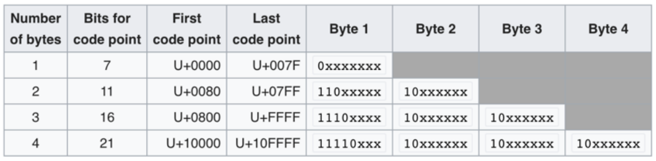
HTTP (most popular version: 1.1) *Acronyms*: TCP (transmission control protocol), MIME (multipurpose internet mail extensions)
Consists of 3 parts: 1) Request line / response line | 2) HTTP header (additional metadata that describes the request or response) | Empty line | 3) Actual body of request and response may follow
Request line: contains method, path, protocol number (GET /cs144/form.html HTTP/1.1)
HEAD method: the server must not return a message-body in the response, good for troubleshooting to make sure GET is working
Why do requests have:
'Host name' attribute: Virtual Hosting: multiple IPs can be hosted on the same machine, so when a request is sent to the server, there needs to be a way for the server to know which website is being asked for
'Connection Alive' attribute: We can say we want to keep the connection alive if we expect to make many more requests to the server so we save on TCP handshake overhead
'Upgrade Insecure Requests' attribute: Client may ask to support https if possible and upgrade the connection
'User Agent' attribute: Tells the server browser and OS details (client software info)
'Accept' attribute: Tells the server what types the client can support; "q" value shows the priority of the desired formats (higher number = higher priority) Accept-Encoding: gives compression detail preferences
HTTP/2.0: tries to break up the request/response paradigm as it is not efficient for large number of assets and uses multiplexed streams (because mobile connections fetch a lot of items)

MIME: HTTP explicitly declares type of data being exchanged ("type/subtype" in HTTP "Content-Type" of header)
Text encoding: ASCII (7bits for 128 char.), local/regional encoding (2 bytes for character; Asia, but there was *fragmentation*); UNICODE (unique number for every character; 21bits (too much and have to update both programs and data); UTF-8 (backwards compatible; maps all ASCII characters to same 1-byte number, allows easy recovery)



Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

HTML used for the structure of the document; CSS for formatting and style; HTML5 is the current standard.
Notable tags: `<iframe src="….html">` embeds anot. HTML page inside; <input type="file"> … enctype="multipart/form-data"
HTML5: removed a lot of formatting tags; `<tt>` (code text, changed to <code>); `<b> <i>` are still recognized but `<strong>` and`<em>` are recommended; added a new tag for audio `<audio src="voice.mp3"type="audio/mpeg"controls>`
HTML characters: `&nbsp;` is a whitespace HTML character; `&lt; &gt; &amp;`
`<div>` is a block element, `<span>` doesn't create a new block

CSS: cascading rule: id > class > tag; Examples: [target =" _blank "] {} : attribute target has value blank; div, p: multiple selectors, div p: descendent of div; >: direct child; +: adjacent sibling; ~: general sibling; margin: top right bottom left; *Positioning*: relative: relative to original position; absolute: relative to nearest positioned ancestor; fixed: positioned relative to "viewport"; static: default value
Block elements: <div>, <ul>, <p>, *inline*: <span>, <a>
Viewport: web pages were initially designed for desktop machines; on mobile, sites were rendered to have static design and fixed size, and then zoomed out; viewport meta tag can be used to render the web page to use its default device width
Media query: allows us to specify CSS rules we want to use for different devices (`@media (max-width: 800px) { p { font-size: 14px; }}`
Flexbox: dynamically set the dimensions of an HTML element; Flex container: flexible container (`display: flex`), all children are also modified Flex-grow: ratio to expand by when container's dimensions are changed; Flex-shrink: ratio to shrink by; Flex-basic: set default values, and resizes elements if the container's dimensions are changed
MVC: *transitioning to dynamic sites*; want to split application into model (handles data storage and access), view (result presentation) and controller (core application logic) (*JSP tag*: <body>Your data : <%= request.getAttribute("data1") %></ body >)
AJAX: uses engine for client-side processing to work in parallel to server-side processing (uses event-driven programming); necessitates JavaScript and parts of DOM model for monitoring events (tree-based representation of HTML)

