CHAPTER 1

First Things First

If you're reading this book, you most likely want to build a mobile app. But must it be *really* an app and not, say, a website? Please read my article at https://probotdev.com/app-or-website-a-guide-to-deciding/ to help you decide. Go ahead. I'll wait.

> Do you still want an app? Because your competitor has it? Because your boss told you so? No worries, we've all been there. I won't judge because, hey, you bought my book!

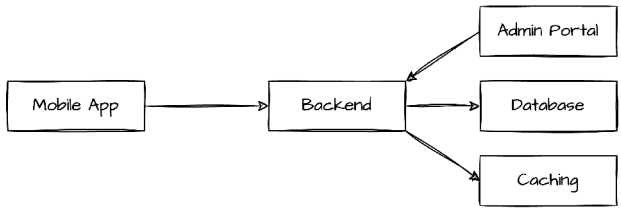
Architecture

So you've decided that you need a **Mobile App**. Let's draw a diagram to illustrate that:

If you're like 99% of app developers, you'll want to show some dynamic data—not hardcoded in your app. You'll also want to collect data about

your users so you can ~~sell~~ monetize it and be the next unicorn. You'll need a **Backend** service to make those happen.

Obviously, you'll need a **Database** to store the data. And sooner or later, you'll need an **Admin Portal** to view and update the data without SQL[1]-ling the Database directly like a savage. And since database access is expensive, and you want to serve things faster, you'll want some **Caching** mechanism. Let's update our diagram:



Looks good. But any serious app allows users to register using their phone number or email, which means your app needs to send **SMS** and **Email**.

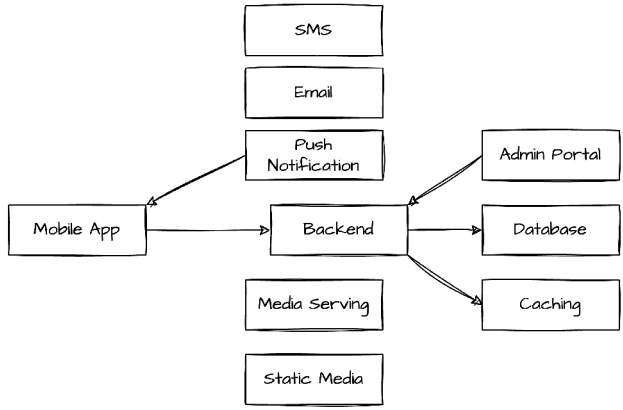
You'll also need **Push Notification** to notify users without spamming their mailbox. Even if you don't need it, Marketing will definitely ask for it.

Many apps allow users to upload pictures and show the uploaded content to other users. Even if your app doesn't allow UGC (User-Generated Content), *somebody* in your company will want to upload multimedia content from the Admin Portal for users to see in the app. You'll need **Media Serving**.

But sometimes, you just want to serve static content but not hardcoded in the app. For example, you want a 4K explainer video (because why not) on the welcome screen. You can't embed the video—you'll get 1-star ratings for the bloat—so you'll need **Static Media**.

[1] SQL (Structured Query Language) is a programming language used to access and manage database.

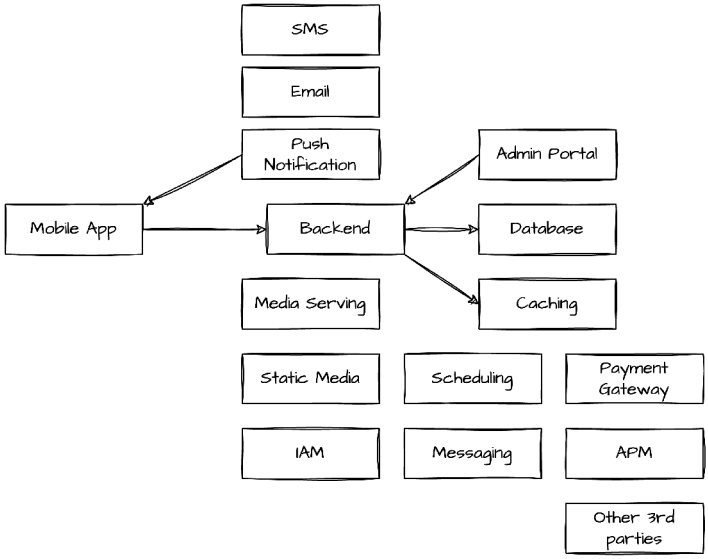
Let's see how our diagram looks now:



