MAD -1

CLASSTIME | Page No.
Date / /

Week-1

L1.1 What is an App?

- App → it is comp. software, most commonly for mobile devices. App stores have emerged to sell mobile apps to phone users.
- terminal / command prompt → to run program on OS
- Desktop → standalone (mail, web browsers), work offline (local data storage), SDK → software dev. kits (custom frameworks)
- Mobile → have constraints (ltd. screen space, user interac^n, memory, power), usually network oriented
- Web apps → the platform, works across OS: devices create a common base, heavily network dependant.

L1.2 Components of an App

- storage, computa^n, presenta^n → show data to the user
  - 'data corresponding
  - how are we manipulating data

Email eg.

where are they stored? how stored?	indexing of emails searching	display list, display of indi. mails

- platforms - desktop, mobile, web-based, embedded

L1.3 Client servers & peer-to-peer architectures

- architectures → client server
  → distributed peer-to-peer
- server → store data, provide data on demand,

may perform computa~~n~~.
- clients – end-users, eg. data, user interac^n, display
- network – connects client to server, can be local, data pipe – no alterations (data flows both ways)

Client ⟷ Network ⟷ Server

- Client server model
① explicit differentia^n btw. the 2
② local systems – local network
③ machine clients – antivirus software
④ variants – multiple servers
⑤ Email, DB, whatsapp, messaging, web browsing
- Distributed (peer-to-peer) model
① all peers have equivalency
② error-tolerance
③ shared inform^n
④ Bittorrent, blockchain-based sys., IPFS, Tahoe (distributed file systems)

1.4 Software Architectures
- MVC, MVA, MVP, Etc.
- design patterns – a gen, reusable sol^n to a commonly occurring problem within a given context in software design
- some patterns made by experts can make design & dev. better, guide me.
- model – store emails on server, index, ready to manipulate

- view — display list of emails, read individual emails
- controller — sort, delete, archive
- M-V-C — paradigm
  - model - core data stored, DBs, indexing, manipulation
  - view - user-facing side, interfaces for finding info
  - controller - business logic, manipulate data
  - handy now: origins in small talk lang. 1979
- User uses controller to manipulate model that updates view that user sees.
- MVA, MVP, MVV, Hierarchial MVC → each has its uses but fundamentals are very similar
- how to build apps that are web-based with central servers use hypertext markup to control and manipulate display

L1.5 Intro to the web

- web is generic — cross platform OS
  → built on sound underlying principles
  → wealth understanding - constraints, costs
- telephone networks (1890+) - circuit switched network, physical wires
- packet switched network (1960s) - data instead of voice, more efficient
- ARPANet (1969) - node to node network
- others IBM SNA, Xerox ethernet
- protocol - format packets, place on wire, each network had its own protocol

- 'inter' network - communicate btwn. diff. network protocols
- IP Protocol (1983) - can be carried over diff. underlying networks: Ethernet, DEC net, PPP
- TCP (Transmission control) (1983) - error control, automate scales and adjust to network limits
- domain names (1985) - replaced IP address, .com resolution
- hyper text (1989+) - text documents to be served, Tim, HyperText ⇒ The www.
- original web ltd - static pgs., ltd. styling, browser compatibility issues
- web 2.0 (2004) - dynamic pgs., http as a transport mechanism, platform agnostic OS, client side computa" & rendering

L1.c How does the Web work?

- web server - old comp. + network connec"
  → software - listen for incoming network connections on a fixed port, respond in specific ways
  → Protocol - what should client ask server, how should server respond to client
- HT - regular text doc., contains 'codes' - how to 'link' with other documents
- http - largely text based, client send requ" server responds with ht doc.

CLASSTIME | Page No.
Date / /

L1.7 Simple Web Server

- simplest server: `while True; do echo -e "HTTP/..." | nc -l localhost 1500; done`
- general web server — listen on a fixed port, on incoming request, run some code and return a result → std. headers should be sent as part of the result, MIME
- typical req:
```
GET / HTTP/1.1
Host: localhost:1500
Accept: */*
```
- viewing responses with curl
  - → curl, wget — simple command line utility
  - → can perform full http requests
  - → verbose output includes headers
  - → very useful for debugging

L1.8 What is Protocol?

- both sides agree on how to talk
- server expects 'requests' — nature of request, nature of client, type of results
- client expects 'responses'
- http — text based, specified as 'GET', 'POST', 'PUT', headers convey info.
- response headers — convey req type, 404 not found, cache info.
- GET — simple req., search queries
- POST: — more complex form data, large text blocks, file uploads
- PUT/DELETE — used in Web 1.0, basis of most APIs (REST, cURL)
- python has http.server

→ serve files from local folder
→ understands basic http requests
→ give more detailed headers and respons
- get /
  ✱ → debugging
- get /server.sh ⇒ the content-type changes → the application the data as required.
- both GET and POST method is used by http protocol to transfer data
- in GET, data is visible in URL but for submitting forms we use POST method

L1.9 Performance of a website
- latency - time to take a request to get response
- data center from Chennai to Delhi takes 1 round trip of 20 ms ⇒ only 50 requests/second
- response size = 1 kb of text (headers, html, css, js)
  with 10 Mbps net ⇒ ~ 10,000 requests/sec
- google response - headers (100 B), content (144 kB, approx. 60,000 requests/sec.
  ↳ ~ 80 Gbps bandwidth
- memory - YT
  → one python http server process ~ 6 MB
  → multiple parallel requests : multiple pro
- storage - 100s of billions web page, 100 petabytes (index size estimate), storage?, retrieval
- Scaling requires careful design

CLASSTIME | Page No.
Date / /

T 1.1 App dev using Replit

- replit.com
- newer repl → Python → App Dev t
  Layout | Work space | output
  plain
- print("hello world!") → Run
- add file ⇒ index.html

+ in main.py -
  import http.server
  httpd = http.server.HTTPServer(("", 800
  http.server.SimpleHTTPRequestHandler)
  httpd.serve_forever()

SC 1.1 How to serve HTML files on LAN?

