Modern Application Development - 1

September Term 2022

Notes

Uriel K Alistair

Preface

I've disregarded the flow presented by the MAD - 1 course at a few junctures. I recommend reading through the document as it is, not making any jumps, but if you insist that you want stuff in the order presented by the course, then this is your index:

Part 1 roughly corresponds to Week 1 of the course.

Section 2.1 roughly corresponds to Week 2 of the course.

Week 3 is located in Section 4.5 - Designing a View.

The content corresponding to the lab assignments of Weeks 4, 5, and 6 of the course are in Part 3, Sections 3.1 through 3.4.

Week 4 is fairly straightforward, and is thus not located in this document; a superset is covered in the DBMS course.

Week 5 and Week 6 are in Section 3.4.

Week 7 is mostly basic things, and the non-trivial aspects are scattered throughout Part 4.

Week 8 is in section 2.3, and section 3.5 corresponds to the Week 8 screencast.

Week 9 is covered in Sections 4.1 and 4.2.

Week 10 has been omitted.

Week 11 is in section 2.4

Week 12 corresponds to Part 5.

That business aside, I just had fun making these notes. They're a labour of love designed more for my learning in the process of teaching than for reception by a 3rd party, so if you're reading it and it feels terrible, this is why. If you can put up with my informal tone, improper person and tense usage, you might take away something from this doc. If you're reading this, I guess it makes me a little happy that I contributed some non zero utility to the world. Thanks! Have fun.

Introduction

In this course, we will apparently be developing applications.

We can tell intuitively that this can be a completely different thing to learn depending on the platform and the OS we are developing for, considering how wildly different the end results appear to be.

By Platform, I mean Computers, Tablets or Phones; A Phone is designed to maximise the time it can go without needing to be recharged, so, typically, it does not house power-hungry processors. Heavy computations are really undesirable when developing for them. Further, the kind of inputs you can take from the user are completely different: it is touch-based.

In order to develop for a particular OS, we use the corresponding SDK, a Software Development Kit. Migrating an app from one OS to another is by no means a trivial task. There are solutions trying to address this though.

What we will be focusing on is something that transcends these boundaries in most ways: Web applications. Web browsers are already well integrated with all platforms, and provide support to run any web app. It isn't an exaggeration to say that the web is almost like a cross-platform operating system.

To get started on Web development, let us first understand the constituents of a web app, or any application for that matter:

- Storage

The end user makes some requests to the application. Depending on the app, the data needed may be located on the device itself or elsewhere. If you are say, running Skyrim, your PC will suffice. For Clash of Clans though, without the internet, that app is basically useless.

The data the user requests is stored in what is called a server—a large bank of data which we can request information from through the internet. When you hear the word server, think of a long row of 6 feet high boxes of buzzing electronics: hard drives and a few processors, working together. I'm imagining central AC, a lot of liquid cooling inside the boxes and fancy lighting but uhh that may be mostly my imagination.

Anyhow, the server may even perform computations in addition to merely serving data.

At this point, I must mention, there is yet another popular architectural paradigm apart from client-server: peer to peer. BitTorrent and Blockchain function on this basis: data is distributed across several devices rather than one single server we

request. In fact, you could even say that all devices are both clients as well as servers.

- Computation

This load is often distributed between the client and the server. (If it's a client-server application, of course) This is an important job that we need to code.

- Display

This is what the end user sees, and typically this is what comes to your mind first when you think of web-based app development.

These ideas are gathered under the title of "MVC paradigm".

Model – Core data to be stored + databases

View – User-facing side of the app; Interfaces etc.

Control – How to manipulate the data?

Now that we have the broad strokes down, and we know what the pieces of an app are, how exactly does a website work?

End of the day, every website is some code. Where are these pieces of code? Who executes them and when? What language is it in? What does each piece of code do? Does any of this change depending on the application?

Those are mysteries I have always wanted answered.

The end that we see, the *"front end"* is made out of 3 separate pieces: HTML, CSS and JavaScript. We go to a website and request a webpage, it sends us the code containing all 3 of these.

HTML conveys bare bones webpage design, and looks really crappy. CSS, (Cascaded Style Sheets) dresses up the HTML and allows pretty webpage design. But that still does not allow complex tasks to be performed. Neither of these are programming languages in the conventional sense. No if statements, no for loops, no variables.

What we now have is a static visual webpage with the capability to send a few very basic pieces of information back to the server. There is another aspect to webpages, complex functional activities. Consider a flash game. That's something that runs entirely on your system, on the web browser. That used to work on the Adobe Flash Plugin. Flash was a *scripting* language that allowed the execution of *code.*

Adobe Shockwave and Flash were just for web games; most of the web uses JavaScript for any and all scripting needs. Uploading stories, liking posts, adding comments? All JS. Just like Flash, the JS code, and the rest of the frontend, *lives* on ***your computer.*** You can simply view the entire code your computer receives by opening developer settings on your web browser.

In contrast, the backend is made of code inaccessible to the average user. It implements the "business logic", takes care of everything that needs to be done behind the scenes and out of your sights. Backend also uses some scripting, maybe done in a language like Java, completely different from JavaScript, by the way.

If you're playing Clash of clans, the entire base game is a frontend system. It's got pretty graphics, an engine to run that on, visual assets, scripting systems to navigate through the game etc. You can possibly tear open the code that runs on your device, and make changes to it. This won't affect anything on the actual server, only the view that you experience. To change code on the server, you'd have to find some exploit and access *its* data or code.

Nevertheless, the game accesses a backend system that verifies your identity and allows you to perform actions that reflect on all other users who also play the game. Retrieving the actions of other players, communicating that with you, and ensuring all changes are committed to the databases, realtime, is all taken care of by the backend code.

The database is the 3rd big deal. It's typically a non-trivial matter to organise and maintain a database system for a complex application. Someone who can do frontend development, backend development as well as database development, is called a "Full-Stack" Developer.

Basic DB dev stuff is covered in the DBMS course. This course helps us make our first steps towards frontend and backend development. By the end of this course, we will have made a fully functional, fairly professional looking webpage. Well, idk how successful that endeavour will turn out, but err... we'll see.

The deployment of the webpage… well, that comes under the operations part of WebOps.

Development of a webpage has several phases: Things start with an idea, then, the concrete things this project shall achieve in its first release are established. Then all 3 levels of coding are done. Front, Back, DB.

Before this code can be deployed to the public, the system must be checked for robustness.

In a large company, an operations team checks any new code the developer team wishes to release for stability, server loads and such; A testing team will check if the functionality of the new release is all good and catch any bugs; A security team will ensure that there are no threats the webpage is exposing itself to by making this release.

Side note: Webpage releases are version tracked, typically with 3 (or 4) numbers. (Eg: Ver 1.2.12) The last number is reserved for minor patches. The second number is for minor changes. The first number, the version number does not change unless something ***very*** serious has been rewritten or some substantial changes have been made. (Think python 2 going to python 3 level serious.) (Okay, that may be a bit extreme.)

Sometimes devs may choose to update ver 1.1 to 1.5, with an update that feels quite heavy, but not serious enough to warrant a version change.

Anyway, once the testing stages are over, the application undergoes something called packaging. I really don't know much about OS-level virtualization, or its relevance to webpages, but it doesn't matter to us at the moment, considering we are just getting started on our journey.

Once the webpage is ready, it is finally deployed and everyone lives happily ever after.

For a day.

Maybe two.

Let's start walking towards that day, shall we?

Part 1: How does Web Communication work?

Packet switched networks

Initially, when telephone wires entered existence, a wire conveyed the raw voice data. This meant that we needed, at any given point of time, a separate wire connecting each pair of users.

This is, of course, terrible when scaled.

There are two problems here: You can connect all users to one telephone station, but then you need to make all pairs of combinations inside this station and somehow switch them on or off, depending on whos talking to who, and then when you have more than one station, you need thousands of wires for each user connected to this station.

This interstation communication at least, it feels like we can get away with one single wire instead of thousands, can't we? Well, if we wish to use a single wire, we can no longer send raw data. We need some efficient way to overlay various pieces of data and still somehow make sense out of it.

Let's assume that we are using digital bits and not analog signals. At the receiving end, this one wire brings one single line of 1s and 0s. We need a clever way to have data concerning a large number of targets on that one single line. We want to convey the sender and the recipient of each phone call on that one superfast high bandwidth line. (that is, it can convey enough data to accommodate 500 phone calls, but you don't want the entire line to be taken up by Bob talking to Marley.)

The big idea here is that we send "packets" of data concerning different targets intermittently. (Thus, packet "switched" networks) So tiny bits concerning 500 phone calls show up one after another and because this is so fast, nobody notices anything. Each packet is a tiny piece of the total message with the details of whom it is going to, whom it is coming from, what it is about etc. This is "metadata", data about the data.

The use of packets actually also solves the problem of connecting all the pairs of computers within that first telephone center; As long as you have a connection to your network center, you can request information from any machine in the whole wide world, and these packets will jump many servers, find that target machine and establish communication, without having to make a giant mess of wires. If that target machine happens to be connected to the same network center, well, lucky for us, the packets are just forwarded from the network center to that user.

In terms of network topology, this is a "star" of users connected to the hub, which is linked to the rest of the internet.

Apart from details about the sender and the recipient, packets need to contain some more data, when each user requests a variety of different packets. For instance, I may be requesting packets for the podcast I'm listening to, as opposed to packets for the movie I'm torrenting in the background, as opposed to packets for the genshin impact game I'm playing as I am doing both of these.

Now, the details of how exactly to format a packet, exactly what metadata it should contain, what are the checksums and error prevention bits it should contain etc. are called a "protocol". Given a protocol, machines using that protocol can talk to each other and understand the communication.

At the dawn of the internet, there was no standard protocol. All the universities and defence agencies that were building a local network between their machines developed and used their own protocol.

The obvious thing to do now was to develop an Inter-Network protocol—a way for these tiny networks to communicate with each other.

TCP/IP

After about a decade of work, people adopted the "Internet Protocol" on Jan 1st 1983. This defined the header types, packet formatting etc. Once the data was received, it could be converted to the other protocols if desired at the destination for use within that local network.

There is an important thing to note here: our job isn't done at all. You can't blindly expect machine A to shout things to machine B and just pray to god that machine B perfectly comprehends every single bit with no mistakes.

There needs to be some *feedback loop*.

Like an enthusiastic first-bench student, machine B needs to respond with "Okay! Got it!", or "That flew over my head!" or "There is some issue in what I heard; could you please repeat the last bit?"

Further, we need to know how fast we can transmit the data before we find that B starts failing in catching all of what is said, so that data can be transmitted *as fast as possible*.

All this is handled by the Transmission Control Protocol, also adopted on the same day as IP.

There is another less used protocol called UDP, User Datagram Protocol, which doesn't bother ensuring integrity altogether. Instead of providing any guarantees about data integrity, the post processing stage typically checks the data for integrity and chucks any corrupt data and moves on instead of re-requesting

and correcting data. Time-sensitive applications often use UDP because dropping packets is preferable to waiting for packets delayed due to retransmission, which may not be an option in a real-time system. (such as a YouTube Livestream)

By the way, this IP that got adopted, is IPv4, not version 1. Versions 1 to 3 were experiments that never took off. Even today, TCP/IP forms the backbone of the net, in almost entirely the same form as in the 1980s.

Surely, it must be outdated then? Aren't there issues? Yeah, there are, and that's because computers are on a whole other level today when compared to the 80s. We can surely do a lot better with what we have today. However, the old standard is still in place, and the benefits of switching do not presently outweigh the pain of it. A lot of work is being done on making more amazing protocols, though.

IPv4 faces replacement by IPv6. IPv6 was ready for use by 1998, but it was accepted as a universal standard only by 2017. Even after all this while, (I write this in 2022) only 20.9% of websites support IPv6. About 50% of users of google connect to it using IPv6, so the deployment of IPv6 is actually not that bad.

The leading alternative to TCP is QUIC (pronounced quick), developed by Jim Roskind in 2012, as a software engineer at google. Despite its serious improvements, QUIC is used only in less than 8% of net traffic.

I'm legitimately impressed that our modern world runs so smoothly on 80s tech. (Considering how we managed to reach the moon, and still run most older satellites on old tech, the things we did with such basic tech is truly incredible.)

DNS

At the dawn of the internet, it became clear that there was a necessity to identify each machine, each node on the network, with an address. For convenience, a set of 4 numbers each from 0 to 255, separated by dots ("dotted quad notation") was used (rather than a single line 32-bit number), and this address was known as the IP address.

The first number in this set may send your request off to say, America, and then the forwarding server will look for the best match prefix of the IP and forward it to a smaller area where this process can be repeated. It's like physical addresses, you zoom into a level with every piece of the address. This system makes it much easier and faster to send messages to their destination.

Typing a 12 digit number in an address bar to access websites, as if they are mobile phone numbers or something, is kind of a pain. Given the number of websites that were to bloom, using numbers for all of them just didn't make sense.

This gave birth to the web address that abstracted away the IP address of the actual server you are accessing. The host server may even completely change its IP address without you ever noticing because you only use the web address.

This Domain Name System was adopted sometime in 1985. Your request first goes to a server that maps domain names to IP addresses and then once the target is known, communication can begin with them.

Services like GoDaddy, Hover etc. are domain name registrars, offering to reserve your domain names. Although ICANN, the organisation that coordinates and maintains the Domain Name System, is a non-profit, Registry operators, like VeriSign, do run on money. Verisign owns 2 of the internet's 13 root name servers, and owns *the authoritative* registry for all websites that end in *.com* and *.net* among many, many others. A Domain name registrar like Wix, Google Domains or GoDaddy, takes care of all the formal procedures and payments that need to be done to establish a domain. Typically, they'll also host your webpage *for you*, meaning they will have a system online running 24/7 ready to serve your webpage to whomsoever may request it. That has electricity and maintenance charges. Actually, it *is* possible to make webpages for *"free",* though all these stages of the process do need to occur. Typically these same companies: wix and such, will allow you to do things for free, as long as you have an advert for them on your page.

If you've ever seen a "DNS server not reachable" error, this is what it means: the servers which know what IP address each domain name corresponds to, aren't responding to your computer's attempts to establish a connection.

Typically the first place your computer goes to ask "Could you please tell me how to go to x?" is your ISP's name server (or if you're in an organisation, for me, IITM, it will go to IITM's name server). So, your ISP or your employer will know which websites you visit, since they will be the ones you ask "I'm looking for the website x, could you please show me the way?"

Besides, the ISP will be fetching and serving you your data, so you don't really have a way to not tell them whom you are asking for stuff. A way around this, is a VPN, or alternatively, an overkill solution: the tor browser. I'll discuss the onion router later if I find the time.

Of course, no single name server could possibly know every website. Besides, since everybody needs to contact name servers, one single server isn't gonna cut it. Name servers face the largest level of requests, no joke.

Firstly, your computer will maintain a cache of DNS addresses of the sites you frequently visit. For instance, if you request google.com today, it just notes down

the IP and saves it for later. This cache is cleared once in a while because the host IP does change occasionally.

If the website you're asking for isn't in the cache, it will ask your ISP's name server. If the ISP has served this IP recently, it will be in *its* cache and can be served to you directly. That is, if somebody else had asked this name server for google.com, it will have noted down the address and said I'll delete this in 10 days and handle fresh queries later.

If it's not in the cache, it asks the global big deal name servers, which, if they don't know again, will say "Oh, but I know a guy who can help you" and send you to the .com name server. The query you're making is given a query ID to track it across all the bounces it's about to make. If not here, it will work backwards in the URL (Uniform Resource Locator) and ask google's server where google.com is. Always, a bouncing game goes on till you find the IP address you need.

When this system was first introduced, a clever observation was made by people trying to break the system: the query ids were incrementing one by one.

So when I ask for google, I can see what id my query comes back with. Say it was no 1000, I can now flood the Local Name servers with "the answer to queries 1001 to 1100 is this:" and give my malicious website's IP, a website that looks *exactly* like google, but is *mine*. I can steal *all* kinds of personal data if I can convince ***the DNS Server*** that I am google. Odds are, at least one of the "answers" you flood, will be believed by the Name Server.

The worst part about this is this: you can't tell by looking at the URL because now the Name server itself believes that this fake IP is the real home of google.com, and it has *cached* it. Now **everybody else** in your locality is going to be redirected to this malicious website, because they'll get served out of the cache, for say, the next *ten days*.

These days such "DNS Poisoning" attacks aren't that big of a deal, because of security certificates. DNS resolvers can take one look at the certificate and go "That's bogus. You aren't google, get lost. Let me ask the .com server anyway."

