Week 1 Lecture 1

Class	BSCCS2003
Created	@September 4, 2021 7:51 PM
Materials	
Module #	1
Type	Lecture
Week #	1

What is an App?

"An app is a computer software, or a program, most commonly a small, specific one used for mobile devices. The term app originally referred to any mobile or desktop application, but as more app stores have emerged to sell mobile apps to smartphone and tablet users, the term has evolved to refer to small programs that can be downloaded and installed at once"

- Source: Techopedia

Desktop Apps

- Usually standalone
  - Editors / Word processors
  - Web browsers
  - Mail clients
- Often work offline
  - Local data storage
  - Possible network connection
- Software Development Kits (SDK)
  - Custom frameworks
  - OS specific

Mobile Apps

- Targeted at mobile platforms: Phones / Tablets

- Constraints
  - Limited screen space
  - User interaction (touch, audio, camera)
  - Memory / processing
  - Power
- Frameworks
  - OS specific
  - Cross-platform
- Network!
  - Usually network oriented

Web Apps

- The **Platform**
- Works across OS, device: create a common base
- Heavily network dependent
  - Workarounds for offline processing

Our main focus in this course ...

Week 1 Lecture 2

Class	BSCCS2003
Created	@September 4, 2021 8:10 PM
Materials	
Module #	2
Type	Lecture
Week #	1

Components of an App

Storage

Computation

Presentation

Example

- Storage
  - Where are the e-mails stored?
  - How are they stored on the server? File formats, etc.
- Compute
  - Indexing of e-mails
  - Searching
- Presentation
  - Display list of e-mails
  - Rendering / display of individual e-mails

Platforms

- Desktop
  - Keyboard, Mouse, Video as I / O
  - Desktop paradigm - folders, files, documents

- Mobile
  - Touchscreen as I / O
  - Voice, tilt, camera interfaces
  - Small self-contained apps
- Web-based
  - Datacenter storage - persistent
  - Cloud: access anywhere multi-device
- Embedded
  - Single-function, limited scope

Week 1 Lecture 3

Class	BSCCS2003
Created	@September 5, 2021 7:38 PM
Materials	
Module #	3
Type	Lecture
Week #	1

Client-Server & Peer-to-Peer Architecture

Architectures

- **Client-Server**
    - Server:
        - Stores data
        - Provides data on-demand
        - May also perform computations
    - Client:
        - End users
        - Requests data
        - User interaction, display
    - Network:
        - Connects the client to the server
        - Can be local
        - Data pipe - no alterations
    - **In a client-server model, the server and client are explicitly defined**
    - Local systems:
        - Both client and the server on the system ⇒ local network / communication

        - Conceptually, it is still a networked system
    - Machine clients
        - eg: Software, antivirus updaters
        - They need not have user interaction
    - Variants
        - Multiple servers, single queue, multiple queues, load balancing front-ends
    - Examples:
        - E-mail
        - Databases
        - WhatsApp / messaging
        - Web browsing
- **Distributed - Peer-to-Peer**

  Data can flow both ways
    - All peers are considered "equivalent"
        - But some peers may be more equal than others

          This just means that some servers with high bandwidth are given higher weightage
    - Error tolerance
        - Masters / Introducers
        - Election / re-selection of masters on failure
    - Shared information
    - Examples:
        - BitTorrent
        - Blockchain-based systems
        - IPFS, Tahoe (distributed file system)

Week 1 Lecture 4

Class	BSCCS2003
Created	@September 5, 2021 8:33 PM
Materials	
Module #	4
Type	Lecture
Week #	1

Software Architectures

What is a design pattern?

"A general, re-usable solution to a commonly occuring problem within a given context in software design"

Source: Wikipedia

- Experienced designers observe "patterns" in code
- Re-using these patterns can make design and development faster
- Guide the design and thought process

M-V-C paradigm

- **Model:**
  - Core data to be stored for the application
  - Databases: indexing for easy searching, manipulation, etc;
- **View:**
  - User-facing side of the application
  - Interfaces for finding information, manipulating
- **Controller:**
  - "Business Logic" - how to manipulate the data

Origins in the smalltalk language, 1979

The user uses the **controller** → to manipulate the **model** → that updates the **view** → that the user sees;

Other design patterns

- Model-View-Adapter
- Model-View-Presenter
- Model-View-ViewModel
- Hierarchical MVC
- Presentation-Abstraction-Control

  Each one of them has its own uses, but the fundamentals are very similar

Focus of this course

- Platform:
  - Web-based
- Architecture:
  - Client-server
- Software architecture
  - Model-View-Controller

Week 1 Lecture 5

Class	BSCCS2003
Created	@September 6, 2021 12:07 PM
Materials	
Module #	5
Type	Lecture
Week #	1

Introduction to the Web

- Platform of choice for this course
- Generic - works across operating systems, hardware architectures
    - Can it be considered as a cross-platform OS?
- Built on sound underlying principles
- Worth understanding
    - Constraints: what can and cannot be done (easily)
    - Costs: storage, network, device sizing, datacenter