- create index.html using terminal
  <!DOCTYPE html> ⇒ write some basic html
+ meta charset (UTF-8) ⇒ indian char
- on google browser ⇒ inspect element
- python 3 -m http.server
- ip address / port ⇒ how to see on LAN

Week 2

L2.1 Info. representation in a machine

- markup
- info. presentation
  - comp. works with bits (0 & 1) 6 (0110)
    - place value: binary numbers
    - 2's complement: -ve nos -6 (1010)
- ASCII, Unicode, UTF-8
- info. interchange - communicate betw. machines, machines only work with bits, standard encoding
- 01000001 TH → string of bits / A / 65 decim
  - matter of interpretation and context
- ASCII - american std. code for info. interchange
  - 7 bits: 128 diff. entities a...z A...Z (0...9, spc. charac,
- unicode - allows codes for more scripts, characters
  - UCS → universal character set
  - UCS-2: 2b/charac. - max 65,536
  - UCS-4: 4b/charac. → 4 billion +

L2.2 Efficiency of encoding

- most common lang. on web → Eng.

Eg. text of 1000 words → 5000 charac. →

UCS-4 → 32b x 5000 = 160,000 bits

ASCII → 8b x 5000 = 40,000 bits

ASCII → 7b x 5000 = 35,000 bits

Huffman or similar encoding ~ 10,000 - 20,000 bits

→ zip
- it depends on text, use 1 byte for most common alphabets, group others acc. to freq., have 'prefix' codes to indicate

1st byte	2nd	3rd	4th		
0xxxxxxx				7	007F hex (127)
110	10			(5+6)=11	07FF hex (2047)
1110	10	10		4+6+6=16	FFFF hex (65535)
11110	10	10	10	3+6+6+6=21	10FFFF hex (1,114,111)

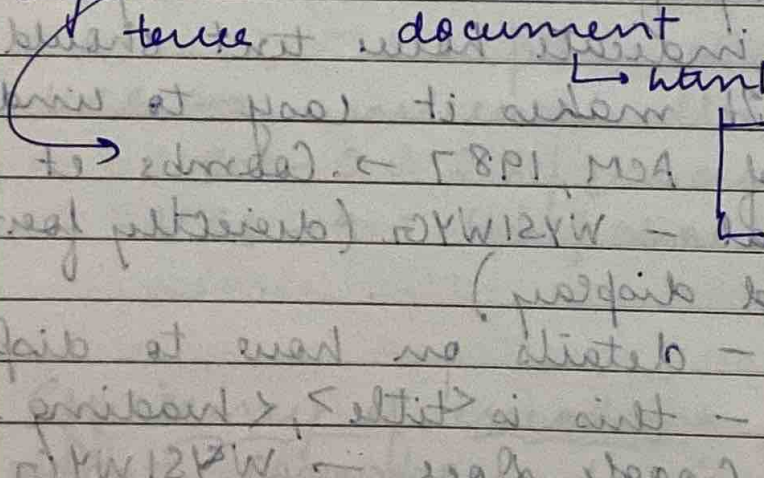
- UTF-8
↳ ASCII subset, more difficult for text processor, most common encoding in use today, it is a good practice to specify that your web uses UTF-8

2.3 What is markup?
- it is a way of codes in the regular flow of text to indicate how text should be displayed. It makes it easy to understand.
- commun" of ACM, 1987 → Coombs et al.
- presentational - WYSIWYG (directly format output and display)
- procedural - details on how to display
- descriptive - this is <title>, <heading>
- MS Word, Google Docs → WYSIWYG
  LaTeX, HTML → focus on meaning
- Content vs Presenta"
- Semantics → meaning of the text, struc./logic of the document

CLASSTIME | Page No.
Date / /

4.2.4 Intro to HTML5

- HTML → Tim Berners Lee (1989), an applicⁿ of SGML
- it meant for 'browser interpretaⁿ', it makes best efforts to display
- Tags → paired, angle brackets, closing tag, nesting, case-insensitive, locaⁿ-specific
- nesting ~
- or <b> which is correct?
  → try to use as it emphasizes on that
- SGML (1989 - 1997) → XML (1997 - 2010s) → HTML5 (first release 2008 - now)
- HTML5 → block (div), inline (span), logical (nav, footer), media (audio, video), remove presentational only tags (center, font)
- document object model → parse the tags



- DOM → direct manipulaⁿ of tree is possible, APIs → (canvas, offline), JS primary means of manipulating, CSS used for styling

L2.5 Intro. to styling

- >hello< ⟶ many possibilities
- style hints in seperate blocks ⇒ Themes
- style sheets – specify presenta" infoma"
- CSS – allow multiple definitions, latest takes precendence.