We as clients can also view the security certificates of websites, and if that is valid, we are connecting to the right guy.

(This, of course, explains the red screen that reads "This website's security certificate is invalid. Attackers may be trying to steal your information. Proceed with caution.")

In addition, people stopped this steady incrementing of query ids and instead chose the query id to be a randomised 16 bit number. Plus, they randomised the starting port, for even more possibilities.

Right, Ports.

Internet traffic is very diverse. You may be receiving web pages, files, sending emails etc. and all of these do ***not*** use the same protocol.

File transfer protocol (FTP) uses port 21.

Simple mail transfer protocol (SMTP) uses port 25.

HTTP uses port 80, HTTPS, 443.

These port numbers are used by the server side. When you want to access google's https service, you first look up the IP address of google.com, and suffix it with the port that you want: here, 443. The IP address will do the job of locating google's servers geographically, but once your request gets there, it's the port number's job to ask for a particular service. If you ask for port 21, you can access google's FTP service.

On the host side, we use a randomised port number to identify us for that session. Each website we want to access, file we wish to download etc, can use a separate port to keep our processes distinct. The IP address is being combined with the port number to act as a more comprehensive address.

The client has some constraints when it generates a port number randomly. Port numbers are always between 0 and 2^{16}-1.

Ports 0 – 1023 are "System/well-known ports"

Ports 1024 – 49151 are "User/Registered ports": open for registration by developers for a new service they wish to offer. For instance, Microsoft SQL server uses 1433 and Oracle uses 1527.

49152-69535 are the "Dynamic ports" the clients are allowed to randomly choose for their own identity during a conversation with another server.

Going back to what I was saying, the Name servers begun listening on randomised port numbers for the message they were waiting for, rather than single standard so that it's probabilistically infeasible to poison the DNS server. Anyway, the security certificate systems are quite robust, so hey, we are past this being a threat. I'll talk about certification sometime later, when I get to *http**s***.

The Net and The Web

The Internet refers to the physical connection between servers all over the globe. Wires have to join them after all. (That makes you kind of wonder, how we communicate across the ocean, and yes, there are wires on the ocean floor. In fact, "98% of international internet traffic flows through undersea cables".)(Wouldn't it be devastating if a natural disaster cut off a continent?)(Damn, even the idea of no internet for potentially, years, makes me wanna migrate out of that country lol)(Apparently, we have a lot of redundancy and backup + geostationary satellites can relay internet info, just a lot slower, but I can prolly live with 4G, so this isn't really an issue.)

Anyway, the web itself was not born along with inter-network communication. Around 1989, the work of Tim Berners lee in cutting-edge physics done at CERN was adapted into a much more general form called "Hypertext". Hypertext essentially told the system things like "the following bits are characters in bold." "The following bits are the name of another file which you need to load" (A Hyperlink) etc. Eventually, this evolved into HTML, the hypertext markup language, the very foundation of the internet. The Hypertext Transfer Protocol is still the main way a server and a client can talk to each other.

Again, a protocol in this sense, is a way of formatting the data and explicitly stating how it should be interpreted. I'll talk a little later about the details of the HTTP protocol, its headers and such.

Keep in mind, the TCP/IP protocols were adopted in 83, so the internet predates Hypertext. But it *was* Hypertext that completely revolutionised the internet, and made it into the "World Wide Web" - a network of interconnected ***documents*** (or more generally, pages,) all of which were now accessible via http// links.

The original web had protocols only to serve static web pages; the executable interfaces where you could request things and get dynamic feedback were in a *very* shitty state. So far, TCP/IP only established a means for people to send *some* data to each other. That in itself doesn't make a webpage.

Today, you should be able to view the HTML behind any webpage you view by hitting Ctrl Shift I; this hardly a few kilobyte text document has all the information on how to make and display that webpage that you're seeing. You need browsers, in essence, pieces of software that can take the data that these machines serve you, and convert it into some visual interactable page.

Although HTML was the solid ground on which formatted webpages could be swiftly communicated across machines, this didn't mean that there was only one possible browser that could be made; Hell there are dime a dozen *today*. Back then,

on top of just displaying these formatted webpages, you needed some interaction and dynamic refreshment of the page. You needed to make ***requests, and get responses.*** There were a large number of conflicting views on how this communication should be formatted, what data must it convey and so on, and this led to various people making their own browsers that offered a dynamic internet service through their own means. HTTP was the first large unifying protocol that gave birth to a standardised "Web".

Further, the dust completely settled when the so-called "Web 2.0" was born around 2004, with the advent of the concept of AJAX and the use of the Hypertext transfer protocol to transfer not just hypertext, but any form of data. Previously, just to update a small thing in the page, you had to send another request to the server, take the response and reload the whole page.

(Facebook's login page redirects you to an entirely different page, an example of reloading the page with a HTTP response while Instagram's uses AJAX cleverly for a much smoother user experience.)(This is true as of 2023 Dec)

The idea behind AJAX was that a client side script, JavaScript, could request specific pieces of information from the server, and that could be used to update only small sections of the webpage without having to reload the entire page, thus making the web dynamic.

Asynchronously, **JA**vascript ran on the machine, and received changes to be made in the form of **X**ml. These days, XML use for this purpose is basically extinct. It's been replaced by JSON objects. Nobody even uses the term AJAX anymore, but the idea is the core of JS.

With the increased computational power of machines, the rise of JavaScript and the restructuring of HTTP, dynamic pages were in full bloom.

HTTP

As I said, HTTP is a protocol that facilitates server client communication. It can convey clearly what each person wants, and lets the other person give it.

HTTP messages are of 2 types, 2 sides of a coin: request, and response. Both share a common structure:

1. A *start-line*. This start-line is always a single line.
2. An optional set of *HTTP headers*.
3. A blank line indicating the end of meta-information.
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.

HTTP Requests

These are sent by the client to the server, requesting something, or delivering things like the contents of a form you filled.

Let's go through the parts of the request. First, the start line. It will look something like this:

```
GET https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages HTTP/1.1
```

There are 3 parts to this start line:

1. First, the HTTP ***method***, that conveys the action that is being demanded or performed.

`GET` indicates that some content needs to be fetched; `POST` indicates that data is being pushed to the server; A `PUT` request is similar to POST, but is idempotent, meaning it can be performed many times without consequence. Placing the same order 10 times in a hotel will yield 10 dishes, not one. Spamming a submit button 10 times, well, that should ideally *not* push 10 forms from you.

Examples: `POST / HTTP/1.1` , `GET /background.png HTTP/1.0`

`HEAD:` This probes the server for what is to happen, if we send a GET request. For instance, it can find out the size of a file about to be downloaded, without downloading it, just by looking at the size of the http body. The HEAD request triggers a response from the server with all the *headers* that will be in the response for the corresponding GET request so that we can view all the high level details about the response before we actually make a GET request.

Example: `HEAD /test.html?query=alibaba HTTP/1.1`

Note: The things following the "?" are called query parameters. Any number of these can be separated by "&", like https//www.domain.com/page?key1=value1&key2=value2

`OPTIONS:` This asks the server for the list of allowed http methods. It needs to know what the ways it's permitted to interact with the server are, after all. Options may request this for a particular webpage, or for the entire server, by making its target an asterisk, acting like a wildcard that means "all".

Example: `OPTIONS /anypage.html HTTP/1.0` , `OPTIONS * HTTP/1.1`

`CONNECT:` This establishes 2 way communications with the server. It can also be used to do something called TCP tunnelling by involving a proxy HTTP server. I'd rather not get into that at the moment.

---

2. The second part of that line is the request target, typically, a URL. (Uniform Resource Locator)

3. The last part is The *HTTP version*, which defines the structure of the remaining message, and acts as an indicator of the expected version to be used by the response.

This first line is typically our *only* concern.

The headers and body just detail out and carry the data that needs to be carried. Nobody types these requests by hand. We merely specify the method, the url and the data, and some software will package it into a proper http request for us.

HTTP Responses

In the case of a response, the first line is of this form:

1. The http version
2. The Status code
3. The Status message.

Example: HTTP/1.1 200 OK

After this, we have headers, a blank line, and then the body. Again, they are not what we need to worry about on a high level.

Status codes are grouped in five classes:

- Informational responses (100 – 199)
- Successful responses (200 – 299)
- Redirection messages (300 – 399)

- Client error responses (400 – 499)
- Server error responses (500 – 599)

Some popular examples: (You will have to look up these when you code a webpage and after a while you'll memorise the important ones. Know your codes, and return an informative status code.)

• 101 Switching Protocols

Informing the client the protocol is changing to something else.

• 200 OK

The request "succeeded".

• 201 Created

The request succeeded, and a new resource was created as a result.

• 301 Moved Permanently

The URL of the requested resource has been changed permanently. The new URL is given in the response.

• 400 Bad Request

The server cannot or will not process the request due to something that is perceived to be a client error (e.g., malformed request syntax, invalid request message framing, or deceptive request routing).

• 401 Unauthorized

The client must authenticate itself to get the requested response.

• 403 Forbidden

The client does not have access rights to the content; that is, it is unauthorized, so the server is refusing to give the requested resource. Unlike 401 Unauthorized, the client's identity is known to the server.

• 404 Not Found

The server cannot find the requested resource. In the browser, this means the URL is not recognized. In an API, this can also mean that the endpoint is valid but the resource required by it does not exist.

Servers may also send this response instead of 403 Forbidden to hide the existence of a resource from an unauthorized client.

• 418 I'm a teapot

The HTTP 418 I'm a teapot client error response code indicates that the server refuses to brew **coffee** because it is, permanently, a ***tea***pot.

A combined coffee/tea pot that is temporarily out of coffee should instead return 503.

(This error is a reference to Hyper Text Coffee Pot Control Protocol, (HTCPCP) a Protocol defined as an April Fools' joke.☺)

• 500 Internal Server Error

The server has encountered a situation it does not know how to handle.

• 501 Not Implemented

The request method is not supported by the server and cannot be handled. The only methods that servers are required to support (and therefore that must not return this code) are GET and HEAD.

Part 2: The Frontend (Basics)

To review, we have understood the fundamental structure of the internet, and how information is communicated through it. The key idea behind the internet communication systems is that they use a packet switched network. The feedback loop and formatting protocols are given by TCP/IP.

Machines communicate with other machines via messages formatted in some pre agreed protocol, which often is allocated a certain "port". HTTP/HTTPS are the dominant protocols for internet communication which allow the transmission of key data that allows usage of "webpages". Regardless of the protocols used, machines use a complex path of wires involving several machines and servers to actually communicate this data. The geographic location, and the "cyber-address" of the target machine is given by its IP address.

Webpages are communicated in compact documents written in a markup language called HTML. It uses tags conveying semantics only, and it is upto the recipient to convert that text into a visual output. In the case of people with disabilities, the document can be converted into a form that is meaningful to the user, as per the "browser" that they use.

Almost all browsers can render plain html, but html doesn't have the tags to design complex aesthetics. This is where CSS comes in.

The power of CSS lies in in its generality: there are multiple levels of priority for CSS, allowing for easy reuse and setting exceptions. For instance, one can colour all text black with top level CSS, and then specify one html tag alone to use red text, with inline CSS, which trumps the top level CSS. Being able to reuse and *Cascade* Style Sheets, and allowing the priority to work things out is the key advantage of CSS.

HTML isn't a programming language. There are no logical operators or loops you can design. You might need such a thing when designing a webpage. That is what JavaScript is for. JavaScript is advanced frontend material, so I will not handle it in this document. In this document, this is the model we will adopt: The backend server will look at the exact page the user wants to access, and use the user state stored in a database to render a complete HTML+CSS webpage and send it to the user via HTTP. (This is often highly inefficient, because it raises server side load; Offloading work to the user is one of the key uses of JS.)

In Part 2, I will give a basic tutorial walkthrough of HTML and CSS, and describe the role of JS without going into the details. There is a bonus section on some background info on HTML, what it tries to achieve and how people are innovating on it even today.

Because JS doesn't change any of the principles of functioning of the server, we are safe in disregarding it for the moment. There is a lot of nuance in using JS and optimising the server load and user experience, and we will get into in the next document. Part 3 goes into the how-to of making a basic web server, and the design principles around it. Contained within parts 2 and 3 are the bulk of the "Doing things" in this document. This said, these sections are the most useless when simply read, as the way to learn the content in these sections is by actually doing things.

Parts 4 and 5 will serve to further broaden knowledge about the running of the internet, and how a variety of complex issues that arise once we have set up this basic server are tackled. Having obtained a clear overview of what we are going to explore, let's get right into it.

2.1 HTML

Intro

Okay. I'm going to list the bits and pieces of HTML that I've learnt. I mostly followed this video, and it was truly fantastic.

Every HTML document starts with `<!DOCTYPE html>`(It's something like a shebang, a declaration at the top that this document conforms with the W3 Standards. There's a site called W3 checker that will all the problems with a given html file. This is incredibly useful.)

HTML works using of tags. <x> opens the tag x, and </x> closes it. The contents of x possess some property given by x, or the x tag allows some more complex functionality which requires the text inside to function. <x/> opens and closes a tag simultaneously. However, note that not all tags are self-closable. It may be mandatory to have an open and close tag.

In fact, this self-closing business is an XML legacy thing which is now **deprecated.** The use of the trailing slash, is redundant, and the name of the tag is checked against a set of void tags (essentially "An element that existed before HTML5 and which was forbidden to have any content") which do not need a closing tag. The W3 checker keeps crying about this every time I offer a page for it to check. The only valid void tags are area, base, br, col, embed, hr, img, input, keygen, link, meta, param, source, track, wbr.

Non void tags cannot be self closed. It's an error if you try to. HTML is a very forgiving language. It just won't throw an error. It takes so much syntactic abuse and won't utter a word of complaint. This means that a large number of mistakes can go unnoticed, and on one fine day, it will all come crashing down when it is realised that the webpage no longer does what is desired.

Please use the W3 checker. (I don't care if you have an IDE that fixes most things.)

Always.

(Don't forget)

(Promise?)

The Head

Moving on, the entire HTML document is enclosed within <html> tags. Although not necessary for code to compile, we must specify the language of the website explicitly like `<html lang="en">` when we open the tag. HTML is a descriptive

language in this manner. It tells what things are, not how they must look. What is to be done with all this information is decided by the end user who is rendering the HTML

There are 2 main tags that make up the website: the head and the body. The head contains a lot of metadata, files to load etc, and is pretty important, despite not rendering directly in the webpage itself. The body forms the bulk of the page and is all that we see.

The first thing the head is expected to contain is the system of character encoding used in the webpage. `<meta charset="UTF-8">`

Any crucial pieces of metadata are specified using the meta tag.

We can add some more meta tags if we like, for instance,

```
<meta name="author" content="Uriel">
```

```
<meta name="description" content="stuff goes here">
```

Notice how there is no need to close the meta tags.

The next most common thing we put in the head tag, is the title of the webpage. This is the text that is displayed on the tab at the top of the screen.

```
<title> Home </title>
```

Next, we can add an icon to be displayed on that tab using the link tag.

```
<link rel= "icon" href= "pathtocustomimagehere" type= "image/x-icon">
```

The rel conveys what it is we are conveying, relative to the webpage. Seems like bad naming, but okay. Again, note the lack of closing tags. The same syntax can be used to add style sheets to the webpage:

```
<link rel= "stylesheet" href= "pathtocustomcsshere" type= "text/css">
```

Here, that type option is not required, because the default is CSS. Pretty much no other kind of stylesheets are used.

The Body

Basic Text

Moving on to the body of the webpage, let's first try to display text. The `<h1/>` to `<h9/>` tags are used for headings. By default, they decrease in size, with h1 being the largest, but since css can control everything to do with display style, strictly speaking, this need not be the case. These tags specify the way the text is perceived with respect to the rest of the document, and implement some form of a hierarchical structure.

The `<hr>` tag draws a horizontal rule (line) that can separate out text neatly. (No closing tag. I'll use `<x></x>` to denote a tag that needs closing, but if I'm lazy I'll use just `<x>`. Void tags are listed above anyway.)

`<br>` is used to add a line break.

`<p></p>` denotes the paragraph tag. Its contents are just added to the page as a simple paragraph. **Note:** Text tags in html collapse spaces, so even if you say it 100 times, it's like saying it once. Basha will be most disappointed. ` ` denotes one space that cannot be compressed. Nbsp stands for Non Breaking SPace.

The & in general lets you access characters that are in some sense restricted. `&lt` and `&gt` are less than and greater than signs for text, since the real characters get parsed as tag openers and closers.

Now that we can display plain text, let's do some slightly fancier things. To put text in bold, we can use the `<b></b>` tag, but it's preferable to use `<strong></strong>`.

