Ask Technically #1

All of your questions, forever answered

Justin
May 12, 2022

24

Hello dear paid subscribers, and welcome to the first issue of **Ask Technically**. A couple of weeks ago, 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.

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

Barry asks...**why do developers prefer certain programming languages? What makes them different from one another?**

This is a great question, and a well timed one: any readers tuned in on the web will notice that Rust, a relatively new programming language, has been getting *very* popular over the past few years, and seems to have a cult following. Meanwhile, developers seem to be making fun of JavaScript on Twitter all day. What gives?

The first thing to note is that there are **hundreds** of programming languages, but there's a sort of power law in how they're used; almost 70% of professional developers use JavaScript, while only 1.7% use Haskell. When thinking about the **difference** between all of these seemingly endless languages, you can cut it a few ways:

- **Strongly typed vs. weakly typed**

When programming, you work in functions. You might have a function that takes a piece of data and sends it to a server, or a function that removes dollar signs from a website input. A function is a reusable piece of logic in code that takes an input and gives you an output.

When passing data around these functions, it's important to focus on what **type** of data you're working with. There are tons of different data types like strings (words), integers, arrays (lists), and many more; and your function is likely designed to only work with one of them. And at a lower operating system level, each of these data types takes a different amount of data to store and manage.

In strongly typed languages (Java, C++), you **specify** the type of data that a function takes and outputs. In weakly typed languages (Python, JavaScript), you don't, and simply deal with the consequences if a wrong type gets given to a function. This is basically a tradeoff between rigidity (strongly typed) and ease of use (weakly typed). There's no correct choice here, just different languages for different use cases.

- **Low level mechanics like garbage collection and pointers**

Some languages give you more control of the little details under the hood, like how data is stored and when it gets removed from memory. In languages like C++, you can get extremely specific about where variables store their data and how.

- **Programming philosophies**

There are different philosophies around how programming should work. The main two camps are object oriented programming and functional programming (read more here); the former focuses on objects and classes, while the latter focuses on functions. The details here are beyond our scope, but just think of this kind of like different styles; some developers prefer one over another.

Every one of those hundreds of programming languages chooses some unique combination of the above, as well as lots of other minute differences. And there are even little stylistic differences in syntax (how you write code) that sway developers to choose one over another. It's a wild world out there.

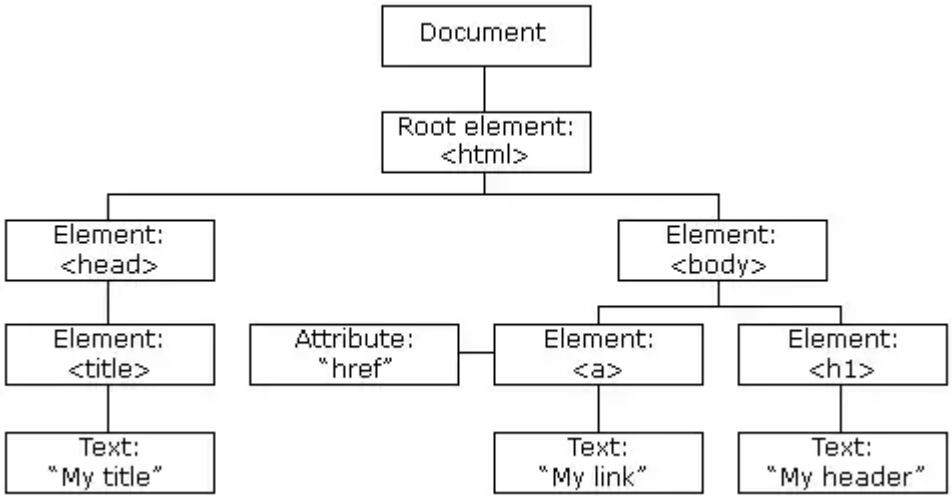
Ben asks...**what is the DOM (document object model) and how does it work?**

(insert sex joke) this is a funny segue from the previous question, because people often ask if HTML is a programming language, and the answer is basically no it's not. But it is the base of every web page you ever load in your browser, and it's predicated on something called the Document Object Model (DOM). It's very simple really. Here's what a tiny web page might look like in HTML:

```
<html>
  <head>
    <title>
      My title
    </title>
  </head>
  <body>
    <h1>
      My header
    </h1>
    <a href="technically.dev">
      My link
    </a>
  </body>
</html>
```

HTML is a **tag system** – you can think of a tag like a specific set of bookends. Each tag tells you what's going to be inside: it might be a title (`<title>`), some header text (`<h1>`), or a link (`<a>`). Note that each tag has an opening tag (`<tag>`) and a closing one (`</tag>`). We've got lots of tags inside other tags: our `<title>` is inside the header tag (`<head>`), and our header (`<h1>`) and link (`<a>`) are inside our `<body>`.

The DOM is how that HTML is represented in data and logic. It's basically what would happen if you mapped the relationship between each one of these "tags" above – which tags are inside of which other tags? Which are next to each other? Which tags have which attributes?