L2.6 Types of CSS styling & responsive websites

- inline css <h1 style
- int. css <style> in head
- ext. css ⟶ extract common content for reuse, multiple css files can be loade
- responsive design ⟶ adapt to screen (respond), CSS (styling), HTML (content
- bootstrap ⟶ commonly used frameworks, standard styles for various components, mobile first: highly responsive layout
- JS – interpreted lang., it is not that part of the core presenta" requirements

SC 2.1 Intro. to inline CSS

- many times you have to do ⇒ prone to errors

SC 2.2 Intro. to int. and ext. CSS

- <style> element in head tag
- save it as style.css in same folder in html file

<link rel = "stylesheet" href = "style-css">

SC23 Intro to css selectors

- tag-based we have already seen
  p { } → style
  .name_para { } → style for class name] <p class = 'name_para'>
  #personal { } → style for id name
- form input { }] descendant inputs of form
- form > input { }] the first child input of form
- form > input:nth-child { }] pseudo-class selector
- form p::first-letter { }] pseudo-element selector
- input[x='y'] { }] based on value attribute thing

SC24 Intro to Bootstrap

- getbootstrap.com
- <meta name = 'viewport' ...> to use responsive design

CLASSTIME | Page No.
Date / /

- Layout → graphs, tables, gutters
- content → headings
- components → directly use them w/o any coding

Week-3

L3.1 Overview of MVC

- model - state, index, ready to manipulate
- view - display, reading
- controller - sort, delete, archive
- origin - Smalltalk-80 ; seperation of responsibilities - abstraction ; roots in obj. oriented GUI dev.
- design pattern for diff. software patterns
  - model - app. obj.
  - view - screen representation
  - controller - how user interface reacts

Eg. Student gradebook
- Input data for model (courses, students)
- Outputs for Views (marks of 1 student)
- modifications for controllers (add, modify)

- interaction of user with the web app. is done by the view part

L3.2 Views

- user interface → screen, audio, vibrations (haptic), motion (door)
- user interac^{n} → keyboard, Touchscreen, spoken voice, custom button
  - determined by hardware constraints, diff. target devices possible, user-agent info. useful to identify context. (may not be under designer control)
- fully static, partly dynamic, mostly dynamic
- output - html (direct rendering), dynamic images, JSON/XML (machine readable)

- UI is any representation useful to another entity

L3.3 User Interface Design
- design for interaction with users
- goals — simple and efficient, aesthetics, accessibility (usefulness)
- Systematic process
  1. functionality req. gathering
  2. user and task analysis
  3. prototyping → mockups
  4. testing - user acceptance

L3.4 Usability heuristics
- Jacob Neilsen's heuristics for design → not specific to web apps, very useful and relevant
  1. visibility of system status
  2. match btw. system and the real world
  3. user control and freedom
  4. consistency and stds.
  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
- gen. principles
  1. consistency
  2. simple and minimal steps
  3. simple lang.
  4. aesthetically pleasing

L3.

L35 Tools - Part 1

- wireframe - visual guide to represent structure of web page, info. design, navigation, user-interface design
- Lucid chart.com → wireframe (annotations)

L36 Tools - Part 2

- PyHTML → Prog. html generation
- composable functions - each function generates a specific output
- import pyhtml as h

```
t = h.html (
    h.head (
    h.title ('Test Page')
    ),
    h.body (
    h.h1 ('This is title')
    h.div ('This is some text')
    h.div (h.h2('inside title'),
        h.p ('paragraph'))
    )
)
print (t.render())
```

gen. of data with valid html

- def f-table (ctx):
  return (tr(
  td(cell) for cell in row
) for row in ctx ['table'])
- templates - std. template - text, readability, basic programmability → Jinja 2 by Flask

- from string import Template
t = Template ('$name is the $job of $company
:
- Jinja (with flask), template functionality with detailed API
from jinja2 import Template
t = Template ("Hello {{ sth }}!")
print (t.render (sth = "World"))
- Jinja templates can generate any output, not just html.

L3.7 Accessibility

- various forms of disability or impairment
- W3.org — accessibility guidelines
- Std. — interplay btw. many comp. of a page
  → web-content - html, images, scripts
  → user-agents - browsers, assistive devices
  → authoring tools - text editors, compilers
- Perceivable - text alt. for non-texts, caption and alt. for multimedia, presenta" in diff. ways, easier for see and hear content
- Operable - keyboards, enough time, navigation and find content, inputs other than keyboa
- Understandable - lang., readable, help in avoid and correct mistakes
- Robust - compatible
- Techniques - Eg. descriptive labels for user interface controls
- Aesthetics - visual app., very impt, simplifica", can vary with time

CLASSTIME | Page No.
Date / /

T3.1 Cmd line arguements in Python

- new eupie → Python → layout, work, output
- import hello ⇒ in the main.py
exec (open ('hello.py'). read())
- Shell
-/Fist$ python hello.py
- import sys
print ("Total parameters", len(sys.argv))
sys.argv[:]

$ python hello.py 123 abhi

print (type (sys.argv[:]))

Week-4

L4.1 Persistent Storage

Eg. which students are registered for which courses

- spreadsheets → arbitrary data org. into rows and columns, op^ns defined on cells or ranges, multiple inter-linked sheets, any kind of tabular data - expressed in tables
- relationships — seperate full entry (redundant), other table 'joining' → rel^n specified with keys
- how to store the data?, can it be persistent?, how to represent relations?, structured way to represent, manipulate data.

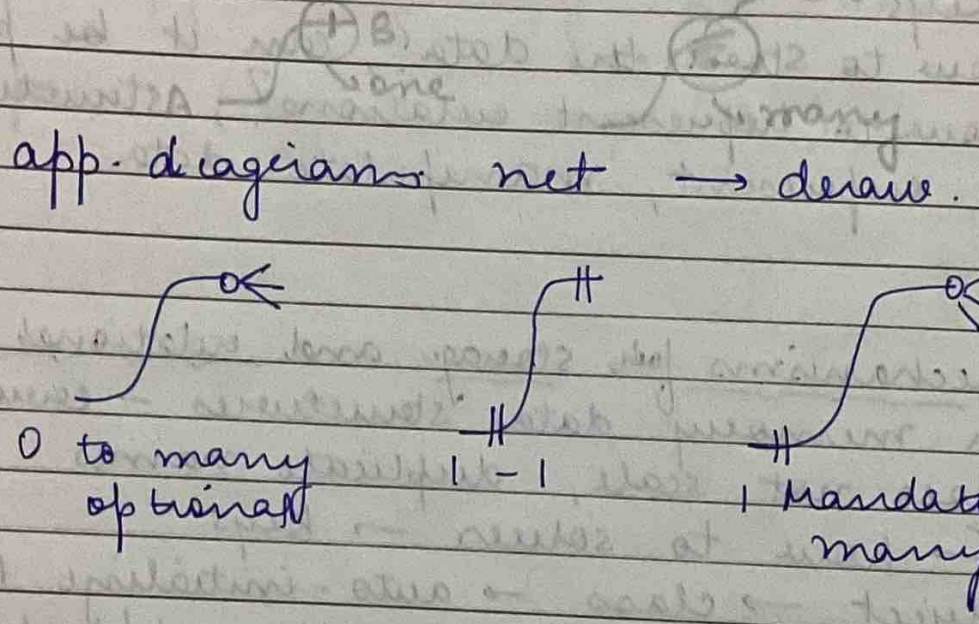
L4.2 Mechanisms for storage and relational DBs