Historical background

- Telephone networks ~ 1890+
    - Circuit switched - allow A to talk to B, through complex switching network
    - Physical wires tied up for duration of call even if nothing was said!
        - The bandwidth used by voice signal is pretty low
- Packet switched networks ~ 1960s
    - Wires occupied only when data was to be sent - more efficient use

      Last link: The physical connection between my telephone and my nearest telephone exchange, it has to be tied up to me

      Aggregate all the data in one place (telephone exchange) and transmit it over a single wire at a much higher speed (connection between telephone exchanges)

    - Data is being sent instead of voice over the wires
- ARPANet - 1969
    - acronym for Advanced Research Projects Agency Network
    - Node-to-node network
    - Mostly university driven
- Others:
    - IBM SNA, Digital DECNet, Xerox Ethernet, ...
- Protocol
    - How to format packets; place them on a wire; headers / checksums;
    - Each network had its own protocol
- "Inter" network
    - How to communicate between different network protocols?
    - Or replace them with a single "inter"net protocol
- IP: Internet Protocol - 1983
    - Define headers, packet types, interpretation
    - Can be carried over different underlying networks: ethernet, DECnet, PPP, SLIP
- TCP: Transmission Control Protocol - 1983
    - Establish reliable communication - retry; error control
    - Automatically scale and adjust to network limits
    - It is a connection oriented protocol
- Domain names ~ 1985
    - Use names instead of IP addresses
        - IP addresses were basically a set of 4 numbers, what they called the dotted quad notation
        - There are 4 numbers in the range of 0 - 255, which are separeted by dots (.)
    - Easy to remember - $.com$ revolution still in the future
- HyperText ~ 1989+
    - Text documents to be "served"
    - Formatting hints inside documents to "link" to other documents - HyperText
    - It was brought into one consolidated framework by Tim Berners Lee at CERN
    - All this lead to the creation of the World Wide Web

Where are we now?

- Original web was limited to ...
    - static pages
    - complicated executable interfaces
    - limited styling
    - browser compatibility issues
- Web 2.0 ~ 2004+
    - dynamic pages - generated on the fly
    - HTTP as a transport mechanism - binary data; serialized objects
    - client side computation and rendering
    - platform agnostic operating system

Week 1 Lecture 6

Class	BSCCS2003
Created	@September 6, 2021 2:01 PM
Materials	
Module #	6
Type	Lecture
Week #	1

How does the Web work?

Digging deeper

- What is a server?
- What is a transport protocol?
- Impact on performance:
  - Network
  - Server compute resources
  - Storage requirements
  - Client compute resources

Web Server

- Any old computer with a network connection can act as a web server
- Software:
  - Listen for incoming network connections on a fixed port
  - Respond in specific ways
  - Opening network connections, ports etc already known to OS
- Protocol:
  - What should client ask server
  - How should the server respond to the client

HTTP

- HyperText
  - Regular text documents
  - Containes "codes" inside that indicate special functions - how to "link" to other documents
- HyperText Transfer Protocol
  - Largely text based: client sends requests, server responds with hypertext documents

Week 1 Lecture 7

Class	BSCCS2003
Created	@September 6, 2021 2:43 PM
Materials	
Module #	7
Type	Lecture
Week #	1

Simple Web Server

Simplest server

```
#!/bin/bash

while true; do
  echo -e "HTTP/1.1 200 OK\n\n $(date)" |
    nc -l localhost 1500;
done
```

Create a new file on the linux terminal

```
$ nano server.sh
```

Write the above mentioned server bash script in that file, and then to run the script, type the following code on the linux terminal

```
$ bash server.sh
```

We can send a request to the running server from another instance of linux shell

```
$ curl http://localhost:1500
```

We can also add a verbose flag to get a ton more info

```
$ curl -v http://localhost:1500
```

What the general web server is actually doing?

- Listening on a fixed port
- On incoming requests, it ran some code, generated results on the fly and returned the results to the client
  - Standard headers to be sent as a part of the result
  - Output can be in text or other format - MIME (Multipurpose Internet Mail Extensions)

What does a typical request look like?

```
GET / HTTP/1.1
Host: localhost:1500
User-Agent: curl/7.64.1
Accept: */*
```

Viewing responses with curl

- curl, wget, etc - simple command line utilities
- Can perform full HTTP requests
- Verbose output includes all the headers
- Very useful for debugging!

What does a typical response look like?

When you make a simple http request to a web server via curl, the following response appears ...

```
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 1500 (#0)
> GET / HTTP/1.1
> Host: localhost:1500
> User-Agent: curl/7.64.1
> Accept: */*
>
< HTTP/1.1 200 OK
* no chunk, no close, no size. Assume close to signal end
<
 Thu Jun 17 08:14:55 IST 2021
```

NOTE: The **::** 1 is an IPv6 address which represents $localhost$

Week 1 Lecture 8

Class	BSCCS2003
Created	@September 6, 2021 3:26 PM
Materials	
Module #	8
Type	Lecture
Week #	1

What is a Protocol?