This is how your browser interprets the HTML and renders it to you on your computer. It's also (at least until React) how most developers added interactivity to their web pages. Want your link to turn blue when your user hovers over it with their mouse? You've got to find the `<a>` tag nested within the `<body>` and write some JavaScript that messes with its `color` attribute.

Joey asks…**what exactly is software architecture and how does one decide to change it?**

Software architecture is the sum of design choices that developers make when building their app. In particular, that usually manifests as a few things that we'll run through. But first, to set the context, let's imagine we're building a run of the mill e-commerce app. It's a web storefront that sells Technically swag like stickers, tees, and coasters.

- **What's the data model?**

When building out our database, we need to decide which tables will exist, what data they'll store, and what relationship they'll have to each other. For our store we'll probably want an orders table; but do we also want a products table? How are we going to handle discontinued products? When we update prices, do we change the existing data or add some sort of new pricing table?

- **What are my endpoints?**

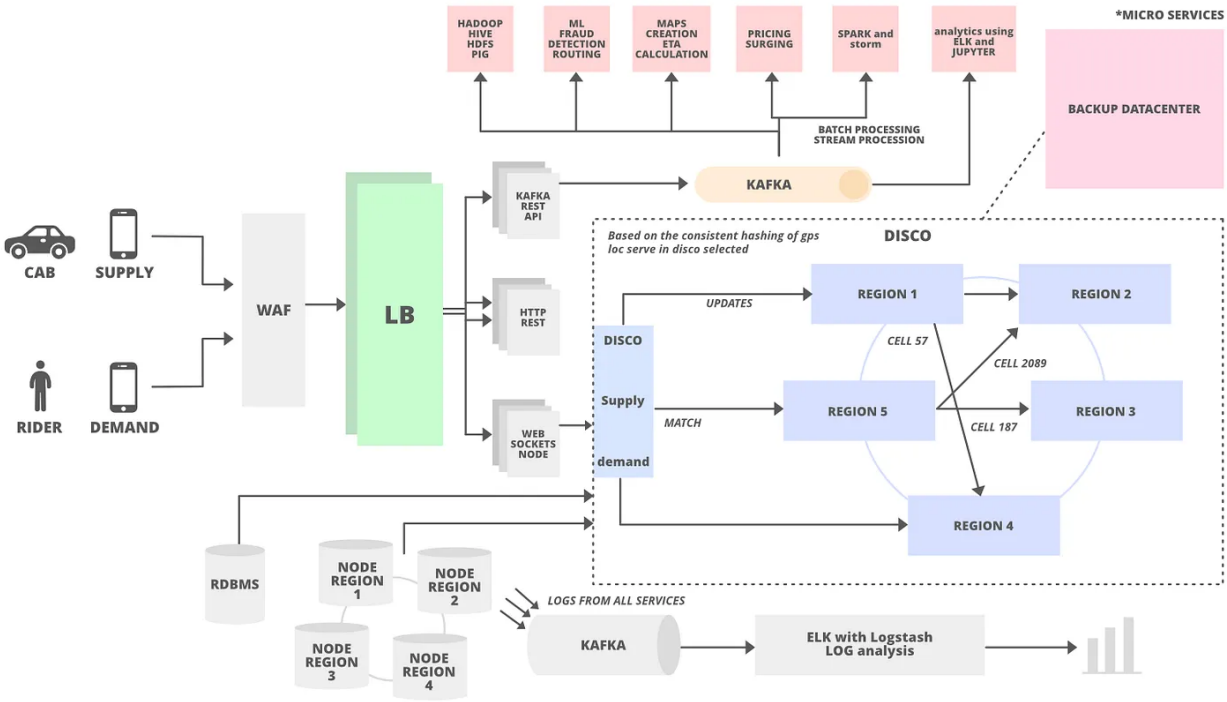
Once we have our database sorted out, we'll need to create some API endpoints so our application can interact with it. Here too, we've got some decisions to make. For example, we need an endpoint that we can use to get data on which products are available (and their details), so we can list them on our homepage. Should that endpoint be the same one we use for *adding* new products too? How do we handle adding filters?

- **What infrastructure am I running on?**

This is a simple one, wherever you choose to run your database, app server, etc. is an architecture choice.

(there are other parts of architecture, but these are illustrative)

For simple applications, these decisions are usually relatively straightforward. But as your app gets more complex, the permutations of how many ways you can design things grow, and as such architecture becomes more both complex and more important. Consider this diagram representing some of Uber's application architecture:



If your app is too slow, too hard to make changes too, or generally not working very well, your engineers may want to make an **architecture change**. That change could be to any of our bullet points above or more. And it's usually a big

deal, since it involves changing pieces of your application that are very important. It may require scheduling downtime (app doesn't work) with your users.

I hope you enjoyed this auspicious inaugural issue of Ask Technically!

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

Comments

Write a comment...