- in memory data structures — error prone, does not scale, duplicate names
  ↳ how to solves → keys
- object → class → auto-initialize ID to ensure unique, func^ns to set/get values, add a new field to object easily
- these store. lost when process is shut down
- CSVs, TSVs, Python Pickles, etc ; essentially same as spreadsheets : limited flexibility
- spreadsheet → natural represent^n, extension, adding is easy, separate sheet for relation
  ↳ Problems → lookups and cross-referencing, stored procedures, atomic operations
- relational DB — SQL (IBM, 1970s), data stored in Tabular format
- unstruc. DB → NoSQL (mongo DB, couch DB)
  ↳ flexible, but potential loss of 'realidy'

CLASSTIME | Page No.
Date / /

L4.3 Relⁿˢ and ER diag.

- spreadsheet → 3 sheets
- 1-1 → 1 roll no. ↔ 1 student
- 1 - many → 1 student stays in 1 hostel but 1 hostel has many students
- many - many → 1 student takes many courses and 1 course has several students
- diag → ER, UML, Class relⁿˢ, etc.
  - ↳ Crow's feet notation (ER)



- app.diagrams.net → draw.io

L4.4 SQL

- key: unique way of accessing a given row
- 1° key - impt. for fast access in DB
- FK - connect to diff. tables
- Queries - retrieve data from the DB
- specific mathematical opⁿˢ - inner, outer join
- Inner join eg. —

```
select students.name, hostels.name
from students
inner join hostels
on students.hostelsID = hostels.ID
```

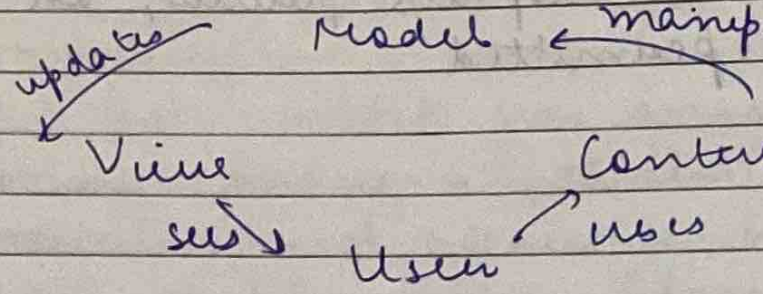
- Cartesian prod. → M × N combinations; powerful SQL queries can be constructed
- no details on displays, views, or what kind of updates permitted

CLASSTIME | Page No.
Date / /

Week-5

L5.1 MVC Origins

- collec^n of design patterns



- view acts as a presenta^n filter
- controller links the user and the sys.
- view shouldn't know about the user inputs
- general concept - ac^n → communicates the model and extracts the view
- originally for GUI Applicat^{ns}, sepera^n of concerns, state of interac^n maintained as part of overall sys. memory
- today → server & client states are diff. client is pure front-end to user, many variants of MVC
- MVC is good conceptual frameworks but breaks down if applied too rigidly
- the web in general doesn't have close knit structures of GUI Applica^{ns}

L5.2 Request and responses

- dynamic web pages → link triggers different behaviours
- Eg. NPTEL page
- client make requests, server send responses

- basic request: URL/link → HTTP GET
- more complex": form submissions → HTTP POST

L5.3 CRUD

- type of operations — Create → new entry, it must not exist, avoid conflicts, mention mandatory or not
- Read → get a list, summarize, plots
- Update → changes, updates
- Delete → remove, delete, unenrolls
- in context of DB ops, reflects cycle of data models, DBs are optimized for various combinations of ops.
  - Read heavy — lots of reading
  - Write " — security archive logs
- API — app. prog. interface
  - → std. way to communicate with server
  - → CRUD is good set of functionality for a basic API
  - → deals only with the data model life cycle
  - → client only needs to know API — not how server implements the DB.

L5.4 Grp. actions by controller

- CRUD etc. are a set of acts. There are other actions also.
- Laravel PHP Framework → resource controller
- acts → interac" btw. view and model
- controller — grp. acts together logically
- API → complex set of capabilities of server

- interaction thru HTTP requests
- HTTP verbs used to convey meanings
- Rules of Thumb –

① Possible to Δ views w/o model knowing
② Possible to Δ storage " views knowing
③ controller should never talk to DB directly
④ views and controllers tend to be more closely interlinked than with models