They have the same impact by default, but strong can be changed to mean much more. `<i></i>` puts the text in italics, and again `<em></em>` (emphasis) is the preferable version that does the same job by default.

Strong and emphasis take into account the adjoining stuff and makes contents stand out. This carries well into accessibility use cases, where the content may be transformed for usage by people with disabilities. (That, is their browser equivalent, can parse strong to mean something, whereas bold means nothing to them.)

`<abbr title="expanded text to display">actual text on webpage</abbr>` The abbreviation tag lets you make a short form which when hovered over gives the full form. The short text will be shown with dots beneath the word to indicate the presence of something more.

If you absolutely need to convey the information in the abbr tag, do *not* put it inside that hover box. It's just a fancy thing meant for *some* users. Some people may not be able to see what's in the box, and you must take every step possible to ensure that your page is accessible.

The `code` tag formats the content text to look like it is code. By default, it doesn't have any colour or anything, but that's a CSS job, not an HTML one.

The `mark` tag highlights the text that's within it.

CSS Labelling Tags

`<div id="…" class="…">` to mark a division or major part of the page. This is particularly useful when you want to use a different style for each div with CSS. I'd

say the majority of the divs I've made are purely for the sake of CSS. The attributes "id" and "class" help segregate and manage CSS on a higher level.

The `<section>` and `<article>` tags exist for similar reasons.

The usage of 'div's is discouraged, if the div has any real meaning at all, and the section tag must be used instead. Div acts a divider but carries no semantic information. It provides no benefits to assistive technology. Section tags however, certainly do, even if you do not use any special CSS on them. It's recommended that you partition your webpage using section tags when it makes sense to, for the sake of accessibility. For the outrageous amounts of CSS manipulations I try to pull though, divs obviously make sense, those don't need to have a deeper meaning.

`<span>` is also for segregating for the sake of CSS. It's div but for small things; it's used as a one liner and is useful to change up things in the middle of a line. Span like div, has no semantic information, and must be used only if the situation calls for it.

Fancier text based things

Tables

`<table>` lets you draw a table. Each row is given by a `<tr>` tag. Inside this, each cell is denoted by a `<td>` or a `<th>` tag. `<th>` for table headings, and `<td>` for normal table data cells. `<th>` looks very different from `<td>`, by default, it's bolt and centred.

`colspan` and `rowspan` attributes allows entries to occupy more than one cell.

Tables just look terrible without CSS, so do look into that. (Random trivia: cell borders by default do not merge. To achieve that effect, apply border-collapse at the table level.)

A `caption` tag can be given above the table to describe it. Preferable for accessibility. This caption will take up the length of the table even without any colspan attribute.

Use `<thead>` to enclose the table header row, which is just another `tr`, for the sake of assistive technologies. Similar story for `<tbody>` and `<tfoot>`. The footer is useful to denote, for instance, a sum or average of a column.

It's not uncommon to see headings along the column of a table. Because of this, again, for accessibility, use the `scope` attribute on every th tag. `scope="col"` denotes it's a column heading, and `scope="row"` denotes a row heading.

It's also possible to achieve a similar semantic effect by giving an id to the `th` tags and making **each** td tag have the ids of the two ths it falls under (in the case of row

and column headings) it belongs to in an attribute called `headers`. Sounds like a pain; a job for an IDE, or a custom script.

Lists

`<ol></ol>` is an ordered list. It's a list that is numbered. Each entry in the list is given by `<li></li>` The "type" attribute of the ol tag grants the choice of the way in which the entries are numbered. (1: numbers, A: capital letters, a: small letters, I: roman numerals, i: roman numerals, but small)

An unordered list, which uses shaded circular bullets by default is created using `<ul></ul>` . Here too the list items are given by li.

A dictionary like list, where one thing has a key like entry, and its data follows it, is made by a descriptive list, `<dl></dl>`. Here, the entries are of 2 types are made using `<dt></dt>` (description term) and `<dd></dd>` description details.

Hyperlinks

The anchor tag, `<a href="...">Display Text here</a>` allows hyperlinking.

The href can be an absolute reference, to an entirely new webpage, in which case the link will begin with an https://, or it may be a relative reference, with just a path from the root to a file given, or it may even refer to another place in the current webpage. For instance to link to the section HTML, you'd use href="#HTML"

The link by default opens in the current page. This behaviour can be modified.

```
<a href="..." target=_blank>Click here</a>
```

Opens in a new tab.

```
<a href="..." download>Download Now!</a>
```

Downloads the file given by the link.

```
<a href="mailto:..." download>My email address</a>
```

Tries to open your mailing app and begin a draft (or some such thing)

```
<a href="tel:...">My email address</a>
```

Tries to open your phone app and paste the number into the dial pad.

When a link is clicked (visited), it turns deep purple, and an unclicked link (active) is blue. These colours can be customised using CSS.

a:visited{}, a:active{}, a:hover{} are used to do this.

Click to open

The `<details>` tag specifies additional details that the user can open and close on demand. The <details> tag is often used to create an interactive widget that the user can open and close. By default, the widget is closed. When open, it expands, and displays the content within.

Any sort of content can be put inside the <details> tag.

The `<summary>` tag is used in conjunction with <details> to specify a visible heading for the details. Put the summary tag **inside** the details tag.

Multimedia Content

The `<img>` tag is a void tag that doesn't need a closing tag; It puts an image into the webpage. The src attribute can be used to refer to the relative link of the image. The alt attribute conveys in text what the image contains, so that people who can't see can know what's in the image. Also, the alt text shows if the image cannot be obtained for some reason and its link is broken. Random piece of trivia, Instagram has always put alt messages using an AI for every picture uploaded on it. I noticed only when my internet failed miraculously on a random day.

W3 checker will cry if you do not have an alt tag. Accessibility is a core concept that cannot be ignored. Please tag your images.

The title attribute contains text that is shown when hovered over, but this cannot be seen by all users, so don't put critical information here.

The width and height attributes specify the dimensions of the image. If only one is provided, the aspect ratio is maintained and is used to scale the other dimension. Although this is not necessary it is **highly** recommended that you give these attributes as well.

The loading attribute determines when the image loads. By default, loading ="eager" which means that it will download as early as possible when the webpage loads. The other option is to set it to "lazy", and this means that it will load only when it's about to be shown when you scroll though the page.

Taking Inputs

HTML already has several different types of fully functional albeit shitty looking means to take input. By fully functional, I mean that you just put a tag and when you hit the submit button, all the post request submission magic happens in the background.

```
<form action="url" method="post">
   <label for="fname">First name:</label>
   <input type="text" id="fname" name="fname">
   <br><br>
   <label for="lname">Last name:</label>
   <input type="text" id="lname" name="lname">
   <br><br>
   <input type="submit" value="Submit">
</form>
```

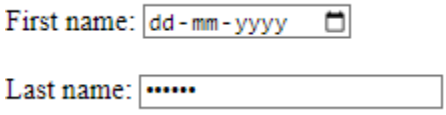
This is a simple text box input inside which a person can enter their name. The label tag sets the label for a particular input box. The input box is identified by its id, a unique name like thing given to tags, which we already saw in the light of CSS.

People like to keep the label tag above the input tag, which is fine, I guess, but that id itself is being defined in the next like, so I kinda find it chronologically weird. I

get that there is no way the page can be made without both pieces of information, and this cannot be looked at like an interpreted language like python, but still.

The input tag defines the id attribute for others to refer to it. The name attribute is given for the sake of the POST request that will be submitted. A variable needs to be filled with this data. We will eventually take the data out of that variable and do what we want with it. The name of that variable is only name.

The type attribute makes the input tag very versatile. It is what converts this form into a large variety of input methods. Here's what happens if I change the type to radio (left) or checkbox (right):



That first one is a date type, it pops up a calendar and lets you choose the date. The second one is a password type, where you cannot see the text that you type inside. Note that this is by no means fit for a password, or rather, any different from the usual text field; the text is just shown as dots. The POST request sends

the password back, **unencrypted** anyway. If you make the method of the form get, the fields entered will go as query parameters in the URL, something even worse.

To securely transmit information, we use https. This doesn't change anything in the html itself, though, it's an encryption and decryption done for every request between client and server, so putting things in the post request can be considered fine, assuming we come back and add https certification to the site.

The type = submit is the post button. Type = reset acts as a reset button which will set all the stuff inside this form tag back into the default state.

All these input tags need not be defined inside a form tag. Only issue is, then the submit and reset buttons become a global level phenomenon. It will reset/ submit every input tag defined out in the open (not inside a form tag) in the entire page.

Type = number takes a number input, but an arrow mark appears and can be used

to increment the entry apart from simply typing the number. Min and max can define the limits, and step defines the increment given by the arrow, and only those values are valid. A bubble will pop up and inform the user if they type in an input that violates min, max or step conditions.

There are a lot of optional parameters that can be used with the input tag.

```
<input type="tel" id="phno" name="phone" placeholder="9999999999" pattern="[0-9]{10}" required autocomplete="on">
```

The placeholder attribute shows something in grey in the form before the user starts typing. This goes away the moment even one character is entered.

The required attribute makes it impossible to hit submit without filling this entry.

The autocomplete option caches the entries made on previous attempts to fill this form, and shows them for filling the next time around.

The pattern attribute only allows those entries that match a particular Regular Expression. (☺)

Overall, the input tag is very powerful. Even so, there aren't types for everything under the sun. HTML does have a few more tags for taking inputs.

For instance, the radio buttons each convey a state of on or off, and the same thing happens with a checkbox. It's not like you can pick only one option. Actually, never mind, input can do that. Just make the variable name the same for all the 3 radio buttons. Because the variable is same, only one button can be picked.

But now, rather than sending an output of on or off for each radio button, we must send some name of the button as the value. To do this, use the value attribute. This sets the value posted when the radio button is on. If it's off, it doesn't output anything.

There isn't a type that lets me list a bunch of values inside the form and makes the user choose from it, like the state or city selection in some web portals.

The select tag is used to make a dropdown list. It plays the **same** role as the input tag:

```
<form action="url" method="post">
   <label for="cars">Choose a car:</label>
   <select name="cars" id="cars">
       <option value="volvo">Volvo</option>
       <option value="saab">Saab</option>
       <option value="opel">Opel</option>
       <option value="audi">Audi</option>
   </select>
   <br><br>
   <input type="submit" value="Submit">
</form>
```

Unlike the input tag which is a void tag, the select tag has all the entries of the dropdown inside it. If we add the attribute multiple to the select tag, multiple entries from the dropdown can be selected by ctrl clicking.

Another cool tag is the textarea tag. It allows a very large box in which paragraph type responses can be given. The size of this box can be defined by the cols and rows attributes.

By the way, putting these input/select tags all inside a fieldset tag puts a border around the specified boxes, and neatly partitions the page, esp for assistive technology.

Some more tags

The `figure` tag can be used to enclose an image. It doesn't add anything visually, and the figure tag need not even be having an image inside it. The `figcaption` tag can be used inside this to provide a caption for the contents. This tag must necessarily be the first or last thing inside the figure tag.

The `nav` tag is used to enclose navigation information, typically, a list of hyperlinks acting as a table of contents. If there is more than one nav tag in the page, it is highly recommended that they are labelled with the aria-label option for assistive technology.

Ideally, you must use only one `h1` tag, and not go beyond `h4`. Random advice, but really, could be broken, but if you're there, I think that it may help to stand back and ask, "What have I done?"

The `header` `main` and `footer` tags allow the semantic labelling of sections of the body.

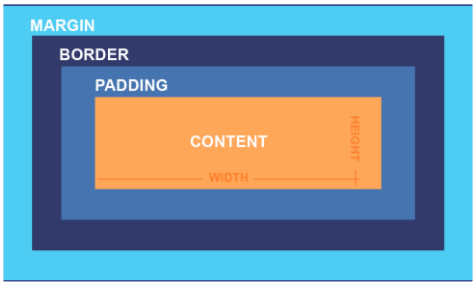
The `aside` tag is used to classify text as something aside from the main content. It's typically housed in the sidebar.

When any date or time is specified in the webpage, it's good practice to enclose it with the `time` tag. The datetime attribute is used to convey the same meaning in a more code-y format, for instance, `<time datetime="PT3H">3 hours</time>`

It does **not** change the experience of the regular user, but again, assistive technology, something you must never forget about, greatly benefits from this.

2.2 CSS

- Heading tags can be customised.
- The entire webpage can be given a background colour by using the background colour attribute on the body tag. Try making it dark mode.
- CSS uses boxes to keep various elements from clashing with each other. Buffer space, if you will. The centre of the box is the actual content. It's of whatever dimension you choose. Surrounding this is the padding, a transparent



material that will take up the colour of the background assigned to the content. Beyond the padding is the border. Outside of this, is the margin, again, transparent and merging with the outside world. Each of these can be customised using CSS. These boxes can be clearly seen by opening dev tools on the browser and inspecting each element in the code, or by hovering over things on the page.

- A Stylesheet can be imported into another CSS file using `@import url("index.css");`
- To allow more customisability, a more specific level of CSS overrides a broader level of CSS. For instance, I may format all paragraphs to have blue text with CSS. If I make an **inline CSS** statement asking a particular para to have red text, this will override the high level CSS. For the purpose of broadly categorising tags and handling CSS, HTML has a lot of tags that seem otherwise redundant.

Any html tag can have an id and a class attribute.

1. A more specific selector takes precedence over a less specific one. (For instance, the id of an html tag has higher priority than the class.)
2. Rules that appear later in the code override earlier rules if both have the same specificity. (For instance, if a tag is given two classes A and B and their CSS are in conflict, the conflicting parameters are taken from the class whose CSS is declared later.)
3. Inline css always overrides imported stylesheets or style tag css.
4. A css rule with !important always takes precedence.

The full set of rules is very long, but we can work with these. If interested, here is a more comprehensive article. (Follow the links in the page for more details)

2.3 JavaScript

Unlike the preceding two sections, 2.1 and 2.2, this section on JavaScript is not meant to be a basic how-to guide about the language. Instead, I'm going to go back to the Part 1 style, and tell a short story about JavaScript.

There are a couple of reasons I want to do this. Firstly, JavaScript is a more advanced frontend tool. So far, we have understood the fundamental structure of the internet, and how a webpage can be rendered with the use of html and css.

JavaScript was created in 1995, for the Netscape Navigator, a very popular browser at the time, which evolved to present day Mozilla Firefox.

Before this point, when people wanted to make interactive content, they would make a sandboxed enclosure for this thing called a Java Applet. Everything from the shape of the cursor would be under the control of the Java code behind this applet. This was a rather restricted use case and could by no means do what JS does today.

Anyhow, in this era, there was a need for some scripting language that could act as a glue between HTML and maybe more than one applet. One of the attempts to bridge this gap between HTML and Java, was JavaScript.

Now, this led to an obvious trademark infringement case, with JS trying to ride off Java's popularity, when neither it nor its founders had absolutely nothing to do with Java in any way shape or form. JS was initially called Mocha, running with the coffee theme, then renamed to LiveScript, but they were able to finally get away with JavaScript, leading to the bemusement of people till today when they find out that Java is not at all related to JavaScript.

Anyhow, JavaScript took the world by storm only around 2005, and it's not that the language went through a drastic change in that year, or that browsers changed drastically, but the ideas around how JS could be used were revolutionised when it was used to create Google Maps.

This large map could dynamically load and you can make these smooth movements, zoom in and out and all that and the page gets modified with just the data that's on the screen. The Key idea that took off, changing the position of JS, was AJAX.

JS has come a long way since then, since the evolving demands caused the language to grow into a different beast. JS was no longer tied to something like Netscape, and all browsers started to support a JS parser (this was still back in 96,97). The European Computer Manufacturers Association ECMA, teamed up with Netscape to set the standards about JS. This is the ECMAScript standard, which is applicable for JS and few very similar languages, and JS conforms with ES4, ES5 etc, like versions or standards to which JS is supported by a browser.

In principle, JavaScript and ECMAScript mean the same thing. Sometimes the following distinction is made:

• The term JavaScript refers to the language and its implementations.

• The term ECMAScript refers to the language standard and language versions.

Therefore, ECMAScript 6 is a version of the language (its 6th edition).

Beyond ES6, which is also called ES2015, ES standards are named with the year of release. Each year in June, a new standard is ratified.