Javascript: ==/!= true if same value (after type conversion); ===/!== to check same value and type; NaN, Infinity are number type; undefined (uninitialized variable) has type undefined; null (absence of object) has type object; object comparison is by reference
Mutators: reverse, sort, push, pop, shift, unshift, splice; *accessors*: concat, slice, filter, map
Scope: **let**: limited in scope to the block, statement, or expression on which it is used, unlike var: global, or local to an entire function regardless of block scope; functions use lexical scope
Interpretation of this: inside object's method, `this` points to object; => retain the `this` value of enclosing lexical context
JSON: both object property names and strings require double quotes

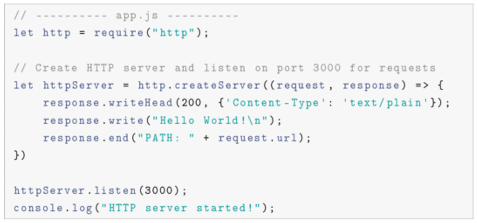
DOM: global object is "window" (window.location, window.history, window.alert(), window.confirm());
Updating properties: `document.body.style.background = "yellow";` `document.body.innerHTML = "<p>new text </p>";` `document.getElementById('warning1').style.color = "red";`
Event handling: event has type and target (a reference to object that dispatched the event);
Return type in handler is important! If false within an associated event handler, the remaining event handlers which would be called in sequence otherwise will not be called.
Event bubbling: When event occurs, its event handler is called -> Parents' handlers are also called -> Parents of parents' handlers events are also called -> … bubbles up until `event.stopPropagation()`
JavaScript execution timeline: single-threaded, have to run tasks async; document object is created -> document.readyState = "loading" -> scripts are executed sync (DOM loading is paused unless async script) -> document.readyState = "interactive" -> browsers fire "DOMContentLoaded" event -> document.readystate = "complete"

Traditional web stack: Middleware: Apache/Nginx, App logic: PHP/servlet, Data: MySQL (most code in server)
Modern: server transformed to be a back-end service providing data-persistence and transaction support; more client-side JS
Model: MongoDB, View: Angular, Controller: Node.js + Express

MongoDB: schema-less; primary operations: create, retrieve, update, delete; uses nested key-value pairs in a JSON-like format (but stores as BSON; can use Mongoose to ensure structure in data
Relational tables: in the database are designed independently of the application; *Normalization*: want to minimize redundancy; JSON sounds reasonable but still, splitting data into multiple tables is necessary to reduce redundancy of memory and updates

	Pros	Cons
MongoDB	Schema-free Useful for loosely coupled objectives	If joining is necessary to produce different views, will be much more complicated than for RDBMS
RDBMS	Efficient joins Useful for tightly coupled systems	Table is designed based on the intrinsic nature of the data, not for a particular application Need to "decompose" for storage and then "join" for retrieval

Syntax: `show dbs; use <dbName>; db.dropDatabase(); show collections; db.createCollection("books"); db.collName.drop(); db.books.insert({title: "MongoDB", likes: 100}); db.books.find({$and: [{likes:{$gte: 10}}, {likes:{$lte: 20}}]}); db.books.update({title: "MongoDB"}, {$set: { likes: 200 }}); {$unset: { likes: ""}} // removes 'likes' field; {$inc: { likes: 1}} // incrs first match; db.books.remove({title: "MongoDB"}) // all; db.books.createIndex({title:1, likes:-1}); multi or justOne to adjust parameters`
Node.js: (JavaScript runtime environment based on Chrome V8 JavaScript engine);
Based on CommonJS rather than ECMAScript2015 standard
Express: 1) provides URL-routing mechanism, 2) integrates HTML template engine (EJS), 3) integrates *middleware* for complex request handling
Routing support: `express.method(path, handler)`; req has query, params, body, cookies attributes; .next() jumps to the next handler (top-down)
More syntax: res.status(200), res.send(body), res.sendFile(abs_path_to_file), res.json(object), res.render('index', {title: "Hello"}), res.direct('/')
EJS: <% %>: Javascript for control flow; <%= %>: result of expression, <%- %>: no HTML escaping
Middleware integration: express.use([path-prefix], middleware) … [code examples to follow]


```
// ---------- app.js ----------
let http = require("http");

// Create HTTP server and listen on port 3000 for requests
let httpServer = http.createServer((request , response) => {
    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.write("Hello World!\n");
    response.end("PATH: " + request.url);
})

httpServer.listen(3000);
console.log("HTTP server started!");
```