- When both sides agree on how to talk
- Server expects "requests"
    - Nature of requests
    - Nature of client
    - Types of results that the client can deal with
- Client asks "responses"
    - Asks server for something
    - Convey what you can accept
    - Read the result and process

HyperText Transfer Protocol (HTTP)

- Primarily text based
- Requests specified as "GET", "POST", "PUT", etc
    - Headers can be used to convey acceptable response types, languages, encoding
    - Which host to connect to (if multiple hosts are there on a single server)
- Response headers
    - convey message type, data
    - cache information

- status codes: 404 not found, etc

Use cases

- **GET**: simple requests, search queries, etc
- **POST**: more complex form data, large text blocks, file uploads
- **PUT** / **DELETE** / ...
  - Rarely used in Web 1.0
  - Extensively used in Web 2.0
  - Basis of most APIs - REST, CRUD

Another web server

```
$ python -m http.server
```

- Serve files from local directory
- Understands basic HTTP requests
- Gives more detailed headers and responses

When the above command is run in a given directory, if I find a file called $index.html$, return me that file else return me the directory listing

Week 1 Lecture 9

Class	BSCCS2003
Created	@September 6, 2021 3:44 PM
Materials	
Module #	9
Type	Lecture
Week #	1

Performance of a Website

- How fast a site can be?
- What limits the performance?
- Some basic observations ...

Latency

How much time does it take to get the response for a given request

- Speed of light: $3 \times 10^8 m/s$ in vacuum
- ~$2 \times 10^8 m/s$ on a cable
  - ~5ns for 1m $\implies$ ~5ms for 1000km
- Datacenter is 2000km away
  - One-way request: ~10ms
  - Round-trip: 20ms

So, what does this mean?

If I am continuously sending requests to a server, and I want it such that whenever I send a request, I get a response back before I can then proceed, then

I am limited to a maximum of 50 requests per second

What about a server that needs to send back responses?

Response size

- Response: 1KB of text (headers, HTML, CSS, JS)
- Network connection = 100Mbps (Megabits per second) $\approx$ 10MBps (Megabytes per second) [Theoretically, it should be 12.5MB/s]
- This limits us to ~10,000 requests per second

  For eg: Just imagine, that the result of some entrance exams were declared in India and if it is all trying to come out of one server with the bandwidth of 100 Megabits per second

  The server would simply crash when it is bombarded with more than 10,000 requests per second

A simple case study: Google response

- Headers: ~100B (bytes)
- Content: 144kB (kilobytes)
- Approx. 60,000 requests per second

Which means that Google server effectively need more than 80Gbps of bandwidth

Memory: YouTube

- One python HTTP server process: ~6MB (measured)
- Multiple parallel requests: multiple processes
  - eg: YouTube will have long running server processes
- 2016 Presidential debate in the US:
  - 2M+ concurrent viewers

Storage: Google

- Index 100s of billions of web pages
- Cross-reference, pagerank
- Total index size estimate:
  - 100,000,000 Gigabytes $\implies$ 100 Petabytes
- Storage?
- Retrieval

Summary

- The Web is a useful device / OS agnostic platform for apps
- Built for HTTP for transport, HTML and related tech for presentation
- Servers trivial at most basic
- Scaling requires careful design

Week 2 Lecture 1

Class	BSCCS2003
Created	@September 14, 2021 12:16 PM
Materials	
Module #	10
Type	Lecture
Week #	2

Information Representation in a Machine

Markup

- Information representation
- Raw data vs Semantics
- Logical structure vs Styling
- HTML5 & CSS

Information representation

- Computers work only with "bits"
  - Binary digits: 0 and 1
- Numbers
  - Place value: binary numbers: eg. 6 = 0110
  - Two's complement: negative numbers: eg. -6 = 1010

Representing Text

- ASCII
- Unicode
- UTF-8

Information Interchange

- Communicate through machines - either between machines or between humans
- Machines only work with ***bits***
- Standard "encoding"
  - Some sequence of bits interpreted as a character

Interpretation

What is "0100 0001"?

- It is a string of bits
- Number with a value of 65 in decimal
- Character "A"
- All of the above statements are correct

It boils down to a matter of **interpretation** and **context**

ASCII

American Standard Code for Information Interchange

- 7-bits: 128 different entities
  - 'a'..'z'
  - 'A'..'Z'
  - '0'..'9'
  - Special characters- ! @ # $ % ^ & * ( ) ...
- Why 7-bits?

  It was a time before even 8-bit computers were standardised

  The number of bits were precious and you really didn't want to use up more than necessary bits

  That is why people tried to optimise the number of bits
- What about other characters? あ आ 아 ᏏᏍ
  - 1000s of characters needed

Unicode

- Allow codes for more script, characters
- How many?
  - All the living languages?
  - All extinct languages?
  - All future languages?
- **"Universal Character Set"** encoding - UCS
  - **UCS-2** 2 bytes per character - max 65,536 characters
  - **UCS-4** 4 bytes per character - 4B+ characters

Week 2 Lecture 2

Class	BSCCS2003
Created	@September 14, 2021 4:50 PM
Materials	
Module #	11
Type	Lecture
Week #	2

Efficiency of encoding

Efficiency

- What is the most common language on the web?
  - As of now, it is English
- Should all the characters be represented with the same number of bits?
  - Example:
    - A text document with 1,000 words, which equates to approximately 5,000 characters (including spaces)
    - According to UCS-4 encoding

      32 bits $\times$ 5,000 = 160,000 bits
    - According to ASCII encoding

      8 bits $\times$ 5000 = 40,000 bits
    - According to the original 7-bit ASCII which was sufficient for English

      7 bits $\times$ 5000 = 35,000 bits
    - Minimum needed to encode just 'a' - 'z', numbers and some special characters

      could fit in 6-bits: 30,000 bits
    - Optimal coding based on frequency of occurrence of a letter
      - 'e' is the most common letter, 't', 'a', 'o', ...
      - Huffman or similar encoding: ~10,000 - 20,000 bits, possibly less

Is this problem solvable in general?