Significant changes occurred in 2015, with ES6, and when I discuss JS in a separate document, we will work with modern features released just a year or two ago which incorporate all of that. Browsers either support it right away, or legacy browsers are updated with parsers that can convert this newer JS code into their old code. Typically, you would look up the level of support offered by browsers for a particular JS feature more than asking if something supports ES2020. (eg: here)

Anyhow, JS takes a lot of abuse so this older browsers business isn't as big of a deal. This is since, from its inception, JS was made to be HTML's romantic partner in BDSM. These days it's ***highly*** encouraged to use the "strict" mode of JS, so that JS doesn't silently crash until the day you pull out every last hair on your head.

At some point people fell so much in love with JS, that they decided they wanted to code in JS for the ***backend*** as well. So they tore out the JS parser from the browser, and put it in its own sandbox, and called it NodeJS. There are a remarkable number of advantages to doing this, because JS is fundamentally tailored to modern webpages.

We will not go into the details of NodeJS and its miraculous single threaded server model here, so I'd like to finish this section with an overview of the kind of things JS has been used for in terms of a high level AJAX viewpoint.

The first use of JS I'd like to point out is client side validation. Let's say you have a simple registration form. When the end user types in an email address that's not valid, (obviously of the incorrect format) or the confirm password entry doesn't

match the first password, or does a multitude of other silly things, you'd like to add a small piece of red text below that corrects the user, rather than waiting for the entire form to get submitted, at which point the server sends back a webpage with a bunch of error divs added.

You cannot expect that every tiny change the user makes gets relayed to the server, and then the server, which can only respond with a HTML doc, makes you reload the full page. That's a typical JS use case.

A login page for instance, can either redirect you an auth_error page, or use JavaScript to prompt an updated div that shows you the error. (Again, Facebook vs Insta login pages. You can feel the difference, can't you?)

You don't have to reload the page, there's less network traffic, and it's a better user experience overall. In comparison to the server's computing, using client side computing makes the results appear faster, and with all these resources available, the bottleneck is almost always the time taken for the message to travel through the internet and back.

In any case, you *must do* server side validation, because people can always bypass the JavaScript and send a post request manually and potentially break something by not meeting the assumptions you make about the data. JS is just for the user experience.

Apart from this, JavaScript allows you to do incredibly cool effects on the elements on your webpage, by dynamically updating the CSS of the divs in the page. Along with CSS animations, this allows you to make almost anything.

JS also finds use in making captchas, and verifying that you're a human. Whenever a website is made, people try to stir up some trouble. People have in the past, written scripts to automatically make tatkal bookings before everyone else, or to book a vaccine in the Cowin app before anyone else could. This kind of automated server pinging is a serious issue, which is tackled both by limiting the number of requests a client can make in unit time, and by putting a captcha barrier.

Sometimes, in your experience, you may have noticed that the captcha box just becomes a tick the moment you click it, and it just lets you through without a gruelling traffic light detection task. That's because there was always a JavaScript code running in the browser tracking your every keypress and cursor movement, and that code decided that your behaviour was human enough. If it had not been convinced, it would ask you for the image detection stuff. Kinda creepy tbh.

JS is powerful, since it's code that runs on your computer, which can access almost the entirety of the power of your processor, and in some cases, your GPU as well. This means that there have been a lot of attempts to misuse the power of JS. For

instance, some websites have tried to mine bitcoin and the like with *your* computational resources. That way, with a swarm of potentially millions of computers running crypto mining algorithms, the owner of the website can make quite a hefty sum. If you try to pull such a thing now, you'll get blacklisted and get into a lot of serious trouble.

Another example of an illegal thing to do with JS is to make a denial of service attack. Just send a request to apple.com from the client's machine via the JavaScript code on your popular webpage, once enough people are opening the website with this piece of JS code, apple's servers simply cannot handle that number of requests, and it will just throw its hands up and sit the storm out. To avoid getting caught, don't run this JS code on *your* website, hack into the source of some popular piece of JS code that a ton of websites use as part of their JS. Retain the functionality and make it send the DoS requests only under some condition. Make sleeper cells, basically. (Do **not** try this at home)

On the opposite end, instead of attacking someone else, attack the client: just use a ridiculous amount of computer resources and hang the clients system, sabotaging it and taking over control of what runs in the system; not that big of a deal now. Typically, your browser will just decide at some point that the website is consuming too much of the system resources and will ask you if you wish to kill it.

2.4 Beyond HTML

Firstly, before we see what will become of HTML, and what kind of changes it needs, we need to look back at its past.

As you might expect, at the dawn of markup languages, (this was in the 60s, by the way) there was no standardisation, and everyone and their grandma made their own language for use by a slightly different target audience.

The first standardised markup language, SGML, took human society out of that godforsaken mess. SGML, Standard **Generalized** Markup Language, unlike, say, HTML, is not something you can write a document in; rather, it's the godfather of all markup languages. SGML doesn't specify what tags are valid, it lets you make your own. The DTD, Document Type Declaration, specifies how exactly a file should be interpreted, and what each tag means.

One of the most important postulates of SGML is that it is declarative. This is as opposed to an imperative way of writing markup. An example of a declarative command is "Make a poster"; The same thing specified imperatively would be "Take an A4 sheet, draw a margin 10 mm from the edges, put down text with height of capital letters set to 5mm, small letters to 7/10 th that of the capitals, and italics text tilted at 25° to the vertical…"

Declarative commands imply that you work at a higher level of abstraction. It only specifies what needs to be done, not ***how*** it is done.

The issue with SGML, was that it was a bit too strict, (Honestly, I understand the sentiment of making something as rigorous as a programming language, and it bewilders me that something as forgiving as html exists, but ok) so HTML began its inception as an application of SGML, which wouldn't actually be valid SGML.

HTML's first version was not compliant to SGML. However, with increasing complexity, it became harder and harder to define a set of rules that would be forgiving. At one point, people just gave up trying to make something that takes so much abuse. HTML 2.0 attempted to become SGML compliant, but because the web already started with HTML 1, they couldn't enforce that compliance, because then, the web would just almost completely break.

With this backwards compatibility issue, HTML trudged towards compliance and reached very close to ideal SGMLality, at HTML 4.0. With HTML 5 the decision was made to completely break away from SGML, and they went ahead and designed their own parser that knew when and where to be strict, or forgiving etc.

Another offshoot from SGML is XML, the eXtensible Markup Language, which also still sees use today (like in RSS feeds, or for parameter storage files); like

HTML, it was also born from a need for a less rigorous markup language. If you recall, AJA**X** began with xml, before JSON took its place. Both JSON and XML satisfy similar roles, but there isn't quite a clear winner between the 2; They both have their place.

Side Note: Scalable Vector Graphics, SVG files, are also specified in XML. Every word document, .docx files, are actually stored in XML.

Now, the interesting thing is HTML5 is in some sense, the last version of HTML. Again, at some point there was a lot of controversy about the development of HTML5.

Two groups were working on it: the W3C, the world wide web consortium, and the WHATWG, the Web Hypertext Application Technology Working Group. WHATWG (I've resorted to pronouncing it double-U Hat double-U G) consisted of people from Mozilla, Google Chrome, Apple Safari, Microsoft; all the people making big browsers.

The browser vendors got together and said "Look, we need to be able to add certain kinds of features, and the way you guys are going about it is not fast enough or convenient enough for our liking; So we're just going to break away, and decide for ourselves how things are going to be done."

As of now WHATWG is the one that maintains the HTML Living Standard. It can be updated at any point, and whenever you say that something is HTML compliant, rather than saying it is HTML5 or HTML5.1, some version number, you just say that it is, as of this date, compliant with the Living Standard.

Anyway, that aside, HTML, unlike XML doesn't allow arbitrary tags. XML permits the use of any namespace whatsoever, while HTML is significantly more well defined.

There are a bunch of problems with letting people define custom tags; for one, there is no longer any semantic information any machine can infer from the tag by parsing the HTML. Secondly, the browser needs to know how to display the HTML. HTML never specified things like font size, or line thickness; It only said this is a heading of level 1, this is emphasised, etc. HTML only gives semantic information about elements, while how things are actually visually rendered is up to the browser which is reading the HTML. Yes, that means that what you see may differ based on the browser.

Despite this, we do wish to go "Beyond HTML" and make our own tags, and the way we solve the problem of displaying arbitrary tags, is with JavaScript. JS has a lot of very cool and very powerful APIs; They push boundaries and make JS functionality limitless.

Browser APIs are built into your web browser, and are able to expose data from the surrounding computer environment, or do useful complex things. For example, The DOM (Document Object Model) API allows you to manipulate HTML and CSS, creating, removing and changing HTML, dynamically applying new styles to your page, etc.

The Geolocation API retrieves geographical information. This is how Google Maps is able to find your location and plot it on a map.

Other APIs are third party services and won't be directly available on your browser. The Twitter API allows you to do things like displaying your latest tweets on your website. The Google Maps API and OpenStreetMap API allows you to embed custom maps into your website, and other such functionality.

For this purpose, we use 2: The Custom Elements API, and the Shadow DOM API. Along with the template and slot tags introduced in HTML5, pretty much anything can be done.

Note: This segment was meant to house a tutorial on how to make custom elements. Due to having strong JavaScript prerequisites, and due to it being very off topic, it has not been included here.

Part 3: Hosting a Simple Server

3.1 Jinja

We have several modules that closely tie python to HTML so that we can write much more sophisticated and long HTML without actually doing any of the grunt work.

Jinja2 is a library that provides convenient ways of templating reusable bits of HTML, and running logical statements or loops inside these text templates. Jinja2 isn't meant exclusively for HTML, it's a general string formatting engine. It sees only strings, not a language.

Typically, using f strings, a variable can be called using {var_name_here}.

Here, double braces {{ }} play the same role.

String formatting the default python way can quickly become inadequate. Jinja is the tool meant for complex string formatting, where logic is often inside the strings.

For instance, you can make for loops inside a string. {% for i in range(1,n) %}{% endfor%} is the syntax for it.

Note: Whatever is in between the start and the end of the for loop is taken as a string. If double curly braces denote a variable or any other jinja technique is used, it works as such. This string is dumped into the place where the for loop began.

{% … %} encloses statements in jinja. {{…}} encloses expressions. {{# … #}} encloses comments (not included in the output string)

The jinja if statement works very similar to the python if statement.

```
{% if kenny.sick %}
    Kenny is sick.
{% elif kenny.dead %}
    You killed Kenny!
{% else %}
    Kenny looks okay --- so far
{% endif %}
```

Another commonly used statement is the block statement. This is used in order to reuse templates within templates, making for some very compact string formatting.

Base Template

This template, which we'll call `base.html`, defines a simple HTML skeleton document that you might use for a simple two-column page. It's the job of "child" templates to fill the empty blocks with content:

```
<!DOCTYPE html>
<html lang="en">
<head>
    {% block head %}
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}{% endblock %} - My Webpage</title>
    {% endblock %}
</head>
<body>
    <div id="content">{% block content %}{% endblock %}</div>
    <div id="footer">
        {% block footer %}
        &copy; Copyright 2008 by <a href="http://domain.invalid/">you</a>.
        {% endblock %}
    </div>
</body>
</html>
```

In this example, the `{% block %}` tags define four blocks that child templates can fill in. **All the *block* tag does is tell the template engine that a child template may override those placeholders in the template.** (placeholders = the contents inside the block tag. They will get discarded if a child template comes along and fills it)

`block` tags can be inside other blocks such as `if`, but they will always be executed regardless of if the `if` block is actually rendered.

Child Template

A child template might look like this:

```
{% extends "base.html" %}
{% block title %}Index{% endblock %}
{% block head %}
    {{ super() }}
```

```
    <style type="text/css">
        .important { color: #336699; }
    </style>
{% endblock %}
{% block content %}
    <h1>Index</h1>
    <p class="important">
      Welcome to my awesome homepage.
    </p>
{% endblock %}
```

The `{% extends %}` tag is the key here. The extends tag should be the first tag in the template. It tells the template engine that this template "extends" another template. When the template system evaluates this template, it first locates the parent.

Then, it builds the parent normally, but it fills in the blocks of the parent template with the blocks defined in the child template.

That covers the basics of template inheritance.

Important notice: Do not be deceived, the jinja statements used everywhere (like if), as well as the expressions used in a jinja if statement, and the basic operations like +,- etc are not just python code. Jinja takes python objects inside and lets you do very pythony things with it, but **jinja has its own syntax for everything.**

For instance, {{len(my_list)}} will NOT work. It's {{my_list | length}} because jinja doesn't have python's len function; Jinja has its own way of doing things. The methods and attributes of python objects that are passed can still be called, but python functions, cannot. *Jinja functions, can.*

So, if we have a custom made function, we cannot call it inside a template?

No, that's not the case at all. In python, everything is an object, even functions. Try defining a function, and print(function_name) and print(type(function_name)). You'll find that the function is actually a object of a class of objects called functions.

In fact, classes are also objects. So, you can pass a class object into the jinja template, and do *classy* (xD) stuff with it.

Moving on, the | used in the if statement needs to be discussed. It applies something called a filter.

Variables can be modified by **filters**. Filters are separated from the variable by a pipe symbol (|) and may have optional arguments in parentheses. Multiple filters can be chained. The output of one filter is applied to the next.

I'm pulling all this from the official documentation of Jinja2.

Read it properly.

3.2 Flask

Now that we can use jinja2 with some level of familiarity, we can move onto Flask, another incredible library built on top of jinja that acts as a **web server**.

Before I get into the details of flask, I'd like to get some python knowledge issues out of the way. As I mentioned earlier, functions are actually objects. This is profound. This means that you can pass around functions like they're regular objects, and do things with them. For instance,

```
def timer(func):
        start=time.time()
        func()
        print(f"function took {time.time()-start} seconds")
```

Takes a function as an argument, and also prints the amount of time the function took to execute. What this has done in essence, is **wrapped** a function into additional functionality.

This is however, not what we typically want. What we want, if for a function to take a regular function as an argument, and return the **wrapped function**. The wrapper is like a chocolate wrapper. (It's a noun not a verb.)(That keeps tripping me up) That makes the function we seek, a **wrapper wrappifyer**, known as a **decorator**.

A decorator would look like

```
def time_decor(func):
        def timer():
                start=time.time()
                func()
                print(f"function took {time.time()-start} seconds")
        return timer
```

It's returning timer, a function object, which now acts as a wrapped function. The time_decor function has successfully decorated our function.

In general,

```
def decorator(func):
        def wrapper():
                #stuff
                func()
```

```
                #more stuff
        return wrapper
```

(PS: I kinda wanna name the wrapper function wrapped_function instead of wrapper, coz the chocolate, the function, is already inside the wrapper. Returning the wrapper at the end, is like dishing out a wrapped cholocate, not just some aluminium foil. But that's just me obsessing over the way the return sentence sounds like. The wrapper function has a provision for a chocolate to go inside, and it's still the decorator that puts the chocolate inside the wrapper and sends it off.)

It's still missing one crucial piece. The wrapped_function isn't taking any arguments. We are calling the function being wrapped with no arguments. When a decorated function is called with several arguments, those arguments need to passed down into the wrapper. The output also needs to get returned, just like in the original function. This means that the general code looks more like

```
def decorator(func):


        def wrapper(*args *kwargs):
                #stuff
                value = func(*args *kwargs)
                #more stuff
                return value


        return wrapper
```

To use a decorated function, decorator(my_function)( ) is done, with the extra pair of parenthesis calling the decorated function object that comes out of the decorator.

This makes for some silly syntax, if you always want to decorate a function.

Well, why not just code the decorator stuff inside the function then? What's the point of decorators? It doesn't make sense to keep decorator code inside every single function, if the decorator is reused across a large number of functions. For a very complex task, the decoration code can get very intense and long and it makes the actual function unreadable if we were to shove it in there. It's a powerful separation of concerns, and an elegant reuse of code.

Okay, so what's python's solution to the painful syntax? Merely type `@my_decorator_here` before the function's definition. This line says "This function here, starting in the next line? Always wrap it with this decorator."

Simple?

Good.

Check this out for details on what `*args` and `**kwargs` mean.

PS: To be perfectly honest, you can do flask without knowing any of this.

Flask in itself is very straightforward. You will have a webpage. It's going to have a bunch of pages within the domain. When each page is accessed, something needs to be done. That something is defined by a function. For instance,

```
@app.route('/home', methods=["GET", "POST"])
def myfunc():
```

The "app" object is a Flask thing. Route is a decorator function of that Flask object, which in itself takes arguments. The first argument is the path of the webpage this function concerns.