Async: Callback functions cause callback hell -> nested callbacks using anonymous functions -> promises -> async/await
Promise: represents the "guarantee" of eventual completion or failure of an async operation (promise.then(resClbck, rejClbck);)
Async/await: await can be used in front of any function returning promise; async returns promise
Session: *Cookies*: `Set-Cookie: username=john; path=/; domain=ucla.edu; expires=…` (where the cookie should be sent); has same-origin policy (can use third-party cookies to identify user across multiple domains); Use secure attribute and signed cookie to prevent cookie theft and poisoning; *JWT* format: header.payload.signature, header: { "alg": "HS256", typ }, signature: secret-key encrypted hash
Session management: server obtains session related states from "session data store" using session ID;
Storage: localStorage persists over multiple browser sessions; sessionStorage persists only within tab
History: pre-HTML5: URL fragment identifier (change in fragment identifier does not reload a page); HTML5: history.pushState(object, title, url), history.replaceState(…)

Asynchronous HTTP request: HTTP requests can be sent only to same-host (use proxy ".php", CORS, or JSONP)
CORS (Cross-Origin Resource Sharing): browser can inquire server-approved cross-request domains through Origin: in header
The server replies the list of allowed domains with Access-Control-Allow-Origin: in header
JSONP: set src to URL which contains a JSON wrapped in function, which can query an external third-party site
Using XHR: `xmlHttp = new XMLHttpRequest();`
```
xmlHttp.open("GET", URL); // method , url; setRequestHeader(header-name, value) adds request header
xmlHttp.send(null); // body of the request; onreadystatechange is event handler
response = JSON.parse(xmlHttp.responseText);
```

TypeScript: superset of Javascript; adds types, explicit declaration of class properties, interfaces, generic functions/classes using parametrized types (`let p = new Pair<number>(1 , 2);`), decorators which are functions that modify JavaScript classes, properties, methods, and parameters (@sealed, @Input)
Component-directive: custom-defined tag representing component (<app-search-box></app-search-box>)
Attribute directive: change the appearance or behavior of an element, component, or another directive
Structural directive: shape or reshape the DOM's *structure* by manipulating elements (*ngIf, *ngFor, *ngSwitch)
```
<div [ngSwitch]="hero?.emotion"> <app-happy-hero *ngSwitchCase="'happy'" [hero]="hero"></app-happy-hero></div>
```
4 techniques for data binding: *Interpolation*: bind data to UI element {{ expression }};
Attribute directive: property binding, event binding, two-way binding

- *Property binding*: <app-search-box [appName]="title"> </app-display> (bind DOM element to UI element)
- *Event binding*: <input type="submit" (click)="showAlert()">; can reference $event, $event.target.value
- *Two-way binding*: [(ngModel)]="property" (requires import FormsModule)

Parent-child inter-component communication

A. Search box component can throw an event whenever obtains a resource from Google server
```
import { EventEmitter , Output } from '@angular / core'; ...
@Output () suggestion = new EventEmitter < string [] >() ; ...
this . suggestion . emit ( result ); // at the end of prcoessSuggestion ()
```
B. Parent (the root component) catches the event, and sends the event to the display child component through property binding

`<app-search-box (suggestion)="display.suggestions=$event"> </app-search-box>`
`<app-display #display></app-display>` (uses a *template reference variable*)
<u>Service</u>: works as a communication channel between components
We create a service that provides this general service to whoever needs this kind of service
Uses dependency injection because we don't want to have to instantiate the same server again for every component that uses it. When the application starts, the AppModule creates an instance of the service class and passes the created instance as a constructor parameter.
```
constructor (private suggestionService : SuggestionService) { suggestionService.subscribe ((suggestions) => {
this.suggestions = suggestions; }); }
```

<u>Caching</u>: Between transport and HTTP, we can use CDNs to cache pages closer to users. Between HTTP and app layer, we can re-serve static parts of reasonably static pages, mimicking a static site or have pre-generated results. For persistence layer, we can add a caching layer (with Memcached, Redis)
<u>Scaling up vs scaling out</u>: up: price increases exponentially but don't have to rewrite code; upper limit to how big a machine can be; out: parallelism, individual machine are cheap (extremely scalable), but almost never get linear scaling
How do we **<u>scale out</u>**?
1. We may need a *load balancer* that distributes work evenly between machines: A) Not practical because there may be changes in one machine and different changes in another B) Have to manage a lot of synchronization
2. Have a dedicated machine for each HTTP layer: A) If a layer needs more computation, we can add machines for that layer and use a load-balancer since synchronization is not necessary B) Encryption layer, and HTTP layers are very easy to scale out C) If application layer maintains its own state, then it is hard to scale out; Otherwise, if it purely depends on the persistence layer, then it is easy to scale out; C) *<u>Persistence layer</u>* may possibly be hard to scale out:
<u>Read-only load</u> (e.g. Google Maps): have copies of the data to scale out persistence layer
<u>Localized read-write load</u> (e.g. banking, Gmail): see a clear a locality in the data since we're only looking at one user's data; can partition the data (sharding) based on their access pattern
<u>Global read-write load</u> (Amazon, eBay): No longer linear scaling (the more machines we have, the more the write IOs saturate the machines) (~ in example, saturates to around 30 sessions per second) … persistence layer is the main bottleneck rather than CPU