Now let's talk about authentication. If your app supports user registration, you need to store passwords securely, have a mechanism to reset them, and allow social logins. You'll need **IAM** (Identity Access Management) for those and more. For the love of mermaids, please don't roll your own solution—especially if the only thing you know about "salt" is that it's a food seasoning (or drugs).

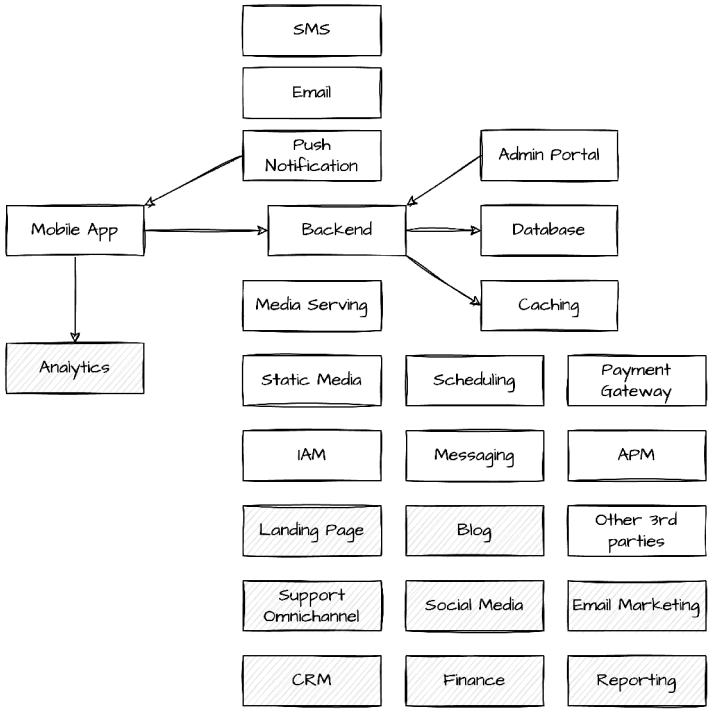
Sooner or later, you'll also need **Scheduling**—for example, to send weekly reports. And when you have multiple systems that need to communicate in a scalable way, you'll need a **Messaging** system.

Most of you build the app to make money, so you'll need a **Payment Gateway** to collect payments. And since you're charging people, you don't want to screw up, so you need **APM** (Application Performance Monitoring) to ensure everything works as intended.



We're done mapping the components needed for an app. But in real life, having an app is not just about *the app*. You will also need the *ecosystem* supporting the app: landing pages to target potential users, support channels for handling complaints, integration with your back office systems (e.g., finance), and many more.

Here's the updated diagram:



That's a lot, and we haven't included indirect support systems, such as project management tools!

You start questioning your career in technology and consider farming instead.

Don't worry. Unless you're a solo entrepreneur, a CEO of a tech startup, or a CTO, some of these might not be your concern. If you are, then this book will even be more useful. This book will discuss most of those components. Not too deep (they each deserve a book) to overwhelm you, but practical enough to get you started. Keep reading!

Platforms

"Platform" can mean many things depending on the context and abstraction level. For our discussion, it's about where to run the server systems.

Unless you work in a highly regulated industry that mandates otherwise, you should always start with the cloud, which is a fancy way of saying, "Someplace I don't know, and I don't own, but I can rent by the hour."

The biggest cloud providers are **Amazon Web Services (AWS)**, **Google Cloud Platform (GCP)** and **Microsoft Azure**—and **Alibaba Cloud**, depending on where you live. They are the "hypermarket" of the cloud with the broadest offering. If you just want a "supermarket," **Digital Ocean** is a good choice.