L5.5 Routes and controllers

- client – server model, stateless – server doesn't know the present state, requests sent through HTTP protocol
  - use verbs to convey meaning
  - use URL (uniform resource locator) struc. to convey context
- Routing: mapping URLs to actions
- Python decorators → add extra functionality on top of a func", "@" – decorator before func" name, effectively func" of a func"

request : GET /
response : invoke home() and return result

- Flask is MVC, more of a framework
- MVC is a way to achieve clean design

CLASSTIME | Page No.
Date / /

Week-6

L6.1 API Design

- distributed software architecture, server-clients, std. protocols needed for communicaⁿ
- the web → client-server are fab apart, authenticaⁿ - not core part of protocol, diff. network with diff. quality
- Roy Fulding → Representational State Transfer (REST), provide guidelines for software architecture styles
- Constraint ① → Client-Server → stores, process data, may perform ops → end users
- Constraint ② → Stateless → they both doesn't know the state of other being
- Constraint ③ → Layered system → Pool of backends, auth. server, load balancer - proxy frontend, network
- Constraint ④ → Cachability → response directly from proxy cached
- Constraint ⑤ → uniform interface → c-s must interact in uniform & predictable manner, 'resources' are exposed
- Constraint ⑥ → code on demand → Server can extend client functionality ⟹ these are not hard rules

L6.2 REST

- representational state-transfer = state info. btw. client and server explicitly transferred with every communicaⁿ

- seq. - client accesses a resource identifier from server → resource op^{n} as a part of access → server responds with new resource identifier
(state of interac^{n} transferred back-forth)
- http → possible to convey rest msgs; we use verbs to indicate ac^{ns}
GET → retrieves representa^{n}
POST → enclose data in request
PUT → create a target resource with data enclosed
DEL → delete the resource
- Idempotent op^{ns} → repeated app^{n} of the op^{n} is not a problem
↳ GET is always safe, PUT partly error prone, DELETE is also once allowed, POST may not be so
- CRUD - database op^{ns}, a common set of op^{ns} needed for web app^{ns}
- REST != CRUD, they work well together
- data encoding →
(i) basic HTML - for simple resources
(ii) XML - struc. data resource
(iii) JSON - simple form of structured data
- JSON → Js obj. nota^{n}, nested aerays, readily complex data struc. like dictionaries, aerays
- API data transfer format | diff. form int. switch and final resp^{n}
Input : text - HTTP
Output : JSON, XML, YAML
- YAML - documenta^{n} and configuration

CLASSTIME | Page No.
Date / /

L6.3 Rest APIs – Eg. 1
- CRUD, variants of listing, specialized functions, formal specifica^n help others to see.
- Eg. Wikipedia → open API, search for pages, history of page, JSON output
- MediaWiki is parent of Wikipedia
- id, key, title, excerpt, description
  ↳ part of documentation
- parameters → q (search) and limit → no. of page
- 200, 400, 500 → search error
  success ↳ q not set or invalid limit eg.

L6.4 Rest APIs – Eg. 2
- CoWin → vaccine registra^n and info.
  Authenticated APIs → Book appointment
- Testing public API
  curl -X GET -H ⇒ header info.
- "User-Agent: Mozilla/5.0" → some browsers react to only specific user agents
- Authentica^n → some APIs must be protected
- Require a 'token' that only a valid user can have, securely gives token only when users log in, API keys

L6.5 Open API
- APIs of interest → Purpose: info. hiding and unbreakable contract: std., should not s
- documenta^n – highly subjective, incomp., outdated, human language specific
- descrip^n files → machine readable, enable

automated processing (assembly lang.), eng. lang. specifica" which needs someone to write code

- OAS - Open API Specifica" → vendor neutral format for HTTP based remote API specifica", we can't describe all possible APIs, originally dev. as Swagger, efficiently describe the comm. use cases, current: OAS 3 - v3.1.0 (Aug 2021)

L 6.6 Impt. Concepts of an API

- describe in YAML / JSON, specific struc. to indicate overall info.
- Endpt. list

Open API Obj.

- open API
- info
- paths

Path obj.

- /endpt.1
- /endpt.2
- /endpt.3

Path item obj.

- get
- put
- post
- delete

Opⁿ obj.

- summary
- descripⁿ
- req. body
- responses

Response obj

- 404
- 200

- Eg. Tic Tac Toe paths, we can have complex schema
- prefer design-first, single source of truth, source code version control, OpenAPI is ... open-public documentaⁿ better to identify problems, automated tools, editors - make use of them

Week-7

L7.1 Memory Hierarchy

- types of storage elements.
  1. on-chips registers : 10s - 100s of bytes
  2. SRAM (static) (caches) : 0.1 - 1 MB
  3. DRAM (dynamic) (PC RAM) : 0.1 - 10 GB
  4. solid state disk (SSD) : 1 - 100 GB (flash)
  5. hard disk drive (HDD) : 0.1 - 10 TB (magnetic disk)
  6. optical, magnetic, holographic, ...
- latency : time to read first value from a storage loc^n (lower is better) 1 < 2 < 3 < 4 < 5
- throughput : no. of bytes/s (higher the better) 2 > 3 > 4
- density : no. of bits stored per unit/cost (higher is better) 5 > 4 > 3 > 2
- CPU has as many as possible, L1 L2 L3 caches ⟹ DRAM ⟹ SSD ⟹ HDD
- backups and archives, high latency of retrieval (upto 48 hrs), very high durability, very low cost
- plan the storage based on app. growth, speed of app determined by types of data stored, some data stores are more efficient for some types of op^{ns}.
- DE must be aware of choices and what kind of DBs to choose for a given app.