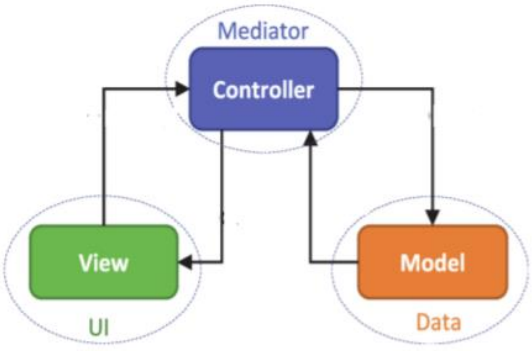
When anyone accesses that path, this function will be called. Accesses how? Those are defined in the methods argument. By default only "GET" and "HEAD" are allowed, since any web server necessarily responds to those 2. If a request is made to this page using a method not specified in this list, then a "405 Method Not Allowed" response is returned.

Thus, Flask hosts your web server, and renders each page on request. One very important thing I learnt during a fun project is that Flask is multithreaded by design. It's non-linear in some sense. The way I realised this was by observing that putting a sleep( ) call inside the response function did not suspend the entire code. Rather, every request sent to the server parallelly executed on a separate thread.

Flask is a very intuitive and simple way to a) Receive http requests from the client, and b) Send http responses back, rendering any webpage you want, both without ever having to worry about the nitty gritty of http.

We can even request other webpages for resources in the middle of the execution of a function, using this python module called **requests.** It's quite useful once the code starts becoming more complicated. We can separate functionality into APIs, and request content from them.

Designing a flask application using the guidelines of the MVC Paradigm is a great idea. Here's a nice image I found:



Keep your code compartmentalised. MVC is not a law, or a compulsory set of rules, but a guiding principle that allows a very natural separation of concerns.

The controller renders the view, when the URI mapped to a function is called. Any further requests behave similarly, going to the function we define, which then requests the model for data, or updates the model and then relays that back to the view.

Week 7

```
│   app.py
│   week7_database.sqlite3
│
├────application
│   │   __init__.py
│   │
│   ├────controller
│   │       course_controllers.py
│   │       student_controllers.py
│   │       __init__.py
│   │
│   └────model
│           db.py
│           models.py
│           __init__.py
│
│
└────templates
    ├────course
    │       add_course.html
    │       course_details.html
    │       course_home.html
    │       existing_course.html
    │       update_course.html
    │
    └────student
            add_student.html
            display_details.html
            existing_student.html
            index.html
            update_student.html
            withdraw.html
```

Above is the structure of my flask app I made for the Week 7 assignment.

The app.py kicks things off, booting the server, calling db.py which initialises the database connection, and then running the controller python files as a final step.

I was afraid to use os.path references to the templates directory of flask, so I just kept it outside at the default location, but I really wanted to put it inside the application folder so that there's just one folder containing MVC.

The models folder is self-contained. db.py initialises the database and Models.py just imports the database object from db and makes the class definitions for the database tables.

The controller codes import the Flask app for routing (from flask import current_app), the db connection and the models for querying.

Flask-sqlalchemy uses a syntax that doesn't require the db object, and only the models, but since I was migrating from SQLalchemy code, I needed it.

The "init.py"s are the secret heroes here, because they label a folder as a python package. They're just empty files. If I'm inside models.db.py, then

import .models import stuff

will work. The dot denotes "current package".

import ..controller.course_controllers

is valid as well. The double dot takes you the parent package. The .after controller lets me access the course_controllers file inside the controller directory. If I add another dot, I can import a specific class, function or variable form within the python code.

Note that you don't need to put an extra dot at the first stage even if we are descending a level. That is, (..).controller is wrong.

One more interesting thing about Flask is that it uses this thing called a WSGI (Web Server Gateway Interface)

- A WSGI container is a separate running process that runs on a different port than your web server.
- Your web server is configured to pass requests to the WSGI container which runs your web application, then pass the response (in the form of HTML) back to the requester.
- Flask uses the WSGI toolkit called Werkzeug. (German for tool)

It's basically some form of interesting port forwarding going on here. It's at a level lower than what we are concerning ourselves with now, so I won't get into it.

Having discussed the View and the Controller, let's set up the model.

3.3 SQLAlchemy

If the objective is just to connect to a database, make queries, and retrieve or insert data, a simple SQLconnector will suffice. We will have to manually create SQL queries as strings in the python code, and then query the database.

This is a totally valid approach adopted by several packages.

We will be doing something much more powerful. It's called Object-Relational Mapping where the relations of the Database fall into our lap as python objects.

We can then manipulate and interact only with python objects, without having to write an SQL query. To be fair, I liked SQL queries. They allow for a surprising level of complexity with very elegant syntax. I'm sceptical of the idea that they can be replaced, but we will see.

The actual database being used to store the data in the background might be MySQL, PostgresSQL or SQLite or some other framework. The cool thing about SQL alchemy is that it is database agnostic, meaning it doesn't matter what system is in the background, you just use very general python syntax, and when you connect the python code with the specific database, the syntax takes on a very specific meaning.

This is pretty much 100% true. You can code the entire thing while testing on SQLite, and then launch it into a server working with MySQL. Just change the connection creation statement and you're done.

SQLite is a very popular choice for this purpose because it's fully self-contained and resides on your system. We will use SQLite to build our apps.

SQLAlchemy is the core package; flask-sqlalchemy is built on top of it with some additional functionality, and a lot of alternative syntax.

Most of flask-sqlalchemy is very simple; it's putting SQL on python. Only the syntax needs to be learnt. One thing stood out to me: the way you handle relationships.

This is typically how a table is modelled in python:

```
from sqlalchemy import Table, Column, Integer, String


class Student(Base):

    __tablename__ = "student"

    student_id = Column("student_id", Integer, primary_key=True,
autoincrement=True)
```

```
    roll_number = Column("roll_number", String, unique=True,
nullable=False)
    first_name = Column("first_name", String, nullable=False)
    last_name = Column("last_name", String)


    def __init__(self, roll_number, first_name, last_name):
        self.roll_number = roll_number
        self.first_name = first_name
        self.last_name = last_name
```

Let's say another table, faculty is made. Some faculty are the advisors of some students. Many students can have the same facad.

A one to many relationship is handled by adding the primary key of the one attribute on the many side table, here, adding the facad id into the student table in a new column.

A one to one relationship can be handled in the same way, with the choice of adding the extra attribute to either table.

A many to many attribute needs an extra association table, where the primary keys of each table are mapped to the other. Here, that would be a table with facad id and student id, and any other attributes of the relationship itself, such as the name of the project the advisor is supervising.

By adding a "relationship" to the table declaration, you can do extremely cool stuff at the python code level. To access the facad of a given student, you can simply do

my_student_object.facad_id

I'm starting to think this ORM stuff is cool, mate. Maybe, SQL queries can be replaced after all.

3.4 APIs and Flask-Restful

APIs, Application Programming Interfaces, are brilliant ways to abstract away large portions of your code.

When it comes to coding and maintaining a large webpage, it becomes necessary to have a good separation of concerns both so that several people can work on pieces independently, not constantly worried about what code others write, and because it is much easier to debug and narrow down the problem if a problem should arise.

APIs perform a specific job. You make a request to them, they do the job and tell you whatever you need to be told. How exactly the API is implemented is not of concern when it comes to the rest of the code. Inside the API, you can tweak efficiency, style of code, hell, maybe even the language of the code; doesn't matter. The inputs and outputs are clearly defined. That's all that matters.

APIs are immensely useful inside an application. Beyond this though, they act as means by which the public can access your application and build something cool on top of it. For instance, Wikipedia has an API which lets me directly pull out information on a particular topic, make search queries and such things. If they didn't give me access to their API, when I want to make an app that uses Wikipedia for something, I'd have to manually poll their pages, scrap info out of the html and go through some painful code.

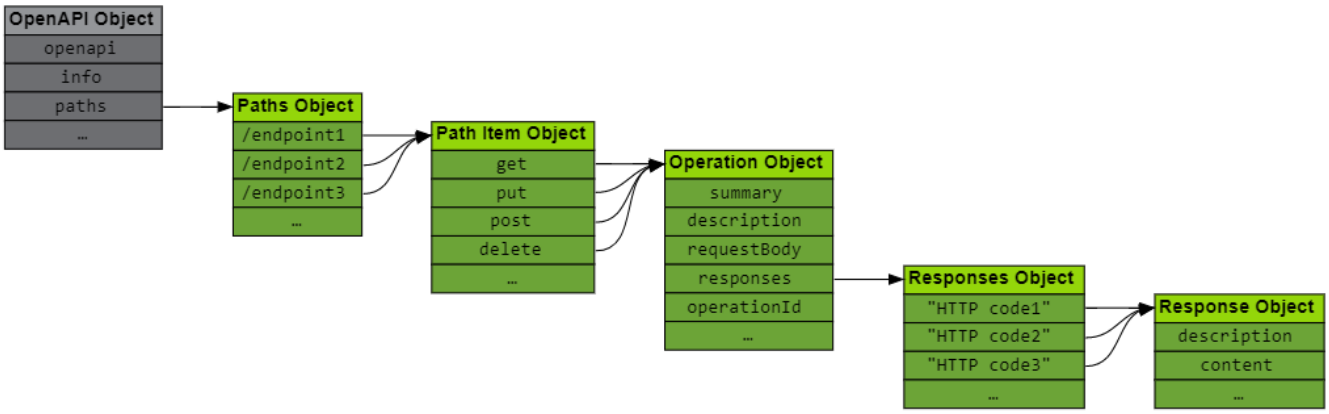
Apart from building on top of the app, the API can let you achieve cool functionality for your personal requirements, which fall within the basic capabilities of the app, if you wish to code it. For instance, I used Whatsapp's API to automatically DM myself the day's mess menu every morning.

These APIs are offered as a public service. Of course, if you abuse the API and send an absurd number of requests to it, they'll probably blacklist your IP.

Since APIs are meant to be used easily, they are well documented. In fact, there is a systematic means to document APIs, called the OpenAPI standard. The documentation itself is kinda like code. It specifies in detail how to make use of every functionality of an API, how it will respond, and what data it will give, sometimes along with some examples.

The OpenAPI documentation can also be used as a means to prescribe what you expect from someone when you ask them to build an API. Someone in the company may just hand you this file and say go build this API, and all the details needed will be in that.

It's highly recommended that you first make the OpenAPI documentation for an API before you even start coding it. Design first, code after that.



This is the basic structure of the Openapi yaml. Think of it like a python dictionary with smaller and smaller dictionaries inside them. Just like a json.

As is shown in that chart, the doc specifies every URI, every method, every operation performed when this is visited, the response type, and even example response. The image by the way, is from the OpenApi documentation. It's really good.

flask-restful allows you to make a http based API. Nothing terribly fancy. Still we just map functions to URIs and the package takes care of the http business.

REST stands for REpresentational State Transfer. It's a set of guidelines for API design. It imposes several conditions on the API. REST is not a strict rule or anything; it's a set of features and ways of functionality that are typically seen as desirable in an API.

The core feature of REST is that the server doesn't assume the "State" of the client. If there is any state the client is in, and needs to tell the server, that is just packaged and sent with every request. The State is Represented in the request and Transferred between the client and the server. Nothing else outside the request is assumed when it comes to satisfying the request.

HTTP requests are one way to implement a REST API. The HTTP verb very conveniently allows us to do the 4 most common operations we wish for an API to perform, Create, Read, Update and Delete, abbreviated as CRUD.

CRUD is not really related to REST; CRUD is just a set of operations we can perform on a database, which are commonly implemented in an API.

Typically, the response of a REST API is in the JSON format (JavaScript Object Notation). It's a very powerful datatype akin to a python dictionary.

That's it about REST and how we should make an API. Let's say we have an API and it's out for use, and we now wish to update the API. If we just went ahead and did this, it would break every script in the world that was relying on our code.

Because of this, the API URIs always carry with them a version number up front. An update is released as a new version, and the older one is kept up till some date, when it will finally be taken down.

To tie things together, in the larger picture, this can roughly be considered to belong in the model section of MVC. If you ever find yourself writing an SQL query in the middle of controller code, you should probably stop there and say "No; this isn't a clean separation of concerns."

Ideally, a change in the model should **not** affect the controllers or the view. The APIs can change, and that way your controller won't break even if your model changes. Only the model must interact with the database.

That said, practically in a flask application, this is not feasible to stick to. MVC is more of a way of thought than a strict set of rules. The great thing about SQLAlchemy is that as an ORM it abstracts away the database, so we are able to put "queries" without actually making a concrete SQL query in the controller code. That way, even if the model changes, the controller code remains intact.

3.5 Full Text Search

The web is big. Most pages have a search bar in them because they themselves carry content that is far too large to link in one page.

A straightforward way to search for content, is to actually scan all the available documents, using an algorithm such as KMP or Rabin Karp. Even with KMP's O(n) complexity, for a search engine like google to scan every page in existence would be infeasible.

That's where Full Text Search comes in. Right off the bat, I'd like to point out that this is an imperfect, but practical solution. There will be relevant documents that are not returned by the search, and irrelevant documents that *are* returned. The percentage of documents that are relevant in the search results is called precision. The percentage of relevant documents returned with respect to all relevant documents, is called recall. A high recall search will give almost all the relevant documents, but it might also end up returning a ton of documents that are irrelevant thereby ending up with low precision.

Now I'd like to briefly talk about how it works. There are 3 major stages I could note in this process: First, is tokenisation.

When I search for "giant yellow flower", I don't necessarily want all 3 words in every search result. It makes natural sense to treat space as "or", but clearly different weights need to be given to different words, plus, if more words match, then it's more relevant.

When I googled it, the first result was an image with the caption big yellow flower, from Wikimedia commons. So, we also need to search for synonyms, here, it realised big was synonymous with giant. The actual search results are mostly about yellow flowers.

Anyway, if I were to search for "how to grow giant and yellow flowers?" and the backend tried to simply "or" separate and conduct a query, we would get results with just "and", "to", or "how". Unimportant words that do not convey subject matter are called stop words. When a query string is tokenized, it's broken down: All letters are made lower case, stop words are discarded, and tenses and word endings are removed. For instance, growing, grew etc all become grow, the root form.

how grow giant yellow flower would likely be the tokenised form of the above query. Typically, "not" is considered a stop word, and most simpler FTS engines discard it completely when tokenising the query. Even in google, "not" carries a very low weightage. The results for "not yellow not giant not a flower" are still

articles about yellow flowers. The Images are very different though, and that's because of (I suspect) the fact that I used not in the query thrice, it started picking up images from pages with the word "not" in it, apart from flower, yellow and giant. The very second image, after a large yellow flower, is a forget-me-***not***)

To be fair, what else can the poor thing do, the list of not giant not yellow not flowers is basically infinite, and only a moron like me would search for such a thing ☺

Regardless, the point I wish to make is that more than sentences, key words are relevant. Do not think of it as words, they are "tokens", broken down, simplified, and modified words. An engine like google will use both position of words in the query and their order in the query as parameters. This is also done by most FTS engines. The tokenization process is different for each language; some languages like Japanese cannot really be tokenized easily, but god, we are not getting into linguistics here. A Japanese coder can make a Japanese tokenizer and it will work just fine with a normal FTS engine.

That's step one, tokenization. I'd like to jump a bit and go to step 3, scoring. As I pointed out earlier, different words have different weightages. If a word in your query is rare, and an article contains it, there's a high chance that the article is relevant. If a word in your query occurs frequently in a page, it is likely relevant.

Now, a function is designed to give a number based on these parameters. If frequency is higher, give a higher score, but beyond 10 occurrences, more occurrences do not matter much. Something like a log curve.

It doesn't necessarily make sense to give points for words directly. If "giant" is 3 points, "yellow" is 1 point, and "flower" is 5 points, a webpage about clash of clans which just has the word giant in it hundreds of times might end up as a result. Rather, a vectorisation is done based on the weightage. Let's use just two words, giant and flower. Points for "flower" go on the y axis, "giant" on the x axis. An article with just giants will be on the x axis, while something about just flowers will be on the y axis. Most relevant results are picked out based on the position on the plane, and less importantly, the magnitude of the vector. Where exactly should the relevant results go on the plane? Not sure. I think it is just a bit away from the axis of the most heavily weighted word, here flower. This process can be done with any number of dimensions, but I don't know the details of the exact cutoffs.

All sorts of factors like this, based on intuition and analysis of language patterns are put into the scoring function. SQLite's FTS5 uses the BM25 scoring function, which uses how important the word is (based on rarity) in the term called Inverse Document Frequency, how frequent the word occurs, but with low weightage beyond a point, and the size of the document itself.

The results with the top scores are returned as answers to the query.

Fuzzy searches also search for words that are similar, such as with a single or double letter typo, but assign them lower scores. Any synonym behaves basically as the same word as the query word, but typically gets a lower score.