- It is impossible to encode by actual character frequency: depends on the text
  - Just use compression methods like "zip" instead
- But can encoding be a good halfway point?

Example:

- Use 1-byte for most common alphabets
- Group others according to frequency, have "prefix" codes to indicate

Prefix Coding

1st byte	2nd byte	3rd byte	4th byte	Free bits	Max. expressible Unicode val.
0xxxxxxx				7	007F hex (127)
110xxxxx	10xxxxxx			(5+6)=11	07FF hex (2047)
1110xxxx	10xxxxxx	10xxxxxx		(4+6+6)=16	FFFF hex (65535)
11110xxx	10xxxxxx	10xxxxxx	10xxxxxx	(3+6+6+6)=21	10FFF hex (1,114,111)

- If the first bit starts with a **0**, take the remaining 7 free bits as an ASCII code

Example

	A	ℵ	好	不
Code point	U+0041	U+05D0	U+597D	U+233B4
UTF-8	41	D7 90	E5 A5 BD	F0 A3 8E B4
UTF-16	00 41	05 D0	59 7D	D8 4C DF B4
UTF-32	00 00 00 41	00 00 05 D0	00 00 59 7D	00 02 33 B4

Source: https://www.w3.org/International/articles/definitions-characters

UTF-8

- Stands for Unicode Transformation Format
- Uses 8-bits for most common characters: ASCII subset
  - All ASCII documents are automatically UTF-8 compatible
  - All other characters can be encoded based on prefix string
  - More difficult for text processors:
    - first check prefix
    - linked list through chain of prefixes possible
    - Still more efficient for majority of documents
  - Most common encoding in use today

Week 2 Lecture 3

Class	BSCCS2003
Created	@September 14, 2021 6:39 PM
Materials	
Module #	12
Type	Lecture
Week #	2

What is Markup?

Topics

- Content v/s Meaning
- Types of Markup
- (X)HTML

Raw content

```
Markup What is markup? Markup is a way of using cues or codes in the regular flow to indicate how text should be displayed. Ma
rkup is very useful to make the display of text clear and easy to understand
```

Types of Markup

- Presentational
  - **WYSIWYG:** directly format output and display
    - What You See Is What You Get
  - Embed codes not part of regular text, specific to the editor
- Procedural
  - Details on how to display something
    - Change the font to large, bold
    - skip 2 lines, indent 4 columns

- Descriptive
  - This is a `<title>`, this is a `<heading>`, this is a `<paragraph>`

Examples

- Microsoft Word, Google Docs, etc
  - User interface focused on "appearance", not meaning
  - WYSIWYG: direct control over styling
  - Often leads to complex formatting and loss of inherent meaning
- LaTeX, HTML (general *ML)
  - Focus on meaning
  - More complex to write and edit, not WYSIWYG in general
  - Fun fact: The current raw document was written in LaTeX and rendered to a PDF later

Semantic Markup

- Content v/s Presentation
- Semantics
  - Meaning of the text
  - Structure or Logic of the document

Week 2 Lecture 4

Class	BSCCS2003
Created	@September 14, 2021 7:21 PM
Materials	
Module #	13
Type	Lecture
Week #	2

Introduction to HTML

Topics

- HyperText Markup Language
- Generalizations
- Variants of Interest

HyperText Markup Language (HTML)

- HTML was first used by Tim Berners-Lee in the original Web at CERN (-1989)
- Considered an application of SGML (Standard Generalized Markup Language)
  - Strict definitions on structure, syntax and validity
- HTML was meant for browser interpretation
  - Very forgiving: loose validity checks
  - Best effort to display

HTML Example

```
<!DOCTYPE html>
<html>
<body>
<h1>My First Heading.</h1>
<p>My first paragraph.</p>
</body>
</html>
```

Tags

- `<h1></h1>` - paired tags
- Represented by angled brackets `<>`
- Closing tag with `/`
- Location specific: `<DOCTYPE>` is only at the head of the document
- Case insensitive

Nesting

Valid syntax

- `<em><strong>Hello</strong></em>`
- The output is
  - ***Hello***
- Invalid:
  - `<em><strong>Hello</em></strong>`
  - `<em><strong>Hello</em>`
  - `<em><strong>Hell<o/em></strong>`

Presentation v/s Semantics

- `<strong>Hello</strong>`
- `<b>Hello</b>`
- **Hello**

Which one is right? Which one is better?