L7.2 Data Search

- O( ) → used in study of algorithmic complexity, rough approx.
  O(1) - constant time indep of input
  O(log N) - log in input size, grows slowly
  O(N) - linearly, often the baseline

$O(N^k)$ - polynomial - not good as input gr
$O(k^N)$ - exp., very bad
- searching for a element in memory, unsorted data in a linked list ; $O(N)$
- sorted data in array - binary searching - $O(\log N)$
- problems - size must be fixed, adding new entries requires resizing, maintaining sorted order, deleting
- better alternatives
1. binary search tree - growth of tree
2. self balancing - BST tells at one time, red-black, AVL, B-trees more complex
3. hash-tables - compute an index for an element

L7.3 DB Search
- tabular struc. - tables with many columns, we want to search quickly through it, we maintain 'INDEX' of columns
- mySQL - comparison of B-Tree and hash indexes ⟹ index-friendly and un-friendly queries
- multi-col. index → compound index
- hash-index
1. only use in-memory tables
2. only for equality comparisons
3. does not help with order by
4. partial key prefix can't be used
5. but very fast where applicable

- query optimisation → DB specific
- make use of struc. in data to organise it properly

L7.4 SQL vs NoSQL

- SQL – query DB with struc., closely tied to RDBMS (relational), all entries in a table must have same set of columns, tabular DBs → but problems
- document DBs – free-form (unstructured documents), JSON encoded, Eg. mongo DB, amazon document DB
- Key-value – Python dictionary, C++ ordered map, map a key to a value, store using search trees or hash tables, very efficient Eg. Redis, Berkeley DB, memcached, in-mem
- column stores – traditional RDBs store all values of a row together, instead store all entries in a column together. Eg. Cassandra, HBase
- Graphs – social network, diff. degrees, path finding more impt. than just search. Eg. Neo4J, Amazon Neptune
- time-series DB – very app. specific, used for log analysis, performance analysis, scaling. Eg. RRD tool, influx DB, Prometheus
- NoSQL – not-only SQL, additional querying patterns for other types of data stores
- ACID – transaction
  - Atomic, Consistent, Isolated, Durable

- many NoSQL DBs sacrifice some part of ACID for performance, but there can be ACID compliant NoSQL DBs as well
- consistency hard to meet, eventual consistency easier to meet, financial transac^ns are essential. eg. ACID
- in-memory - fast, doesn't scale - Disk - diff. data struc., organizing ^n needed

L7.5 Scaling

- Redundancy - multiple copies of same data, connec^n with backups, 1 copy is still the master
- Replica^n - in context of performance, multiple copies of same data
- BASE - Basically available, soft state - and eventually consistent, stress on high availability of data
- RDBMS replica^ns are possible, load balancer
- Scale-up → traditional, larger machine, More RAM, faster network, restart
- Scale-out → multiple servers, harder to enforce consistency, better suited to cloud model (not good for ACID
- highly app-specific, financial transac^n is not possible, for social media BASE is fine

17.6 Security of DBs

- non-MVC app - can have direct SQL queries anywhere
- MVC - only in controller, but any controller can trigger a DB query
- parameters from HTML takes w/o validation
- "Valida" - valid text data, no punctuation
- validation should be done on the DB query
- Web App. Security
  1. sql injection - use known frameworks
  2. buffer overflows, input overflows
  3. session hijacking - protocol implementation
  4. possible outcomes - loss of data, exposure of data, manipulations
- HTTPS - secure sockets, secure communication
  - server certificate
  - it secures the link but does not perform validation, caching of resources like static files, some overhead on performance

Week-8

18.1 App. Frontends

- user-interfacing, gen. GUI app., device/OS specific controls and interfaces, web browsers, standardizing, browsers vs Native
- HTML + CSS + JS → borrows interac^n
  - what to show → look up
- how to generate these lang., functional reuse, common framework, server/client load implica^n, security
- all or most of web pgs are statically generated excellent for high performance, adapt to run-time conditions; popular generators - Jekyll, Hugo, Next.js, Gatsby
- run-time HTML gen. — Traditional CGI/WSGI based apps - Flask, Django, Ruby (RoR)
- traditional CMS app. - Wordpress, Drupal, Joomla
- client load → client does more; scripting JS → NodeJS
- Tradeoffs
  1. server-side rendering - flexible, easy, less security issues, load on server
  2. static - cache-friendly, very fast, interac^n impossible, compila^n phase
  3. client side - less load on server but still client, potential security issues
- static speed (10000 req/s) dynamic (100 req/s)

18.2 Asynchronous Updates

- in old times, for updating any pg., client had to render whole pg. all

UX → User experience

CLASSTIME | Page No.
Date / /

over again → screen load, screen render (more work), slow updates
- Asyn. update → update only part of the pg. quick response gives better UX. Eg. AJAX
- refresh part of the doc. based on asyn. queries to server
- DOM (Document obj. model)
1. prog. interface for web doc.
2. DOM is abstract (tree struc.) of doc.
3. Obj. oriented manipulates like known
4. tightly coupled with JS
- study from → developer.mozilla.org
- DOM manipula^n through programs, lot more flexibility, complexity in front-end

L8.3 Browsers / Client Operations
- minimal req. → rendered HTML, cookie interac^n, accept cookies to allow sessions, text mode browsers (lynx, elinks)
↳ no image, ltd. styling
- page should not rely on colours or font size to convey meaning
- page styling - CSS, difficult in text browsers, freedom to user & browser
- interactivity - JS (de facto std.), independently to create more complex forms
- JS engines - Chrome / Chromium / Brave / Edge : V8, Firefox : Spider Monkey
↳ best in performance
- client load - JS engines use client CPU power, extensive graphics support

- energy drain - website carbon.com
- machine client - not always human, embedded devices (post sensor info.) they can't handle JS - only HTTP.
- Python inside a browser → brython.info
  ↳ transpilation (translation + compilation)
- WASM → web assembly, binary instruction format, similar to JAVA.! executable format for web, handles high performance execution
- Emscripten → compiles C & C++ to WASM potential for creating high perfor. code, limited usage so far.
- Native mode - File sys., Phone, SMS, web payments, functionality can be exposed through APIs - eg. platform support
  → adds additional security concerns

L8.4 Client-side computation and security implication
- validation - server-side validation essential, it reduce hits on server, validation in backend and front-end, extra-work but better UX
- inbuilt HTML5 form controls - min-, max, req., type, pattern
- JS validation - constraint validation API
- Captcha -: automate web pages, Railway Tatkal, CoWin appointments ⇒ proves u are a human, ltd. no. of click generate some token (required)

- Crypto-mining — JS is comp. lang., modern JS engines very powerful
- sandboxing — secure area that JS engine has access to, cannot access files, similar to a VM but at a higher level.
- DoS — denial of service, runs a script that takes over web browser and runs at high load, potentially exploit bugs in browser, secure attack
- access to native resources — permit only these browsers
  - reduce browser overheads
  - smoother interac^n with sys.