That's about it. It's not so much magic as it is math. What I glossed over, was step 2, actually searching for phrases in documents, AKA, the hard part. Note that because we tokenized the query, and because of our function parameters, we need only the frequency of each token in every document to gauge a document's relevance.

This particular information is precomputed and stored. When we involve the indexing of documents, the searching process needs to check only the indices of each page, not the contents, making things tremendously faster. That however, is likely far from the end of the story. I am satisfied with this primer, though. For our purposes, we know far beyond necessary. I'm not up for reinventing Google as yet.

For the purpose of small to medium scale projects, simply use SQLite's FTS5. The documentation is pretty good, please read it. I hope it suffices. You really can't make it anywhere as a programmer without the valuable skill of scavenging documentation.

Part 4: Design Challenges

Welcome back to "Exploration time!" where we jump into rabbit holes and gain a deep understanding of the internet. The utility in this may seem ambiguous, but I am tackling this purely to satisfy my curiosity and access more sophisticated regions of study and development. Let's gooooo!

4.1 Certificates, HTTPS, and an Introduction to Cryptography

Let's say Alice wants to send Bob a secret message. She wishes to send it over the mail, but the mailman is super snoopy. Alice has an unbreakable box to put the letter in, and both of them have unbreakable locks. What should they do?

Pretty simple: Alice locks the box, sends it over the mail. The mailman cannot open it or do anything with this. Well, Bob cannot either, you say. Here's the fun part: Bob locks the box as well, at a second place, and sends it back. Poor mailman cannot do anything. Alice opens her lock, sends it back, and Bob can now open his own lock and read the message.

Clever.

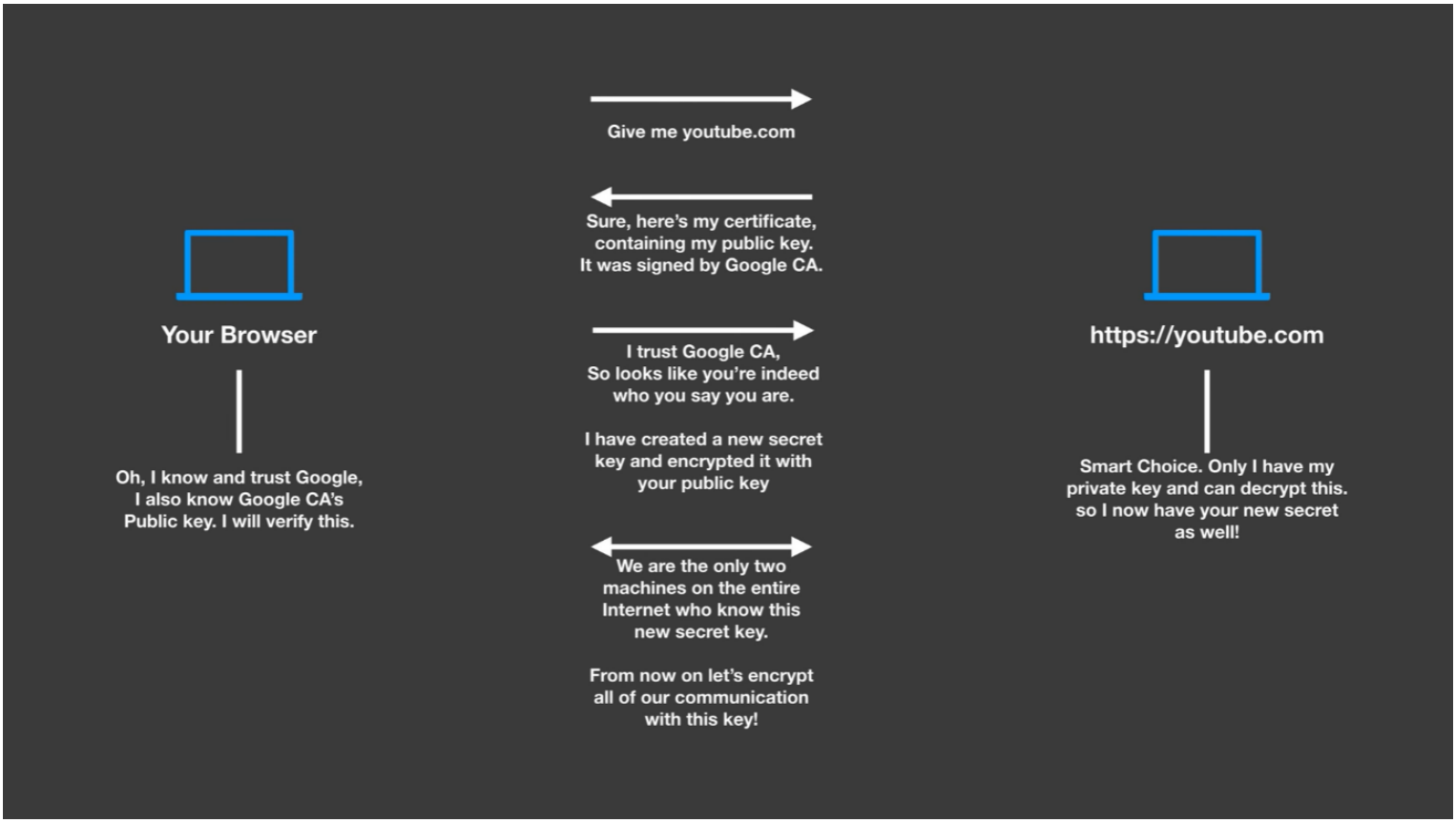
The mailman is now wise to what's going on here, and so does something even more clever: He picks up the box from Alice, waits for a day, and returns the box, *with his own lock*. Alice thinks that Bob has put his lock on, so she takes off her lock and gives it back. The mailman takes it away, and gleefully reads Alice's secrets, puts on a lock pretending to be Alice, gives Bob the locked box, and proceeds normally, and nobody's any wiser.

That's called a man in the middle attack.

You know a way to establish an unbreakable communication with *someone*, but really, it's pointless unless you can tell for sure *who* you're talking to. *Who* is locking these boxes?

The locked boxes explanation is just an analogy. As you can tell, certificates and https comes in and saves the day by allowing us to tell who's lock it is on the box. I would like to get into the actual details of encryption though, not just settle for some box analogy.

I'll quickly run through the certificate exchange before that.



I'm pulling this image from this brilliant video, which I have summarised below.

There are 2 facts you need to know before you understand this exchange:

1. A message encrypted with Alice's public key, can only be decrypted by Alice's private key.

Think of this like Alice keeping her unbreakable lock up for public use. Also, think of it like the public key to be akin to the product of 2 very very large primes. It's impossibly hard to factorise the public product, and access the factors. Alice however, knows the 2 factors, and can instantly decrypt the message. This exists solely to send Alice secrets only Alice can read.

2. A message encrypted with Alice's private key can only be decrypted by Alice's public key.

This is not to send secrets; It's for Alice to verify her own identity. It's like Alice's signature. Anyone with access to Alice's public key can be 100% sure that it was in fact Alice's private key that signed it, and it was this exact message that was signed.

Now, the real exchange:

When I ask for youtube.com, it returns a certificate identifying itself. This is the public key of youtube.com, its domain names and some other metadata, but encrypted with a certificate authority's private key. In this example, Google CA does the signing.

Think of it like a box that says "Open with Google CA's public key" and inside, lies a piece of paper with some scribbling that reads "I am u2b. Here is my public key."

This scrap of paper isn't worthless, purely because the Google Certificate Authority's public key opened the box, and only Google CA can make such a lock. Google has attested, for a fact, to **this particular message** being true, "I am u2b".

This cannot be faked, because only Google has Google's private key, and we trust that Google is not a troll.

It is impossible to edit the website name or the public key in the certificate to hijack the system: You'd need Google's private key to encrypt a message that can be decrypted by Google's public key.

The system is, at its core, based on the browser's trust of some list of certificate authorities such as Google CA, Verisign etc, whose public keys our browser comes pre-installed with. (Hmmm sounds like something hackable, eh?)

Now that we know that the certificate we got belongs to YouTube, we will use the public key enclosed in the certificate to encrypt our secret message. Only the *real* owner of this certificate knows the private key needed to decrypt this, so even if someone else pretended to be YouTube using the public YouTube certificate, it would be of no use, because they cannot decrypt our communication. Beautiful, isn't it?

Using this first bit of secret communication, we establish a shared key. That allows us to encrypt any further messages with a **shared**, secret key. A symmetric key encryption is much faster than other asymmetric means.

That's just about it.

Marvelous story, time to go home, except for the fact that I wish for math.

I'd like to make sense of the phrase "encrypt with a key" and derive the 2 facts I stated earlier, that a pair of keys can only decrypt each other's encryption.

It's actually fairly straightforward.

Before we dive in, let me sort out some stuff at the high level. There are 3 ways to set up a secure channel:

1. Asymmetric Keys

2. Symmetric Keys - blazing fast and basically bullet proof. But a shared symmetric key needs to be established somehow. How?

   a) Use an asymmetric key set up to communicate a shared key

b) Do some magic maths to make the 2 people derive a common key using only public information, which nobody else can do, due to their lack of private keys.

1. is fairly straightforward. We already saw this in action. A has a private key and a public key. B has their own pair. A can encrypt a message with their private key to sign the message, and on top of that encryption, encrypt it with B's public key, so that only B can decrypt it. A very simple exchange.

We needed the certificate system to attest to the fact that the public private key pair claimed to be owned by A, is in fact, only owned by A. Otherwise, a man in the middle attack would be possible.

This scheme is called RSA, after Rivest, Shamir and Adleman, the 3 inventors of the algorithm and it's based on modulo arithmetic, and the near impossible factorability of large semiprimes.

Roll with me for a while. Note that I will work only with integers, in case it is not obvious. Also, note that "a while" is code word for a 3 hour detour.

4.1.1: Setting up the Fundamentals

(1) Definition:

$$a \equiv b \bmod n \text{ iff } n|(a-b)$$

Read as "a is congruent to b modulo n if and only if n divides a-b"

(2) Random Property - Cancellation:

$$\text{if } ar \equiv br \bmod n \text{ and } \gcd(n,r)=1, \text{ then } a \equiv b \bmod n$$

Since $ar \equiv br \bmod n$, $n|(ar-br) \Rightarrow n \mid (a-b)r$ but $n \nmid r$, so $n \mid (a-b) \Rightarrow a \equiv b \bmod n$

(This may seem simple, but this is so profound that you can stop right here and prove Euler's theorem.)

(3) Seeking an Inverse

Given some a, can you find an x such that $ax \equiv 1 \pmod n$?

$6 \equiv 0 \bmod 2$.

Regardless of what I multiply 6 with, it will remain 0 mod 2.

$6 \equiv 6 \bmod 15$

Let's say $6k \equiv 1 \bmod 15$. Then, $15 \mid 6k-1$. The thing is, 15 is a multiple of 3. 6 is also a multiple of 3. If $15 \mid 6k-1$, then 3 also divides 6k-1. But 3 perfectly divides 6k. 3 cannot perfectly divide 6k-1. It's just one less than a multiple.

Thus, There cannot exist a modular inverse if $\gcd(n,a) \neq 1$

Let's now assume $\gcd(n,a)=1$

Does x exist such that $ax \equiv 1 \bmod n$?

Rephrasing it, do solutions exist for $ax - 1 = kn$?

We have the numbers a and n, and we need the coefficients x and k in the linear equation $xa - kn = 1$. Such an equation is called a Linear Diophantine Equation, LDE, and can be solved using something similar to Euclid's division algorithm.

(4) Euclid's Division Algorithm for finding GCD

Euclid's Division Lemma: given numbers n and d, there exists q and r such that

$n = q*d + r$ where $0 \leq r < d$

Goal: find gcd(a,b)

We know a = kb+r. If x divides a & b, then it must also divide r. thus, gcd(a,b)=gcd(b,r). Recursively doing this, we reduce the problem to a trivial one, where we are taking gcd of a number and its factor.

For instance, gcd(42,27)=gcd(27,15)=gcd(15,12)=gcd(12,3)=3

$42=27\times1+15$

$27=15\times1+12$

$15=12\times1+3$

$12=3\times4+0$

(5) Modifying EDA for LDEs

If instead of asking gcd of 42,27, I ask for 27^{-1} modulo 42, that is, Oh, wait. That would sadly be impossible as we discussed earlier, since their gcd is not 1.

Let's instead do 10^{-1} modulo 27. We want 10x-27k=1.

$27 = 10\times2 + 7$

$10 = 7\times1 + 3$

$7 = 3\times2+1$

$3 = 1\times3 + 0$ (Ignore)

$7 - 3\times2 = 1$

Rewrite 3 using the previous step. $7 - (10-7)\times2 = 1$

$7\times3 - 10\times2 = 1$

Now rewrite 7 using the previous step:

$(27-(10\times2))\times3 - 10\times2 = 1$

$27\times3 - 10\times8 = 1$

We found 10's inverse: It's 8.

In general, we can also view this top down.

From EDL,

$$a = b\,q_0 + r$$

$$b = r\,q_1 + r_0$$

$$r = r_0\,q_2 + r_1$$

...

$$r_{n-2} = r_{n-1}\, q_{n-1} + \gcd$$

$$\gcd = \gcd \times 1 + 0 \text{ (Ignore)}$$

Now,

$$(1) \Rightarrow r = a - b\, q_0$$

$$(2) \Rightarrow b = (a - b\, q_0)q_1 + r_0 \Rightarrow r_0 = b - (a - b\, q_0)q_1$$

$$(3) \Rightarrow (a - b\, q_0) = (b - (a - b\, q_0)q_1)q_2 + r_1$$

And so on.

In each equation, the two r terms are substituted from the previous 2 results. Eventually, we get to the point where only gcd is left over.

Thus, we can find solutions (x,y) for

$$ax + by = \gcd(a,b)$$

EDA and the extended EDA(this one) are both O(log(min(a,b)). Proof's irrelevant. Fact's also more or less irrelevant; relevant fact is that it is algorithmically solvable quickly.

4.1.2: The RSA Protocol

Let's say we have a message m, which we want to send to Bob. m is just a big binary number. Some integer.

Let's ask Bob to take 2 random primes, p and q, preferably gigantic, and make their product n publically available. n is Bob's public key, and p & q are his private keys. We will use Bob's public key n to encrypt our message.

Let's raise m to an exponent e, which is publically declared, and take modulo n.

Although m is considerably smaller than the giant n, when exponentiated, m^e shoots past that, and if you imagine a clock face with "n" as the period, m^e will cycle around many times and end up at a seemingly unpredictable spot, which we will call c.

Let $m^e \equiv c \pmod{n}$

We publically send out c. Public as in, we don't care if someone intercepts it: they can't figure anything out from it. The public, at the moment, know e as well as m^e mod n. If we sent out just m^e, then they can take an "e"th root. But because this is on a clock face, this is basically impossible to do, you would have to without brute force trying everything.

We have now happily sent "c" to Bob, but can he even retrieve the message m from it? This is where Euler's theorem comes in. (refer next section for more)

$$m^{\varphi(n)} \equiv 1 \pmod{n}$$

So, $m^{k\,\varphi(n)+1} \equiv m$

The idea is to thus further exponentiate c while taking mod n, that is, on the clock face, and eventually coming back to m on the clock face.

Basically, we must find d such that $ed = k\varphi(n) + 1$, and raise c to d.

Thus, d is just the modular inverse of e with modulo $\varphi(n)$, which is trivially computed. Thus the decryption is done, and Bob has the message. Since nobody knows prime factors p and q, nobody can compute $\varphi(n)$. Thus, nobody can derive the modular inverse d, and nobody can unravel the message.

Only the person with the private keys can do so. At this point we must note that the exponent cannot have been completely random: It must be possible to find a modular inverse for the exponent with respect to $\varphi(n)$, so it must be coprime with $\varphi(n)$. Typically, e is taken to be 65537, a prime.

Note that Alice is able to send secure messages to Bob, but bob can do so only if Alice sets up her own prime pair. Using RSA like that for all communications is silly and slow.

Since m cannot be larger than n, it's impractical to use this for larger messages. The limitation on the packet size and the level of computation for each packet is the reason the only thing communicated through this protocol is a shared key, which is then used by both parties to switch to a shared key cryptosystem.

4.1.3: Proving Euler's theorem

$\varphi(n)$ is the number of natural numbers smaller than n and co prime to n. For a prime number, all numbers smaller than it is co prime to it, so $\varphi(p) = p-1$.