It is not an easy question to answer, both of them may be correct

`<strong>` is putting an emphasis meanwhile `<b>` does not

Timelines

- SGML based
  - 1989 - HTML original
  - 1995 - HTML 2
  - 1997 - HTML 3, 4
- XML based
  - XHTML - 1997 - mid 2010s
- HTML 5
  - first release in 2008
  - W3C recommendation - 2014
    - World Wide Web Consortium

HTML 5

- Block element: `<div>`
- In-line element: `<span>`
- Logical elements: `<nav>`, `<footer>`
- Media: `<audio>`, `<video>`

Remove "presentation only" tags:

- `<center>`
- `<font>`

Document Object Model (DOM)

> The Document Object Model (DOM) is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects; that way, programming languages can interact with the page.

Source: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

```
<html>
<head>
  <title>My title</title>
</head>
<body>
  <h1>A heading</h1>
  <a href="https://onlinedegree.iitm.ac.in">Link text</a>
</body>
</html>
```

Source: https://en.wikipedia.org/wiki/Document_Object_Model

- It is a tree structure which represents the logical layout of the document
- Direct manipulation of this tree is possible!
- Application Programming Interfaces (APIs) are also defined on the DOM:
  - Canvas
  - Offline
  - Web Storage
  - Drag & Drop
  - ...
- In most of the cases, when we talk about manipulating the DOM, we use JavaScript
- Cascading Style Sheets (CSS) is used for styling

Week 2 Lecture 5

Class	BSCCS2003
Created	@September 15, 2021 12:59 PM
Materials	
Module #	14
Type	Lecture
Week #	2

Introduction to Styling

Topics

- Markup v/s Style
- Themes
- CSS

Markup v/s Style

```
<h1>Hello</h1>
```

https://codepen.io/21f1003586/pen/bGRoJJX

https://codepen.io/21f1003586/pen/BaZweBX

Separation of Styling

- Style hints in separate blocks
  - Separate files included
- Themes

- Style sheets
  - Specify presentation information
- Cascading Style Sheets (CSS)
  - Allows multiple definition
  - Latest definition takes the precedence

Week 2 Lecture 6

Class	BSCCS2003
Created	@September 15, 2021 2:23 PM
Materials	
Module #	15
Type	Lecture
Week #	2

Types of CSS styling & Responsive Websites

Inline CSS

- Directly add style to the tag
- Example:

```
<h1 style="color: blue; text-align: center;">A heading</h1>
```

Internal CSS

- Embed inside the `<head>` tag
- Now, all the `<h1>` tags in the document will look the same - centrally modified

```
<style>
body {
  background-color: linen;
}

h1 {
  color: maroon;
  margin-left: 40px;
}
</style>
```

External CSS

- Extract common content for re-use

- Multiple CSS files can be included
- Latest definition of style takes the precedence

Responsive Design

- Mobile and Tablet devices have smaller screen
  - Different form factors
- Adapt to the screen - ***Respond***
- CSS controls the styling - HTML controls the client

<u>Bootstrap</u>

- Most commonly used framework for CSS
  - Originate from Twitter
  - Widely used now
- Standard styles for various components
  - Buttons
  - Forms
  - Icons
  - ...
- Mobile first: highly responsive layout

JavaScript

- It is an interpreted language brought into the browser
- Not really related to Java in any way - formally known as ECMAScript
- Why was JavaScript brought into the browser?
  - HTML is not a programming language
  - CSS is not a programming language too (There are ways to make it Turing complete, google it)
- We wanted to have "programmability" inside the browser
- It is not a part of the core presentation requirements
  - But still, it is very useful and will be considered later

Summary

- Presentation - Human interaction
  - Separate content from style
    - Markup - HTML
    - Styling - CSS

Week 3 Lecture 1

Class	BSCCS2003
Created	@September 18, 2021 5:06 PM
Materials	
Module #	16
Type	Lecture
Week #	3

Overview of MVC

Outline

- MVC paradigm
- Views and User interfaces
- Tools & Techniques
- Accessibility

Model: Store e-mails on the server; indexed and ready to manipulate

View: Display a list of e-mails; Read individual e-mail

Controller: Sort e-mails; delete; archive (Usually the business logic of an app)

Model-View-Controller

- Origins: Smalltalk-80
- Separation of responsibilities - **Abstraction**
- Roots in Object-Oriented GUI development

Design Patterns

- Common software patterns
- **Model:** Application object
- **View:** Screen representation
- **Controller:** How does the user interface reacts to user input

Running Example

Student Gradebook

- Input data: for **Model**
  - Student list
  - Course list
  - Student-Course marks
- Outputs: for **View**
  - Marks for individual student
  - Summary of the course
  - Histograms
- Modifications: for **Controller**
  - Add new students
  - Add new courses
  - Modify the marks in a course

Week 3 Lecture 2

Class	BSCCS2003
Created	@September 18, 2021 5:20 PM
Materials	
Module #	17
Type	Lecture
Week #	3

Views

User Interface Design

User Interface

- Screen
- Audio
- Vibration (haptics)
- Motor (door open / close)

User Interaction

- Keyboard / Mouse
- Touchscreen
- Spoken voice
- Custom buttons

User interaction is ...