} APIs

Week-9

C9.1 Access Control -

- access - being able to read/write/modify info., not all info. is for public access
- Types - read-only, CRUD, modify but not create, etc.
- root/admin/superuser has the power to Δ permissions
- Discretionary -
  - you have control over who to share
  - forwarding, Δ access modes possible
- Mandatory -
  - not share info. w/o permission
  - in military or high security reasons
- Role-based access control - access associated with role instead of username, single user can have multiple roles
- Attribute-based - attribute, extra capability over role-based
- permissions - static rules usually based on simple checks
- policies - more complex conditions possible can combine policies
- Principle of least privilege - entity should have minimal access required to do the job. Benefits - better security, stability, ease of deployment
- Privilege escalation - user can gain an attribute ('sudo'), usually combined with explicit logging, extra safety measures
- Web apps - admin dashboards, user access

- enforcement
① Hardware - security key, hardware token, locked doors
② OS - filesys. access, memory segmentⁿ
③ App. - DB servers can restrict access to specific DB
④ Web app. - controllers enforce restrictions, decorators in Python used in Flask

L9.2 Security Mechanisms

- Obscurity - app. listens on non-std. port known only to specific people
- Address - host based access / deny controls
- Login - user/psswd provided to each person needing access
- Tokens - access tokens can't be duplicated, can be used for machine to machine authenticaⁿ w/o passwords
- HTTP authenticaⁿ - enforced by server, 401 / unauthorized client, 404 - not found, 403 - forbidden
  - Problems - plain text (base 64 coding), no std. process for logout
- Digest authenticaⁿ - message digest, cryptographic funcⁿ, one-way funcⁿ, can define such funcⁿˢ on strings
- HTTP digest authenticaⁿ - nonce → no. used once, server & client know all parameters
- Client certificates - given to each client, kept sent secure on client side

- form input — transmitted over link to server, get (very insecure, open to spoofing), post (slightly secure, better secure links needed)
- eg. conn. security — "one TCP connec", w/o connec" KeepAlive
- cookies — sessions are maintained, server checks some client credentials, then 'sets' a cooking'
- API security — eg. interactive use, basic auth pop-up windows, use token as API key for access.

L9.3 Sessions

- client send multiple req., same some 'state' info., server customizes responses based on client session info.
  client → stored in cooking
- cookies are set by set-cookie browser, can be used to store info.
- cross-site requests — same machine, same browser
- server-side info. → maintain client info at server, cookie provides minimal info
- some parts of site should be protected, protect access to controller → use decorator
- transmitted data security

CLASSTIME | Page No.
Date / /

L9.4 Logging
- record all accesses to app
- server logging → build in to Apache, Nginx, just accesses and URL accessed
- app.url.logging → python logging framework, details of app. access, all server errors
- log rotation → high vol. logs, mostly written, can't store indefinitely
- logs on custom app engines, google app engine
- Time series analysis → logs associated with timestamps, RRDTool, etc use time-series DB
- Analyze → Identify → Fix

L9.5 HTTPs
- normal HTTP → open connect to server on port 80, transmitted data can be tapped or altered
- secure socket layer → set up an 'encrypted' channel between client and server
- types of security →
  1. channel (wire) security
  2. server authentication
  3. client certificate
- chain of trust
- problems → old browsers, stolen certificates at root of trust, DNS hijacking
- HTTPS → security against wiretapping, better in public WiFi networking

CLASSTIME | Page No.
Date / /

Week-10

L10.1 Applicaⁿ testing

- why?
  - requirements - specificaⁿ
  - respond correctly to inputs
  - respond within reasonable time
  - installaⁿ and envt.
  - usability & correctness
- static testing ← (code not working)
  - code review, correctness proofs
- dynamic testing ↑ (code working)
  - functional tests
  - apply suitable inputs
- white-box testing
  - detailed knowledge of implementaⁿ
  - can examine int. var.
  - tests can be created based on int. state.
  - more detailed info.
  - can lead to focus on less impt. parts
  - clean abstraⁿ is absent
- black-box testing
  - only interfaces are available
  - test based on how to look from outside
- grey-box testing
  - hybrid approach
  - enforce interface as far as possible
- Regression - loss of functionality introduced by some 's in-the code. Future modificaⁿ should n't break the existing code. This is sometimes necessary.
- Coverage - how much of the code is covered branch, condition, funcⁿ coverage ...