For semiprimes, that is, the product of 2 primes, the only possible way to share a factor, is to be a multiple of one of the constituent primes. There are q-1 multiples of p lesser than pq. There are q-1 multiples of p lesser than pq. There won't be any overlap as the LCM is pq for two primes.

$$\varphi(pq) = pq-1-(p-1)-(q-1) = pq-p-q+1$$
$$=(p-1)(q-1)$$

It's not very hard to derive $\varphi(n)$ for any general n. Let's say 2 is a factor of n. Every multiple of 2 lesser than n is ditched out. The multiples of p occur once in every p numbers. Thus $n(1-\frac{1}{p})$ will be the left over numbers eligible for counting.

$$\varphi(n) = n \prod_{\forall p:\, p|n} \left(1-\frac{1}{p}\right)$$

We can easily verify that this works out for semiprimes.

Anyway, if gcd(n,a)=1, then $a^{\varphi(n)} \equiv 1$ is what Euler's theorem states. The Proof of this not very involved, but you'd never stumble on it unless you were playing elsewhere.

Firstly, here's a fun fact: The modular inverse of a number is unique. Meaning, a and b cannot both have inverse c modulo n. This is fairly obvious:

$ac = kn + 1$; $bc = tn + 1$; $t \neq k$ since $a \neq b$

Now, $(a-b)c = (k-t)n$. Since c is coprime to n, (otherwise it cannot have an inverse) n divides (a-b), meaning $a \equiv b \bmod n$. Same guy; Hence Proved.

Now, use this fun fact for finding yet another fun fact: Wilson's primality test:

$(p-1)! \equiv -1 \pmod p$ for any prime p

(p-1)! has all numbers from 1 to p-1. Each number can be paired with its modular inverse resulting in no contribution to the modulo. The pairs cancel out, leaving behind only 1 and p-1, which are their own inverse.

Thus $(p-1)! \equiv p-1 \equiv -1 \pmod p$

Where did we use the fact it's prime? Sneaky much? We used the fact that every number less than p ***has*** an inverse. If there are *factors* for p smaller than it, it won't be co prime to p, and thus won't have an inverse.

Now that we have Wilson's theorem…We will simply keep it aside and not use it. I told you, it's a fun fact.

Consider Fermat's Little theorem, a special case of Euler's theorem:

Given a prime p and any number a (not a multiple of p), then $a^{p-1} \equiv 1 \pmod{p}$

Proof: Consider the set S={1,2,3…p-1}

Now take the set {a,2a,3a,…(p-1)a} and take modulo p for all elements in this set. Call the new modded set T.

Now, I claim that no two elements in this set T are the same.

If r, s ∈ S produce the same result in T.

$ar \% p = as \% p \Rightarrow ar \equiv as \pmod{p} \Rightarrow r \equiv s \pmod{p} \Rightarrow kp = (r-s)$

But r and s are smaller than p. Their difference cannot be a multiple of p. r=s. Note that all the heavy weight lifting is done by the cancellation property.

Thus, we have p-1 unique values in T, but since there are only p-1 possible values for x mod p, set T is just a permutation of set S.

Multiplying all of set T we have,

$a^{p-1}(p-1)! \equiv (p-1)! \pmod{p}$

Cancellation of (p-1)! which is co prime to p, on both sides yields

$a^{p-1} \equiv 1$

No, we did not use wilson's theorem, we don't need it, I told you, it's for real, just a fun fact. This one is also a fun fact by the way. I could've directly gone to the proof of Euler's theorem, but goofing around is quintessential for growth.

Now this one should be doable without any guidance.

Prove Euler's theorem.

$a^{\varphi(n)} \equiv 1 \pmod{n}$ if $\gcd(a,n)=1$

Spoilers ahead.

It should be really obvious. It's the same proof as last time.

- Consider set S of numbers co prime to n

- Consider set T of a times numbers in S modulo n

- T is a permutation of S. This is again obvious becoz we can cancel a if two things are equal modulo, and no two elements in T are same.

- Multiply everything in T and S, equate and cancel.

I uhh think we can call it a day. Let me recap.

1. We started with the basics of modulo arithmetic. We note the existence of modular inverses and their uniqueness. We also observe the cancellation law.

2. We note that finding the modular inverse corresponds to solving a linear Diophantine equation, and this can solved using the Extended Euclid's Division Algorithm.

3. We observed Wilson's theorem using the uniqueness of modular inverses.

4. We proved Fermat's little theorem using the cancellation law, a proof with contradiction and a lot of clever construction. The same proof was extended to Euler's theorem.

5. Armed with math, we construct the RSA asymmetric key exchange system. A message m is exponentiated, taken modulo a large semiprime, sent over, and then decrypted using the modular inverse of the exponent with respect to the totient function of the semiprime, which is unknown to the world.

6. Certificate systems were set up to prevent clever mailboy in the middle attacks.

For certificate systems to work, we needed two facts. Encrypting with the public key can only be decrypted by the owner of the private key. This is obvious from what we have seen. The second fact, have not discussed: "encrypting" with the private key, or signing, can be decrypted only by the public key.

This fact seems to be much, much more involved, and the obligatory stack exchange post about how that works can be found here. And a complementary debunkathon can be found here.

4.1.4: The Diffie Helman Key Exchange

I mentioned earlier that there are 2 ways to establish a shared key, one to use an asymmetric key system to share a key, and the other, a magic math way of two people deriving the same shared key.

This is based on the difficulty of the discrete logarithm problem rather than the prime factorisation problem, which is what RSA relies on. Don't worry, both can be broken by quantum computers.

We are also yet to discuss any symmetric key cryptography system. At this point, I should probably stop and go back to y'know, actual AppDev stuff, but you want me to stop here and go back to what, MongoDB? ACID systems? Pfff. Let's do Diffie Helman, it's not even that hard!

We're going to start at this simple property: $(g^a)^b=(g^b)^a=g^{ab}$

Let's publically announce g. Let's say a is my secret, and b is Bob's secret. I can do g^a, send it over to Bob, bob raises that to b to get g^{ab}; meanwhile bob does g^b, sends it over to me, I raise it to a and we both arrive at g^{ab}, our common key.

The issue is that g^a , g^b and g are out in the open. We can simply take the logarithm of g^a base g, and raise g^b to that power to find the shared key, and break all the communication encryption. To crack the encryption, you just need either a or b.

This taking logarithm business, by the way, is not so obvious: Do you know what power of 3 is 19,683? How would you find it? Don't use a calc. You'd be listing out powers of 3, right?

There are better algorithms for logarithms, of course, but there is **none** for a **discrete logarithm**. If we use a tremendously large number n and take modulo with respect to that number at every stage, the logarithm is no longer easily invertible. So we transmit g^a mod n, g^b mod n and the shared key of g^{ab} mod n is established. Of course, n needs to be public.

However, you should note something: Powers are cyclic in modulo. This is something you already know if you have seen those problems asking things like the last digit of 7^{324}. We know that 7 has a set period in powers, it goes 7 9 3 1 7 9 3…, Only 4 long in mod 10.

5 for instance, has a single number cycle modulo 10, it's only ever 5. This suggests that some values of g when we use mod n might be a dead giveaway, or atleast severely narrows down the values of a or b from the observations made on g^a mod n and g^b mod n.

That is why g is chosen to be a *generator* of n, a number that cycles through the maximum number of possible values for mod n in its power cycle. Not all numbers n have valid generators.

The multiplicative group of integers, mod N, is the set of numbers lesser than N, co prime to N, together with the operation called modulo multiplication. Including the operation here is because of the mathematical definition of a group, but we can go down that rabbit hole some other time. It's pretty deep, I've checked.

For 10, the mod 10 group consists of 1,3,7 and 9. That's a size of 4. A member of this group, which cycles through the whole group, is called a generator. Both 3 and 7 are valid generators of this group.

The mod 8 group, sadly, does not have any generators. The numbers in its group are 1,3,5,7. 1 stays on itself. 3 goes 3,1,3,1… 5 goes 5,1,5,1… 7 goes 7,1,7,1…

In fact, n can only be of 2 forms in order to have a generator, apart from 2 trivial exceptions:

n=2, which has (1) with 1 as the generator

n=4, which has (1,3), with 3 being a generator

n=(odd prime)^x, or n = 2*(odd prime)^x for any value of x.

This generator business is just to maximise the confusion in trying to invert the discrete logarithm. With a large enough n, brute force becomes simply infeasible. That's all Diffie Helman is. Clever exponentiation.

By the way, since we have come this far, I'd like to point out something about generators: the maximum cycle being the number of co primes smaller than N, that is, the totient, is no coincidence. Any x co prime to n, raised to φ(n) will become 1 mod n, at which point reaches x again.

The fact that the period of a number when exponentiated modularly is one more than the totient function, can allow us to crack RSA cryptography. RSA was secure purely because φ(n) is needed to compute the modular inverse of e, and there's no way for you to know φ(n) without p and q, and factorization is impossibly hard, or so we thought.

How to break RSA?

Step 1: Pick a random number a. We are praying to god that this is a generator.

Step 2: Raise it to powers mod n incrementally, and find the period. Take r = period -1. r is the **smallest number** such that $a^r \equiv 1 (\text{mod } n)$.

Step 3: Check if r is even, since (p-1)(q-1) absolutely has to be. If it's not, discard and pick a new random number a.

There's one more check that we need to do to ensure we have a generator:

Check if $a^{r/2}+1$ is **not** 0 mod n, if so, discard this a, and repeat with a new random number. Let me explain why this makes sense.

$a^r \equiv 1 \pmod{n} \Rightarrow a^r - 1 = kn \Rightarrow (a^{r/2}-1)(a^{r/2}+1) = kn$ (Since r is even)

Now, either $a^{r/2}-1$ or $a^{r/2}+1$ are multiples of n, or, alternatively, $a^{r/2}-1$ has only p in it, and $a^{r/2}+1$ has only q in it. (or vice versa)

That is, GCD($a^{r/2}-1$, N) = p and GCD($a^{r/2}+1$, N) = q, which means we can easily determine p and q.

In order for this to work, we need assurance that neither of the factors are multiples of n. ($a^{r/2}-1$) cannot be a multiple of n, since that means that $a^{r/2} \equiv 1 \pmod{n}$ but r is the **minimum number** that satisfies that. This only leaves ($a^{r/2}+1$), which we need to manually check and discard.

If either of these checks fail, you chose a bad starting number (Not a generator). Try again with some other number a. If you pass both these checks, congratulations, you just broke RSA.

This discarding process may seem brutal, but practically, there is a 50+% success rate that you pick a good value of a.

The issue is actually in Step 2, which is ridiculously hard. Exponentially hard, in fact. Manually going through powers, finding the period is expensive.

A Quantum computer can do Step 2 in the blink of an eye. What we just outlined is something called Shor's algorithm, which uses a Quantum Fourier transform to achieve step 2. This process is not very hard to understand but I'd have to use handwritten notes to explain. Neilsen and Chuang's QCQI did a nice job; if I get around to making notes, I'll link it here.

4.2 Cybersecurity

Boy, oh Boy, is this a big deal.

Since this is a little too broad, let's start at a fundamental idea. What is security all about? Preventing people from accessing things they do not have the authority to access.

Access control conceptually, is typically "role based". People of certain ranking or role can only access some content. A simple example: Linux permissions.

Alternatively, it could be attribute based, for example, a time window during which a competition's submission page is open, which is based on the attribute "time".

Regardless, some form of checking is done to see you have the authorisation to access some data. There are a number of ways this can be done. When I ssh into other systems, I use an address based access control. Only a list of known hosts are allowed to ssh into the Abhiyaan vehicle.

Purely address based access is a bad idea. That's why we use a password on top of it. An ssh private + public key system can also be used to authenticate who you are. Each of the known hosts has a registered public key on the vehicle, and if you can prove your identity with your private key, you'll be let in.

A password system has a few vulnerabilities. First, I have the password on me. If someone snoops around and finds it, looks over my shoulder when I type it etc, the security is lost. That's on the silly end. People are typically dumb and reuse passwords. Check out haveibeenpwned.com, to find all the data breaches your personal data was leaked in. This data is sold on the dark web to hackers, who will just try your passwords on every site possible. (That's if the organisation lost password data in the breach, which shouldn't happen if they used a good hashing system. Buuut given that even Adobe did some dumb unsalted hashing a while ago, and lost data in an embarrassing leak, you might as well give up on your faith in humanity. A lot of other private data like location, age gender and such however, are pretty much open for all in such a breach; no hashing, no escaping the loss.)

Second, I input the password on my computer. If my computer is compromised due to a virus that is constantly watching my screen, or is stealing data at the browser stage, from your end itself, the security is compromised.

Third, I send over the password to the server. This is an incredibly secure step, thanks to brilliant encryption cryptographic protocols.

Fourth, it goes to the server, and gets verified against a salted hash stored in the database. The salted hashing is to prevent a database breach from ever disclosing sensitive data. (Hashing was discussed in System Commands, but it was so off topic, I've included it in the SideQuests document as well.)

It feels like 1,2 are really only avoided if we cautiously tread the internet and have a good antivirus system on our personal computer. This is the most common place an attack can occur. There a billion ways to fool us into doing something dumb, and making us compromise the integrity of our own system.

For instance, you might get redirected to a webpage that looks exactly like the site you wish for, recreated by attackers to perfection, just with a url that is ever so slightly different. If you don't notice the fake url, and give away your credentials, you're doomed, and you won't even know it. The page will make note of your credentials, log you in to the actual site, and send you on your way without a hitch.

Let's say that steps 3 and 4 are unbreakable. Even so, as and when new features are added, more points of failure are introduced. For one, nobody wants to type their google password every time they open Gmail. That's why the browser sets a cookie claiming that you are you, and when you open the page, the cookie is sent to google, and they go, "Yeah seems legit, let em in"

What I log in as myself, and edit the cookie to have someone else's mail ID? A naïve system would just log me in as another user. It should be impossible to edit a cookie.

Even trickier, what if I make a cool webpage, which you visit, and on my page, I put some javascript in the background that goes and makes a request to your google account, asking it to share some pictures with me? Your computer has the cookie. It's gonna think it's you making these requests, when really, a script from another webpage is triggering these actions from your system.

This is a cross-site request forgery (CSRF) attack, which is countered by checking the source of the request and blocking suspicious ones.

Step 2 breaches: There are a lot of very clever ways of breaching into the user end, but it's a much more invasive and predatory type of hacking, because most of them can't be patched, but rely on user caution as a fix. Due to poor digital awareness, almost everyone is left vulnerable to many exploits.

Step 4 breaches: This is a more glorified robin hood-esque thing, and often lead to exploits that are patched and improve security. Some hackers don't work with the companies, of course, and just expose their data, sometimes revealing secrets that burn them to the ground.

Before we move on, some resources: Hackthebox, tryhackme for some hacking practice, and OWasp for up to date vulnerabilities. OWASP also has some cool projects you can learn from, I'd recommend starting with Juice Shop. That one project has basically all the major vulnerabilities you'll run into, a hundred or more in fact, ranked from one to 6 stars. 1 and 2 star challenges are a great starting point.

Bugcrowd, Hackerone and Synack maintain comprehensive lists of the bounties put out by various companies for bugs and vulnerabilities you can find in their code. Try facebook, swiggy, ola etc etc.(Btw, please don't go and hack into the official pages of companies, that's illegal, bug bounties will have an attackable page available. Do it there.) (If you ever get that good, that is.)

If you're looking for a reference book, go for "The Web Application hacker's Handbook – Discovering and Exploiting Security Flaws" by Dafydd Stuttard and Marcus Pinto. If you're not a fan of books, use this site. I've not read the book myself, so I cannot comment, I've just been told that it's comprehensive and good.

Hacking is basically a death note like game of "I knew you were gonna do that so I did this!" multiplied 100 fold. Fun stuff. Every place where you can type things in a webpage, you attempt to cause chaos and destruction. Put javascript code in search bars and see if it executes, put sql queries in the username box, see if it executes, put scripts hidden with pictures and make a comment on a social media site which will now run on every machine that views the comment, etc etc etc.

The list is endless. Very very fun. Maaayybee we should be *developing* applications in MAD instead of *destroying* them? As vital as defending against these well known attacks may be, should I discuss something else? Probably. Let's move on.

4.3 Building the Right Model

At the get go, I'd like to state that SQL databases rock, and need careful design to shine best. The two major parts of that design, in my eyes, are schema design and indexing. Indices are hand crafted for expected query patterns.

"I am a developer; indexing is my concern" is just one of the many golden nuggets I found on this talk. That said, I've discussed that talk and more in the DBMS document. Further discussion here won't be on the same lines, which means, of course, that we will look into NoSQL, which is en vogue for a variety of reasons.