$$30x\ (x = number\ of\ machines) \geq \left(\left(2\ read\ \frac{IOs}{s}\right) + \left(1 * x\ write\ \frac{IOs}{s}\right)\right) * n\ (number\ of\ requests)$$

<u>Approaches to scaling out persistence layer</u>: rather than single drive, we want to use multiple machines to have a distributed file system that simulates a single hierarchical directory structure (i.e. Hadoop file system (HOFS))
<u>Hadoop</u>: uses master server and chunk servers; master server has a mapping of file name to location at a specific chunk; different chunk servers hold different chunks of the data; there may be a lot of chunk servers so data may go down; in order to avoid losing data, instead of storing one copy we may need replication; master server now needs to monitor which chunk servers are available and clean
<u>Examining scalability</u>: 1 billion files, 64 byte filename, 4 GB / 64 MB = 64 chunks
1 billion files * 64 chunks * 8 (size of individual chunk locations) = 512 GB (size of the chunk)
1 billion files * 64 byte filenames = 64 GB; 512 + 64 = Around 576 GB
<u>Scaling out the actual database</u>: People tried to see if could scale out MySQL databases but there was Byzantine general problem
Many distributed systems try to achieve ACID transactions or at least most of its guarantees; new theorem: CAP (2/3 possible max)
<u>Consistency</u>: after an update, all readers will see the latest update; *<u>Availability</u>*: continuous operation despite node failure; *<u>Partition tolerance</u>*: continuous operation despite network partition (mainly consistency guarantee is relaxed; will have delayed consistency)
<u>NoSQL</u>: First generation: given the key, store this value (Redis, Memcache, Amazon SimpleDB); Second generation: column-stores: rowkeys map to different columns (Google BigTable, Cassandra); third gen: document-stores: attribute name, key value (MongoDB, Amazon DynamoDB, CouchDB)
<u>Consistent hashing</u>: given a key, how do we distribute the values over multiple machines?
Apply a hash? hash(key) mod 3 with three machines but what if we're adding more machines? Might need a lot of internal shuffling
Consistent hashing: Apply a hash function on the key to generate a number between 0 and 1; Hash both the machine and the key we will store in the machine; The key will be stored in the machine whose hashed number is closest in succession to the key's hash
Non-uniformity of data distribution: Fix: virtual node mapping: split nodes (representing machines) and add the split node parts into different locations in the circle
<u>Guaranteeing atomicity</u>: two-phase commit (ask everyone if ready before commit); asynchronous transaction (no coordinated wait and synchronous commit, every participant "commits" whenever possible and moves ahead, downside is rollbacks)

Cluster-based computing: power and heat issues causing hardware failures
MapReduce/Hadoop provides: 1) automatic computational task distribution, 2) failure handling (monitors health of nodes, and reassigns tasks), 3) speed disparity handling (reassigns task if node is too slow)
Computational job on thousands of machines in parallel: 1) map/transform step 2) reduce/aggregate step

---

Four properties of HTTPS Internet security (for access and guarantees): 1) *<u>Confidentiality</u>*: making sure others cannot understand messages; 2) **<u>Authorization</u>**: want to make sure only authorized users can access and modify data; 3) *<u>Authentication</u>*: making sure other parties are who they say they are; 4) *<u>Message integrity</u>*: no tampering with messages
<u>Confidentiality</u>: encryption; requires a cipher (encryption + decryption function) using either symmetric or asymmetric key
<u>Symmetric key</u>: using the same secret key, encrypt and decrypt; requires encrypter and decrypter to have synchronized the key securely; also necessitates n^2 keys for every pair of parties for n parties
<u>Asymmetric key</u>: using different keys for encryption and decryption; If I want a secret message from anyone, I send my public secret key to the other party, and other parties send it to me using this public key, and I decrypt using my secret key
Factor of 1000 more expensive in terms of computation than symmetric key; so we can *use asymmetric key just to encapsulate the symmetric key and we use the symmetric key for the actual decipher*
<u>Authorization</u> (how does Amazon make sure that we are authorized users?): username and password