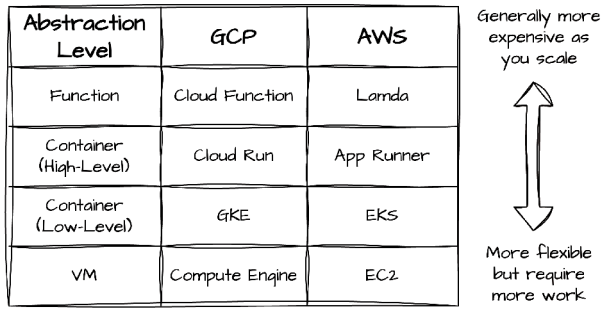
Why would you want to go to a smaller "market"? Usually, they're simpler (because there are not many overlapping choices and they're more opinionated, so you don't have to think about everything yourself) and cheaper (because they don't support all regions in the world, for example).

If your code uses a supported framework and you're willing to adopt a particular workflow, "boutique" clouds such as **Fly.io** (https://fly.io/) and **Render** (https://render.com/) can be a good choice. Using such clouds, you probably don't need a dedicated Ops person since there's very little to "manage."

> **Tips:**
>
> The cloud is always cheaper to get started but can get pretty expensive as you scale. You might want to keep this in mind.
>
> Get the phone number of your cloud provider's local account manager. You'll thank me later.
>
> These days, even regulated apps can be in the cloud as the big players have data centers physically in your region and are certified in various compliance.

After picking a cloud provider, the next step is deciding the abstraction level. You don't need to do this if you use a "boutique" cloud like Fly.io since there's only one level.

A good approach is to choose the highest level that can run your code with no to minimal changes. Here's a table summarizing the levels with some products from both GCP and AWS:



The topmost is the most convenient—you just write code, but it's usually the one with vendor lock-in (it won't run in other clouds). It will scale flawlessly, but there are strict rules that your code needs to follow.

My sweet spot is where I can run my own container image[2] unchanged. I can run the same image in one cloud, in another cloud, or on my laptop, without adding "if in cloud x do this else do that" in the code or creating different "versions" of the image.

Many people also use the cloud to create a VM (Virtual Machine) and install everything by hand. That's like sleeping inside a tent erected in the living room of a mansion. It works, but you're not maximizing the potential. You should only use VM if nothing else works.

[2] We will discuss container images in Development. For now, just think of it as a way to package a server component.

Tips:

You don't have to use the same level of abstraction for all server components—you can mix and match! For example, run the Backend in Cloud Run, IAM in GKE, and Media Service as a Cloud Function.

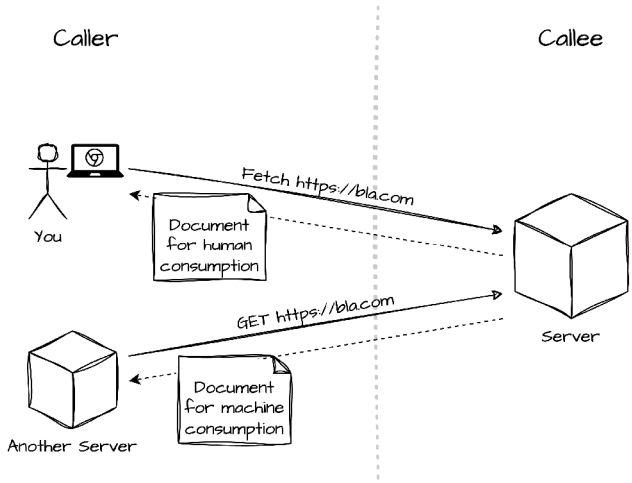
You can also go "multi-cloud," which means using more than one cloud to host your server, either as redundancy (just in case one cloud goes down) or "best fit" (e.g., put media serving in the cloud that specializes in that). Note that cloud skills tend to be specialized—an expert in GCP might not know much about AWS.

API

An API, or **Application Programming Interface**, defines how the components (e.g., Mobile App, Backend, Admin Portal) interact with one another and external systems (e.g., Firebase[3]). In other words, it's how the machines talk among themselves.

API calls are no different than when you, a person, open a website using Chrome or Safari. The difference is how it's called and what gets returned. Here's you calling a server using Chrome while another server, a machine, also calling that same server:

[3] Firebase (https://firebase.google.com/) is a service from Google that provides many building blocks for rapid app development, such as analytics and authentication.