Firstly, the key feature of the so called SQL databases, is the tabular structure, with every row having an entry for each field.

(Side Note: I'm not sure it's right to call them SQL databases, because SQL is just a structured query language. The querying in NoSQL databases is directly inspired by, if not exactly the same as the original SQL in a lot of occasions.

Rather, this is a transition from RDBMS to something beyond. What started as NOt SQL, is now viewed to be Not Only SQL. Regardless, we will use the popular lingo.)

NoSQL databases depart from this in that each entry need not have the same attributes. What was a Table before, is now a "collection"; a collection of objects with potentially, totally different attributes.

Why would someone want something like that? Flexibility. If I want to store student details in a table, I might want to add hostel name, mess name, registered duration and other things. But if that student happens to be a day scholar, they'll have 5 null attributes. They'll instead have some other attributes not applicable to hostellers too, which causes nulls everywhere.

To resolve that, we would have to split it to 3 tables: one for hosteller details, one for day scholar details, and one master student table with common attributes. We need to make a join when we need the data. Joins, Joins everywhere.

NoSQL doesn't need joins. It just has everything you need right there. Every entity is a JSON object, with each attribute potentially having sub attributes, because everything is JSON. (such as address: { street: no: locale: city: country:}) (At least I know that JSON syntax is adopted in MongoDB)

As you can imagine, this can only be born if we make some tradeoffs. What are we trading? I would imagine range based queries are harder, but MongoDB actually allows for range based queries and even indices. A lot of other NoSQL systems don't support such stuff, btw.

There are 4 principles prized by RDBMS systems when it comes to transactions: atomicity, consistency, isolation, and durability: ACID.

Atomicity means that you can guarantee that all of a transaction happens, or none of it does; you can do complex operations as one single unit, all or nothing, and a crash, power failure, error, or anything else won't allow you to be in a state in which only some of the changes have happened. This is especially relevant when millions of data points are changed as part of a single command.

Consistency means that you guarantee that your data will be consistent; none of the constraints you have on related data will ever be violated. This is somewhat hard to meet, and is often sacrificed by NoSQL systems.

Isolation means that one transaction cannot read data from another transaction that is not yet completed. If two transactions are executing concurrently, each one will see the world as if they were executing sequentially, and if one needs to read data that is written by another, it will have to wait until the other is finished.

Durability means that once a transaction is complete, it is guaranteed that all of the changes have been recorded to a durable medium (such as a hard disk), and the fact that the transaction has been completed is likewise recorded.

There could be ACID complaint NoSQL systems: the two concepts are disjoint. It's just that NoSQL typically stores data in a very interesting manner, which offers a lot of benefits, but sacrifices on some of the ACID rules.

For instance, let's say that I add person X, who lives in the US, as a friend on facebook. Another person Y, also adds X as a friend at the same time. Maybe, Y gets added to X's list earlier. Maybe, I get added earlier. There is a chance that you view X's page, and see Y as a friend, but not me, because an update that happened in the server closest to me, did not communicate and complete the transaction over all servers everywhere. There are 2 separate server versions which now need to reconcile and ensure things are good, nothing was violated etc.

Because these 2 disjoint, simultaneous operators can potentially violate an integrity constraint, the database has lost consistency. It's not a big deal in this example, moments of not being in X's friend list may make me a bit bitter, but nothing more. But if I take $1000 from X, and so does Y (Rather, X is paying us, but let's just imagine I am pulling money out.), and because of distinct servers, X magically pays $2000 while only having $1000 in his account. Maybe now one of the transactions need to roll back. Money that I just got, might get taken away. This kind of nonsense is unacceptable. Consistency is paramount.

NoSQL systems are "Eventually consistent". If you wait for 10s, you can be on X's friend list too. Just to oppose ACID, an acronym was made to represent NoSQL

systems, BASE: Basically Available, Soft state and Eventually consistent: a bit too contrived, if you ask me.

The idea remains that the focus is on highly available data, so multiple servers are the go to choice. This is called "Scaling out", as opposed to "Scaling up", where the server is just beefed up many fold to deal with greater demands. Scaling up is done for use cases where ACID is absolutely essential, and Scaling out, for non-ACID use cases.

Some use cases could end up using ACID for only some parts of their system. For instance, Amazon could store most of the product data in a system that is non ACID compliant and spread out, leading to a better user experience, but have only the final transaction stage under ACID. They cannot bill you and then say, ah, the database got updated the moment you started paying, lol, no pieces left for you.

4.4 Scaling

When it comes to scaling up a website, the tangible nature of the world catches up to you.

Problem 1: Latency

Typically, the bottleneck in speed in terms of user experience is in the network. The latency in this is very important, not only because a response needs to come to the user once, but because the TCP feedback loop might require much more than one back and forth depending on the lossiness of the communication channel.

With users increasingly spread around the globe, it would be impossible to have a low latency solution with one single server machine. Keep in mind that the length the information travels is that of the cables which will be much more than the distance by land. These cables will have repeater units that receive and retransmit the signals to make sure the signal strength doesn't die out. This needs to be done every few km, much less if it's not optic fibre that's carrying the information.

Problem 2: Throughput

Let's say that each request to your website warrants about 1KB of data sent in response. If the server has an internet connection of say, 100 Mbps (A fairly standard high speed connection), it's just not going to cut it. That can handle only around a puny 10,000 requests a second. (small b is always bits, capital is Bytes.)

Let's consider something more realistic. Google gets say, 60,000 requests/s. That's not the worst case, it's just a very safe assumption we are making now.

Each request warrants approximately 144kB of data in response. (obtained by inspecting traffic in the browser) That means that every second, the google server needs to have a bandwidth of about 70 Gbps. Much much more if we account for peak traffic.

It's unlikely that a single data centre is even capable of such a bandwidth.

Problem 3: RAM

Consider the case of YouTube. Running a video, for instance, takes up some amount of RAM in the server's system. It needs to fetch the video you asked for and continuously serve it to you.

An optimistic estimate of the size needed per user is 6 MB. Some popular livestreams garner millions of users. Let's say 10 million people are watching YouTube videos right now. That's 6e7 MB ≈ 60 TB of RAM.

Good luck building a server with 60 TB of RAM.

Problem 4: Actual Data:

We've been doing this "Model" business for quite a while. Where will you store the database? Every YouTube video ever? Every website indexed on Google? How much of space do you think that will take up?

Ridiculous.

Of course, this data cannot be just stored once. You need a backup. And a backup for that, maybe. At that scale, corruption will be a daily affair. You put a file in google drive, google cannot come to you and say, "Ah sorry it got corrupted. It's gone now." It's just not an option. And still these guys dish out 15 GB of free storage to ***every*** google account.

SSDs are not an option. There are serious reliability and financial issues. But still, I can download things from drive at very good speeds.

One hard drive is slow, but what if you split the data across 10 HDDs? Then, you can access the same data 10 times as fast. There are systems for optimising speed, redundancy and reliability in large servers. At the small scale, I know it as RAID. I assume an extended and more sophisticated version is used for larger systems.

Regardless, the sheer volume of storage required to handle a big webpage is a formidable foe to counter.

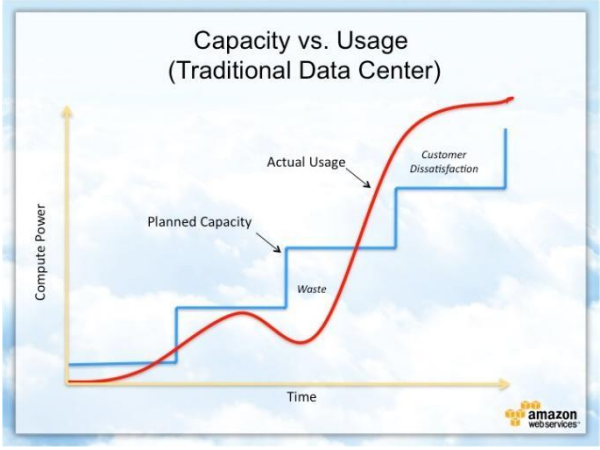
The solution all 4 problems mentioned is of course to build a *ton* of servers. Store most crucial your data on *all* of them, so that it's possible to serve people with very low latency, the collective throughput is very large, and the overall RAM amounts to these monster numbers.

This distributed server system is typically implemented with a lot of role division. Some machines are made attuned to receive a high number of requests, which acts the front facing machine, while another is a dedicated database machine. There is typically much more than one database server, and many processing machines. One single "load balancer" acts as the front facing machine, whose IP is more or less constant, and all the myriad machines behind it are abstracted from the user.

The costs of actually making a webpage is immense, if you wish to scale. First, you need to buy the expensive machines that are tuned for this job. Then you build a place where you can keep them, employ staff to maintain them and replace faulty parts once in a while. You need to maintain a temperature, cool the place and ensure that accidents don't disrupt the servers. You'll also need to employ security for these buildings.

You'll also need to make sure that you do this in a lot of different countries, and have systems in place for when one or more of the server machines crashes. It needs to always be the case that any user asking for the page gets it.

In the dawn of the internet, people weren't really sharing resources, and scaling to become a big webpage was incredibly hard. It is no longer the case, and it is possible to host servers very reliably for low prices (starts at $5/month). This is the market AWS and Google capitalise on. They have the servers ready. They can now sell it to you, and it's a win win.



If the company making the webpage actually hosted the page, their upgrades would be like that blue staircase. These companies match any needs you have and charge only for what you use. The sheer economic feasibility is why AWS dominates the web. (Netflix pays AWS 29 **million** dollars a *month* instead of going through the pain of dealing with scaling themselves.)

Apart from offering "Hot Storage" solutions, where HDDs are ready to serve your webpage at all times, AWS also offers this "Cold Storage" service called Glacier. Google's equivalent is Coldnine. They store tremendous volumes of data that don't really need to be accessed that frequently at ridiculously cheap rates. How? Magnetic tapes. They are still used (*who knew?*), because they don't randomly get corrupted, and have an extremely high storage density. The only disadvantage is that the read times are *terrible*.

These services will store your data on multiple tapes, check them frequently and ensure that all the data is fine and the tape can still be used in the future.

Glacier charges extra every time you make them go fetch their data for you. They are an archival solution, for data that is just getting stockpiled and doesn't need to be accessed in years or maybe even forever. Typically the data is stored for legal reasons.

4.5 Designing a View

General Design Procedure

- Understand your client's needs. (If it's your idea this step will not happen, of course.)

(but general advice: think of yourself as a one man company, and split the various roles/posts that need to be taken up. Write down a job description for each of these roles in two to three bullet points which you can look at the end of the year and ask "did this person do their job" and be able to say yes or no to that. This will drastically change the way you think about the things that need to be done and how you go about them.)

- User and Task Analysis – What are the user's preferences? What are the needs of the tasks we must accomplish?

- Prototyping – Confirm if the expected functionality, aesthetics, accessibility, needs are all at an acceptable level. Make wireframes for every page, and plan out the workflow.

- Doing Things.

- Testing – User acceptance, usability, accessibility

Usability Heuristics

Jakob Nielsen's 10 Heuristics. Mostly common sense, but once you begin trying to flex fancy design skills or cool effects, or get a bit stuck up in your head with obscure ideas, you might begin to make unintuitive and less usable websites.

There are a good set of guidelines to give you a sanity check.

1. Visibility of system status;
2. Match between system and the real world;
3. User control and freedom;
4. Consistency and standards;
5. Error prevention;
6. Recognition rather than recall;
7. Flexibility and efficiency of use;
8. Aesthetic and minimalist design;
9. Help users recognize, diagnose, and recover from errors;

10. Help and documentation

Accessibility

It is a core concept. Please don't ignore it if you can help it. I've seen some ridiculously cool pages and at that point, I think the target audience has to exclude some subset of people. But if you're making a general purpose application, then it's important that you keep it accessible to as many people as possible.

Regardless, I don't think I have much to add in terms of design guidelines for increasing accessibly, whatever I had written in the HTML section is about it.

1. Perceivable: Make the webpage perceivable with good html structure and appropriate tag usage. Provide captions for visual multimedia content and transcripts for audio content.

2. Operable: Make the webpage possible to use with just a keyboard. Give people enough time to read and consume what's on the page. Make an intuitive design and page content flow. Help users navigate and find content. Do not accidentally cause a seizure with flickering lights or something.

3. Understandable: Make text clear and readable. Provide language change support if possible. Help users avoid and correct mistakes.

4. Robust: Maximise compatibility. For instance, there is a lot of javascript syntax that came out in recent versions of JS, which isn't supported by all browsers.

Part 5: Testing, Deployment and Misc Content

5.1 Testing

I had the experience of making a full webpage as part of the MAD-1 project, and I cannot emphasise enough just how important this is.

"Tests" are a check of how the system responds to a particular type of input, and covering all the edge cases and core functionality in clear testable operations. I personally went through manual testing of each feature that I implemented.

So when you make a search bar, you first check if you are able to search by product name, then by category name, then check if the filters work, category by category, and then together.

It's sometimes a very helpful method of *coding* to think what are the tests that this needs to pass to be considered functional, and coding the backend to first meet those goals. This is called test driven programming, and has its strong proponents.

However, it testing manually isn't always ideal. I had by the end of my project checked my core functionality a ridiculous number of times, just to see if what I added didn't break anything. In the end, when I added the search bar, tested it and submitted the project in a hurry, I broke the entire admin side of my code, because I set up incorrect triggers on the database for the search table updates, which triggered on any admin operations, crashing the webpage. I had to wait a term and resubmit my project, and it really drove it into me that it's super important to have a checklist of tests and preferably, write down code that automatically does all this testing on every patch. In python, the pytest module can aid with this task.

In larger projects, this kind of automated testing becomes absolutely essential. I hope I have made a convincing case for why. In fact, this is an essential part of the DevOps Pipeline, the so called "CICD" Continuous integration and continuous deployment.

The continuous integration part of it is the automated testing and merging of the modified source code, often contributed to by several individuals (or teams). The tests of CI are optimised to be very fast, and thorough tests are run a few times a day. Once all the tests are passed, the CD pipeline automatically deploys the latest version of the code that has passed all tests, so people always access the absolutely latest version of your server.

The minor disadvantage in CD is that you may not catch all possible errors with the tests, and potentially break the page in some edge case that was newly created. Deploying new versions with vulnerabilities can prove dangerous.

5.2 Deployment - Containers

If you've ever set up prerequisites for complex applications, you may know that it is a royal pain.

First, you find out the correct versions of all the dependencies, and attempt to install them one by one. Next, one of these commands will throw a screen full of red, and you spend an hour on stackexchange figuring out how to get this thing to install. And you repeat this 5 more times.

Such merriment is exponentially greater when you do this on a server machine with already pre-existing dependencies which potentially clash with what you are now adding.

Not only do you need to go through the pain of installing all these prerequisites, you must give a set of often complex instructions, which may or may not work for everyone, and get your entire development team to follow all of it. And do it on the official servers as well.

And then one member will come and tell that they use their own custom version of Kali and your instructions are garbage. And they will spend 10 hours on stackexchange just setting up the app that everyone is supposed to be working on.

This joy, which I have experienced many fold over, is what containers seek to absolve this world of. Containers are OS level units, capable of executing by themselves. The idea is to use the same kernel as the main OS, but keep the container's process separate.

Not only is this highly portable and easy to work on, it's possible to version control the entire container, including the prerequisites. Container software has come a long way from its origins in chroot, now culminating in the incredibly popular Docker system.

However, it is often the case that using one single container for an entire project is a bad idea. Splitting the thing into many containers, for example, a Logger, a load balancer and a database manager, is very helpful.

Now that we have several containers running independent sandboxed processes, someone needs to communicate between them and execute things in a particular order. It must "orchestrate" the containers. The most popular tool to achieve this today, is Kubernetes.

Container tech prompted the shift from monolithic programming to containerised microservices that are each made for a particular job. This leads to often,

hundreds, if not thousands of containers, calling for a very clean and systematic management framework, which is what Kubernetes offers.

Actually using kubernetes is well beyond the scope of this course.

We have learnt the fundamentals: what makes up a webpage, the structural division in the MVC paradigm, how the internet works, and how we can run a functional web server by ourselves. The loose threads we are left with are: advanced frontend coding with fancier CSS and JS, optimising the server load and potentially switching to a better server framework, protecting against advanced cyber-attacks, and scaling to a large audience while being able to very quickly push updates via the DevOps pipeline, which in itself requires knowledge of many more technologies.

The road is long indeed. But it's fun.

In the sequel document, MAD-2, I'll tackle advanced JS and CSS, and potentially switch into NodeJS for the server side code.

Alright then. Farewell, weary wanderer. Until next time.