- determined by the hardware constraints
- different based on various types of devices
- user-agent information useful to identify context

Types of Views

- Fully static
- Partly dynamic
- Mostly dynamic

Output

- HTML - most commonly used - direct rendering
- Dynamic images
- JSON / XML - machine readable

View - any "representation" that is useful to another entity

Week 3 Lecture 3

Class	BSCCS2003
Created	@September 18, 2021 5:37 PM
Materials	
Module #	18
Type	Lecture
Week #	3

User Interface Design

- Design for interaction with the user
- Goals:
  - **Simple:** Easy for the user to understand and use
  - **Efficient:** User achieves their goal with minimal effort
- Aesthetics
  - Something which is pleasing to the eyes and looks nice
- Accessibility

Systematic Process

- Functionality requirements gathering - What is needed?
- User and Task analysis - user preferences, task needs
- Prototyping - wireframes, mock-ups
- Testing - User acceptance, usability, accessibility

Week 3 Lecture 4

Class	BSCCS2003
Created	@September 18, 2021 6:27 PM
Materials	
Module #	19
Type	Lecture
Week #	3

Usability Heuristics

Guidelines / Heuristics

Jakob Nielsen's heuristics for design

https://www.nngroup.com/articles/ten-usability-heuristics/

- Not specific to web apps, or even software UI design
- But it is very useful and relevant

General principles

- Consistency
- Simple and minimal steps
- Simple language
- Minimal and aesthetically pleasing

Week 3 Lecture 5

Class	BSCCS2003
Created	@September 18, 2021 6:36 PM
Materials	
Module #	20
Type	Lecture
Week #	3

Tools (part 1)

- Wireframes
- HTML generation
- Templates

Wireframes

- Visual guide to represent **structure** of the web page
- Information design
- Navigation design
- User interface design

Source: https://en.wikipedia.org/wiki/Website_wireframe

Example Tools: LucidCharts

Source: https://www.lucidchart.com/pages/wireframe

Title
http://lucidchart.com
Account
Documents
Contacts
Help
Website Name and pithy phrase
Call to action
Customer quotes.
Panel 1
Content
Panel 2
Panel 3
ABOUT US * CONTACT INFO * AUTHORIZED DEALERS

Notes

1 Details about the image
2 Where this links to
3 List of possible quotes
4 Drop down menus that link to other pages
5 Content of video

Week 3 Lecture 6

Class	BSCCS2003
Created	@September 18, 2021 6:47 PM
Materials	
Module #	21
Type	Lecture
Week #	3

Tools (part 2)

Programmatic HTML generation: PyHTML

- Composable functions - each function generates a specific output
- Example:
  - To generate an `h1` heading: the function should return .. `"<h1>text of heading</h1>"`

```
import pyhtml as h

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

More complex HTML

```
def f_table(ctx):
  return (
    tr(
      td(cell) for cell in row
```

```
    ) for row in ctx['table']
  )
```

Templates

- Standard template text
- Placeholder / Variables
- Basic (very limited) programmability
- Examples:
  - Python inbuilt string templates - good for simple tasks
  - Jinja2 - used by Flask
  - Genshi
  - Mako
  - ...

Jinja

- Ties in closely with Flask
- Template functionality with detailed API

  Remember: Templates can generate any output, not just HTML

  ```
  from string import Template

  t = Template('$name is the $job of the $company')
  s = t.substitute(name='Tim Cook', job='CEO', company='Apple Inc.')
  print(s)
  ```

  Jinja example

  ```
  from jinja2 import Template
  t = Template('Hello {{ something }}!')
  print(t.render(something='world'))

