot notation for referencing specific JSON elements (super useful when communicating with engineers!)
        - E.g. “The ACTIVE or INACTIVE status of the user will be based on days since `user.activity.lastActiveDate` in the response model.”

Tools Worth Knowing (When Applicable)

⚠️ I’ve not yet gotten around to fleshing this out, so it’s more of a list of topics to research…

- Text Editor or IDE
    - Get help from a code editor, when working with structured data like JSON (e.g. syntax highlighting, folding, find/replace)
    - Microsoft’s VS Code (cross-platform) is great and free, but there are a billion of them if you wanna go down a rabbit hole…
- HTTP Client, for hitting APIs directly from your laptop/desktop
    - Curl
    - Postman
    - Paw
    - Etc.
- SQL

Grab-bag

⚠️ These are good topics, but haven’t had the chance to work them into the doc yet…

- Sync/async
- Cache and TTL
- State