The most widely used API architecture is **JSON over HTTP**[4] **REST**. JSON is a popular data interchange format, while REST (REpresentational State Transfer) is an architectural style. If your API follows REST, it's called a **RESTful API**[5].

Another popular approach is **GraphQL** (https://graphql.org/), usually over HTTP and returns JSON. GraphQL is suitable when the client needs to dictate the data points it needs ("Give me only `first_name` and `last_name`"), unlike RESTful APIs where the data points are fixed ("If you call API X, you'll get A, B, C").

REST sounds restrictive, but in practice API contracts don't change often (and when they do, you can create a new version). Note that you *can* specify data points in RESTful APIs, but the approach is not standardized (e.g., `/api?fields=first_name,last_name`).

---

[4] Usually, this means you can paste the API URL in Google Chrome—as if it's a web link—to see the returned data.

[5] In reality, many APIs out there are not 100% RESTful. It's a mess, but so is the real world :)

gRPC (https://grpc.io/) is also popular. It's a binary format[6] and thus smaller than JSON. Unless your services are very chatty and your app is very popular, I don't think it matters much. Some people mix and match; they use JSON for customer-facing APIs and gRPC for server-to-server communications.

Feel free to experiment with your API structure during development, but once your app is live, it's set in stone: For obvious reasons, you must update all consumers if you change your API. You *can* create a new *version*—existing consumers will use the old one, but putting more thought into designing your APIs is still a good practice.

Even if your Mobile App is the only consumer of your API, you still can't change it willy-nilly. If you do, it might stop working, and people who have it installed on their phones will give you a 1-star rating. Tell them to update the app? Good luck.

Database

If your app has a Backend, a **Database** is almost definitely needed. Most people build apps to display and collect data, and those data are usually stored in a database.

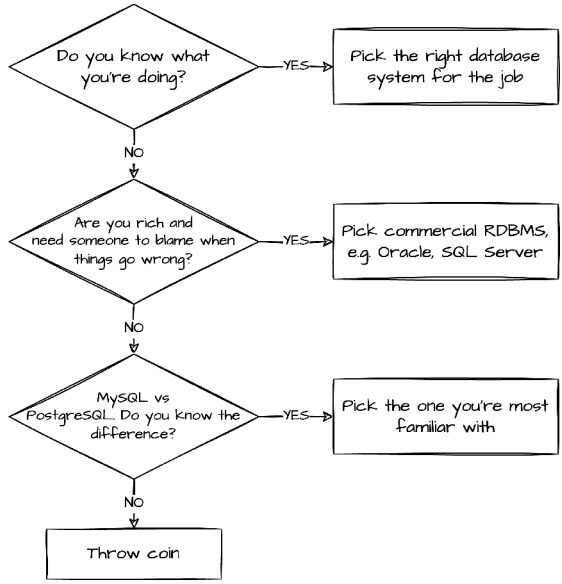
Picking a database system is like choosing a religion—all the popular ones are good, and people tend to go with what they're most familiar with.

Is your team more productive with a specific system? Is it easy to recruit people with the skills required?—Those are the important questions if you go with the popular options. Don't worry about compatibility since the popular ones are widely supported.

[6] As opposed to text format, which you can paste in Windows Notepad to inspect the content.

RDBMS[7] (Relational DataBase Management System) is most suitable for OLTP (OnLine Transaction Processing), a fancy term for "systems that create and update data and can be accessed by many users at once"—which is what 99% of apps in the market is.

MySQL and **PostgreSQL** are the most popular RDBMS in the market. Here's a flowchart that can help you decide quickly:



If open-source scares you, don't be! There is a lot of commercial support for popular open-source databases.

Do note that even though the system itself does not matter (as long as you go with the popular one), the database *structure*—the tables and their relationships—does matter. We will discuss database design in more detail when we talk about Development.

Just like API, you should put more thought into designing your Database.

[7] Also known as "SQL database" to differentiate it with NoSQL solutions.

Once your app is live, it's tough to change. Your code can be thrown away and rewritten multiple times, but the data stays.