  t = Template('My favourite numbers: {% for n in range(1, 10) %}{{ n }} ' '{% endfor %}')
  print(t.render())
  ```

Week 3 Lecture 7

Class	BSCCS2003
Created	@September 18, 2021 8:25 PM
Materials	
Module #	22
Type	Lecture
Week #	3

Accessibility

- Various forms of disability or impairment
  - Vision
  - Speech
  - Touch
  - Sensor-Motor
- Can a page be accessed by people with impairments?
- How can the accessibility of a page be improved?

W3.org - World Wide Web Consortium (W3C) - Accessibility guides

https://www.w3.org/WAI/fundamentals/accessibility-principles

Standards

Interplay between many components of a page:

- Web content: HTML, images, scripts, etc
- User-agents: Desktop browser, Mobile browser, speech-oriented browser, assistive devices
- Authoring tools: Text editor, Word processors, Compiler

Principle - Preceivable

- Provide text alternatives for non-text content
- Provide captions and other alternatives for multimedia

- Create content that can be presented in different ways, including by assistive technologies, without losing meaning.
- Make it easier for users to see and hear content

Principle - Operation

- Make all functionality available from a keyboard
- Give users enough time to read and use content
- Do not use content that causes seizures or physical reactions
- Help users navigate and find content
- Make it easier to use inputs other than keyboard

Principle - Understandable

- Make the text readable and understandable
- Make the content appear and operate in predictable ways
- Help users avoid and correct mistakes

Principle - Robust

- Maximize compatibility with current and future user tools

Aesthetics

- Visual appearance
- Very important
- Simplicity preferred

Can vary with time

Source:

iOS: A visual history

In what is widely regarded as his greatest presentation ever, Apple's Steve Jobs introduced the iPhone to the world on January 9th, 2007. In the five-plus years since then, the iPhone, iPad, and iPod Touch have literally redefined the entire world of mobile computing.

https://www.theverge.com/2011/12/13/2612736/ios-history-iphone-ipad

Summary

- View - any screen seen by the human or machines
- User-interface and User-interaction guidelines
- Accessibility is a core concept
- Tools for automatic generation, consistent layout

Week 4 Lecture 1

Class	BSCCS2003
Created	@September 27, 2021 1:48 PM
Materials	
Module #	23
Type	Lecture
Week #	4

Persistent Storage

Example: Gradebook

- Students: ID, name, address, ...
- Courses: ID, name, department, year, ...
- Student-Course Relationship: which students are registered for which courses

Gradebook

Name	IDNumber
Sunil Shashi	MAD001
Chetana Anantha	MAD002
Madhur Prakash	MAD003
Nihal Surya	MAD004
Shweta Lalita	MAD005
Raghu Balwinder	MAD006
Gulshan Kuldeep	MAD007
Kishan Shrivatsa	MAD008
Purnima Sunil	MAD009
Nikitha Madhavi	MAD010
Lilavati Prabhakar	MAD011
Rama Yamuna	MAD012

Gradebook

CourseID	Name
EE1001	Introduction to Electrical Engineering
AM1100	Engineering Mechanics
MA1020	Functions of Several Variables
ME1100	Thermodynamics
BR1010	Life Sciences

Spreadsheets

- Arbitrary data organized into rows and columns
- Operations defined on cells or ranges
- Multiple inter-linked sheets within single spreadsheet

Any kind of tabular data - expressed in tables

Relationships

- Students - Course?
- Separate entry with full details - student name, ID, address, course ID, name, department, etc.
  - Redundant
- Separate table "joining" students with courses
  - Only ID specified
  - Relation specified with "Keys"

Aa A	B	# C
MAD001	BT1010	78
MAD002	EE1001	30
MAD005	EE1001	68
MAD009	AM1100	62
MAD012	AM1100	77
MAD007	BT1010	41

Questions

- How should the underlying data should be stored?
  - Can it be made persistent - survive server restart?
- How should the relations be represented?
- Structured ways to represent, manipulate data?

Week 4 Lecture 2

Class	BSCCS2003
Created	@September 27, 2021 2:41 PM
Materials	
Module #	24
Type	Lecture
Week #	4

Mechanisms for Persistent Storage

In-memory data structures

```
names = ['Alice', 'Bob', 'Charlie']
courses = ['Introduction to EE', 'Applied Mech', 'Calculus']
rels = [('Alice', 'Introduction to EE'),
        ('Bob', 'Calculus'),
        ('Alice', 'Calculus'),
        ('Charlie', 'Applied Mech')]
```

- Error prone - easy to make mistakes in entry or referencing
- Does not scale
- Duplicate names?

In-memory data structures: Keys

```
names = {0: 'Alice', 1: 'Bob', 2: 'Charlie'}
courses = {0: 'Introduction to EE', 1: 'Applied Mech', 2: 'Calculus'}
rels = [(0, 0), (1, 2), (0, 2), (2, 1)]
```

- Data entry errors less likely
- Duplicates not a problem - Unique key

Objects

```
class Student:
  id_next = 0 # Class variable
  def __init__(self, name):
    self.name = name
    self.id = Student.id_next
    Student.id_next += 1
```

- Auto-initialize ID to ensure unique
- Functions to set/get values

```
class Student:
  id_next = 0 # Class variable
  def __init__(self, name, hostel):
    self.name = name
    self.id = Student.id_next
    self.hostel = hostel
    Student.id_next += 1
```

- Add a new field to the object easily

But what about persistence?

- In-memory data structures are lost when the program is killed, server is shut down or rebooted (restarted)
- Save to disk? Structured data?
  - Python Pickle and similar modules
  - CSV - Comma Separated Values
  - TSV - Tab Separated Values
- Essentially the same as spreadsheets - limited flexibility

Spreadsheets

- Naturally represent tabular data
- Extension, adding fields are easy
- Separate sheets for relationships

Problems with spreadsheets:

- Lookups, cross-referencing harder than dedicated database
- Stored procedures - limited functionality
- Atomic operations - no clear definition

Relational Databases - SQL

> Shameless plug, I do have notes for DBMS here

- From IBM ~ 1970s
- Data is stored in tabular format:
  - Columns of tables: fields (name, address, department, ...)
  - Rows of tables: individual entries (student1, student2, ...)

Unstructured Databases - NoSQL

- Easily add/change fields
- Arbitrary data
- NoSQL
  - MongoDB
  - CouchDB
  - ...

- Flexible, but potential loss of validation

Week 4 Lecture 3

Class	BSCCS2003
Created	@September 27, 2021 3:39 PM
Materials	
Module #	25
Type	Lecture
Week #	4

Relations and ER diagram

Relationship types

- One to One:
  - One student has one roll number
  - One roll number uniquely identifies one student
  - **Example:** assign unique message-ID to each email in the inbox
- One to Many (Many to One):
  - One stays in only one hostel
  - One hostel has many students
  - **Example:** save emails in folders - one email is in only one folder
- Many to Many:
  - One student can register for many courses
  - One course be taken by many students
  - **Example:** assign labels to emails - one email can have many labels and vice-versa

Diagrams

- Entity-Relationship (ER)
- Unified Modeling Language (UML)
- Class relation

Entity-Relationship Diagram

ER examples

Tool: Draw.io → https://app.diagrams.net

Week 4 Lecture 4

Class	BSCCS2003
Created	@September 27, 2021 4:29 PM
Materials	
Module #	26
Type	Lecture
Week #	4

SQL

Relational Databases - SQL

- From IBM ~ 1970s
- Data is stored in tabular format:
  - Columns of tables: fields (name, address, department, ...)
  - Rows of tables: individual entries (student1, student2, ...)
- Key: unique way of accessing a given row
  - **Primary Key:** important for fast access to large databases
  - **Foreign Key:** connect to a different table - Relationships

Queries

- Retrieve data from a database

eg: `"Find students with name beginning with A"`

`"Find all courses offered in 2021"`

Structured Query Language

- English-like, but structured
- Quite verbose
- Specific mathematical operations:
  - Inner join

- Outer join

Example: Inner join

Name	IDNumber	hostelID
Sunil Shashi	MAD001	1
Chetana Anantha	MAD002	2
Madhur Prakash	MAD003	2
Nihal Surya	MAD004	3
Shweta Lalita	MAD005	2
Raghu Balwinder	MAD006	3
Gulshan Kuldeep	MAD007	1
Kishan Shrivatsa	MAD008	1
Purnima Sunil	MAD009	2
Nikitha Madhavi	MAD010	1
Lilavati Prabhakar	MAD011	3
Rama Yamuna	MAD012	3

ID	Name	Capacity
1	Jamuna	300
2	Ganga	300
3	Brahmaputra	500

Student - Hostel mapping