- Hidden assumption: only the user knows his/her username and password

Authentication (how do we make sure Amazon is Amazon?): challenge/response system
We generate a random number and send it using their public key. They can decrypt the number using their private key and then send it back.
How do I know public key of Amazon? Ask for a certificate from a *certificate authority* that I trust
I have to now verify that the certificate is from an authority that I trust… after certificate verification, we can use challenge/response
Message integrity (how do we ensure that the other party can tell that it is really the message I'm sending?): electronic signature
After the body, add a specific sequence of bytes that only I can generate (Requirement: seq. of bytes should be unique to that particular message)
1) Hash the document and attach that to the end of the body 2) Encrypt the entire message using public secret key 3) Once anyone receives the message, they want to verify if the signature is intact. They decrypt the message using the private secret key and then verify the hash.

Buffer overflow: user carefully constructs input string to overwrite return address to wherever they want in the stack and takes over control flow of the program; *Protection*: Use modern language, Stackguard (option in compiler to detect most types of buffer overflow in program): adds canary (random number) between local variables and return value. If canary is not the same value before we transition into a new function, there has been buffer overflow; Important for the canary number to be random. Otherwise, attacker can figure out what it is; ALSO *Never trust user input*
SQL/command injection: use prepared statements and never trust user input
Client-state manipulation: If server has to maintain state, it has to be synchronized over interactions with multiple clients
Problem with maintaining state on the client: how do we know the state the client sends is the state the server sent to the client? (a message integrity problem); Clients can manipulate state to trick the server
To avoid stolen session id attack, pick random session ID from large pool and make it short-lived. For cookies, make sure they are signed.
Cross-site scripting (XSS): B injects script that steal's each visitor of A's cookies. Every time C visits A's site, cookies are sent to B
Protection: either don't allow HTML tags for input (unpractical), or use whitelist, blacklist, and input validation, and output sanitization; be very careful about third-party libraries (library will have complete access to your data and you don't know what they do)
Cross-site request forgery (XSRF): I (browser) authenticate with banking website (victim) and banking site sends me a cookie. As long as I contain valid session ID, I don't need to authenticate anymore. I then forget to logout and start visiting other sites. I access a hacker site (which doesn't have access to cookie). Form in hacker site can't see the cookie but can still use it (may return a form sent when we access victim.com)
`<form action="victim.com">  <input type=hidden value="1000"> </form action="submit">  </form>`
Protection: Always make sure to logout after a session. If a request is not genuinely coming from the user, we should be able to detect it. Same-origin policy helps. Action token: whenever a particular form field is downloaded from website, it should contain some string that only the website can generate. The hacker's sent form field can not replicate that string.

New startups are no longer using the approach of buying your own domain+server+internet connection pipe. Cloud computing is becoming more popular to install apps, and the company makes sure the machines are running fine (rent VMs, rent computing power)

- Popular cloud computing websites: AWS, DigitalOcean, Azure, Google Compute Platform

Three different ways to use cloud computing sites:

1. Infrastructure as a service (IAAS): Physical machines supplied as VMs or containers, we are still responsible for maintaining the software layer we place on top
2. Platform as a service (PAAS): Providing the core API that is used by any kind of application (provide a particular service s.t. "this is the database, "I guarantee that you can store as much as you want up to this limit with a guarantee of this delivery time")
    - Basic building blocks of service so that we can build our own applications
3. Software as a service (SAAS): user doesn't have to do anything about maintaining software, user just uses software supplied by cloud computing site (e.g. Office 365, Gmail)

// URL routing

```
// ---------- app.js ----------
let http = require("http");

// Create HTTP server and listen on port 3000 for requests
let httpServer = http.createServer((request, response) => {
    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.write("Hello World!\n");
    response.end("PATH: " + request.url);
})

httpServer.listen(3000);
console.log("HTTP server started!");
```

```
// routes/index.js
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```

// middleware integration

```
// ----  app.js ------
let express = require('express');
let app = express();

app.set('view engine', 'ejs');
app.set('views', '.')
app.get('/', (req, res, next) => {
    res.render('index',
      { title: "Hello",
        posts: [{title: "Title 1"}, {title: "Title 2"}]
      }
    );
});
app.listen(3000);
```