- code coverage useful metric → doesn't guarantee all scenarios actually tested

L10.2 Levels of Testing

- who are stakeholders?, functionality, non-functional requirements
- functional req. of a particular page and non-functional req.
- requirements gathering — discussion with end-users, avoid lang. ambiguity, capture use cases & eqs.
- break functional req. down to small, implementable units, each one may become a single controller
- test each indi. unit of implementa"; may be single controllers, clearly define inputs & expected outputs
- Integra" — app consists of several modules, continuous integration (CI), combined with version control systems
- sys. level testing → after integra", includes server & env., mainly it includes black-box-testing
- System-testing automa" — Simulate actual user interac", eg. selenium (between automa")
- user-acceptance testing — deploy final testing, beta-testing

L10.3 Test-generation

- API based testing — APIs are abstraction for sys. design, openAPI, swagger
- use cases
  - input API def^n
  - generate test for specific endpts
  - record API traffic
- abstract test — semi-formal verbal descrip^n
- model-based testing — scenarios & model
- models abstract tests — abstract tests to apply to generic models
- GUI testing — user interface: visual output
- browser automation → some tests can't be directly auto programmatically, req. generation Eg. Capybara (Ruby)
- Security testing — generate invalid inputs, try to crash server, fuzzing — generate large no. of random/semi-random inputs

L10.4 PyTest

- framework to make testing easier in Python
  unittest → pytest alternative
- temporary directory etc.
  → (assert 0) is always false
- test fixtures — set up some data before test, remove it after test
  Eg. initialize dummy db

- conventions - test directory starts from current dir on test paths visibly search ten file name test.py
- testing flask applica" - create a client fixtures - known to flask
- continuous testing is essential for overall system stability

Week-11

L11.1 Beyond HTML

- markup lang. → origin from 60s, mostly used for typesetting and document mgmt. sys.
  → lack of standardizn, machine readability
- SGML → meant to be a base frame which any markup lang. could be designed
- DTD → document type defn, used to specify diff. families within the umbrella, each could have its own tags
- HTML - very lenient with parsing, HTML5 is not SGML appn — defines its own parsing rules
- XML - based on SGML, custom tags, easily readable, well-structured, focus on simplicity, generality, usability, Eg. Rss feeds, SVG
- XHTML - based on XML, reformulan of HTML4, modular & more extensible, XML namespaces
- HTML5 - away from SGML, add support for latest features, remains easily readable, remain backward compatible
- Extension → JS → allow new tags to be added, custom elements, very powerful mechanism, merging of new tags could be not properly rendered

L11.2 JS

- high level lang., dynamic typing, DOP
- multi-paradigm, event-driven, functional comparative

- relatively easy to learn, similar to Python
- most web-browsers have a dedicated engine, APIs, most powerful when used for DOM manipulation

Lec 3: Custom Elements
- XML allows arbitrary namespaces & tag def", but HTML5 doesn't have this approach.
- req. for elements —
  - meaning
  - rendering
  - states
  - customized | automa" buttons
- Web components — custom elements, Shadow DOM, HTML templates
  - ↳ API to keep styling of comp. seperate from the rest of pg.
- Goal: reuse
  Problem - limited standardiza" } web components

Lec 4: Frameworks
- purpose —
  (i) basic functionality already available
  (ii) problem - code repetition, reinventing the wheel
  (iii) sol" - std. techniques, flask for web python apps, react for JS comp.
  (iv) SPA - single pg. applica"
- Eg. React → library for building user interfaces

CLASSTIME | Page No.
Date / /

- declarative components (diff. from web components)
- angular — from google team, 2010
- embeerJs — comp. & service framework
- Vue
- HTML + CSS + JS = best!
- but difficult to code, frameworks in the gaps
- front-end dev. for dynamic applications

CLASSTIME | Page No.
Date / /

Week-12

L12.1 Components of an App

- developing an App — Idea, local deployment, single computer, multiple services
- permanent deployment — dedicated servers, always on-internet connec", uninterrupted power; ⇒ infrastructure → data centres, cloud
- scaling → https + load balancer → logging
  user → front-end → database
  " "
  " "
  content delivery network
- base dev. is an easy task
- deployment is hard
- infrastructure → use cloud (cloud service providers)

L12.2 Service Approach-1

- specializa", data centres (infrastructure), dev. (app dev.), std. software deployments
- SAAS (software as a service)
- online office, CMS (wordpress), issue tracking
- hosted sol" — all the software is installed & maintained
- IAAS (infrastructure)
- raw machines, power, install your OS
- cloud sys → AWS, Azure, Google compute engine
- platform — comb" of hardware & software, custom app. code, specific " & " specifica"
- PAAS (platform)
- dev. need to manage code, and specify requirements

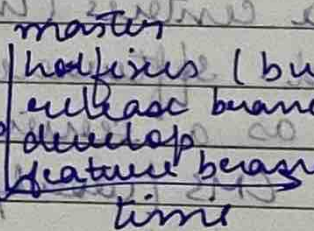
CLASSTIME | Page No.
Date / /

L12.3 Service Approach - 2

- google app engine → google cloud - console.cloud.google.com
- cloud shell editor
- very professional
- VS code is itself written in JS
- PAAS - provides platforms to build on
- varying degrees of complexity, ease of use
- integrate with other code dev practices
  - version control
  - conti integrat^n & deployment
  - scaling & automa^n

L12.4 Deployment

- version control → Git, retain backups of old code, develop new features, fix bugs
- Git flow



- centralized - central server, many clients push Δs to server each time
- distributed - can have central server not needed, Δs managed by 'patches'
- Conti integra^n - practice of automating the integra^n of code Δs from multiple contributors into a single software project
- CI workflow - automat^n
- best practices - test + design deployment

code review, integration pipeline optimization
- CD (continuous delivery / deployment)
  ↳ once CI passed, package files for release, automated delivery of 'release package' on each successful test
- nightly builds, beta versions
- CD ⇒ extend beyond delivery: deploy to prod"
  ↳ tests may not catch all problems

L 12-5 Containers
- self contained envt. with OS & libraries
- full OS impossible to version control
- create self-contained images
- sandbox ≠ image can't affect other processes on system
- kernel well supported needed
- orchestration — app consists of multiple processes, communicate btw. processes that are isolated, mechanisms to automate, key to understand & manage large-scale deployment
- HTML + CSS + JS → front-end
- dB, NoSQL, cloud store → back-end
- authentication, proxying, load balancing → middleware
- PAAS — deployment & mgmt.