```
SELECT Students.name, Hostels.name
FROM Students
INNER JOIN Hostels
ON Students.hostelID = Hostels.ID;
```

```
---- OUTPUT ----
Sunil Shashi, Jamuna
Chetana Anantha, Ganga
```

Cartesian product

- N entries in table 1
- M entries in table 2
- M $\times$ N combinations - filter on them

Powerful SQL queries can be constructed

Example: find all the students in Calculus

- Find ID number for the course
- Look up StudentsCourses table to find all entries with this course ID
- Look up Students to find the names of students with these IDs

```
SELECT s.name
FROM Student s
JOIN StudentsCourses sc ON s.IDNumber = sc.studentID
JOIN Courses c ON c.ID = sc.courseID
WHERE c.name = 'Calculus';
```

Summary

- Models - persistent data storage
- Mechanisms:
  - csv
  - spreadsheets
  - SQL
  - NoSQL

- Entities and Relationships
  - Different ways of representing

No details on display, views or what kind of updates permitted

Week 5 Lecture 1

Class	BSCCS2003
Created	@October 4, 2021 12:18 AM
Materials	
Module #	27
Type	Lecture
Week #	5

MVC Origins

Model-View-Controller

- Design pattern: or collection of design patterns
- Originally introduced in the context of GUI design in Smalltalk-80
- Many different variants, interpretations, ...

By RegisFrey - Own work, Public Domain, Wikipedia

From: Trygve Reenskaug

Date: 10 December 1979

MODELS-VIEWS-CONTROLLERS

MODELS

Models represent knowledge. A model could be a single object (rather uninteresting), or it could be some structure of objects. ~~The proposed implementation supports knowledge represented in something resembling *semantic nets* (If I understand Laura correctly)~~

VIEWS

A view is a (visual) representation of its model. It would ordinarily highlight certain attributes of the model and suppress others. It is thus acting as a *presentation filter*.

A view is attached to its model (or model part) and gets the data necessary for the presentation from the model by asking questions. It may also update the model by sending appropriate messages. All these questions and messages have to be in the terminology of the model, the view will therefore have to know the semantics of the attributes of the model it represents. (It may, for example, ask for the model's identifier and expect an instance of Text, it may not assume that the model is of class Text)

CONTROLLERS

A controller is the link between a user and the system. It provides the user with input by arranging for relevant views to present themselves in appropriate places on the screen. It provides means for user output by presenting the user with menus or other means of giving commands and data. The controller receives such user output, translates it into the appropriate messages and pass these messages on .to one or more of the views.

A controller should never supplement the views, it should for example never connect the views of nodes by drawing arrows between them.

Conversely, a view should never know about user input, such as mouse operations and keystrokes. It should always be possible to write a method in a controller that sends messages to views which exactly reproduce any sequence of user commands.

General Concept - Action

- Take action in response to user input
- Communicate with the model, extract the view

Applicability

- Originally designed for GUI applications
- Separation of concerns - model vs views - and connection through controller
- State of interaction maintained as part of overall system memory

Web?

- Server does not maintain the state of the client
- Client is pure front-end to the user
- Some of the analogies break down - hence many variants of MVC

Present Context

- MVC is a good conceptual framework to understand separation of concerns
- Breaks down if applied too rigidly
- The web in general does not have the close knit structure of GUI applications needed for MVC
- Other aspects of static typing and type inference of objects also broken in Python-like languages

Apply basic learnings from MVC, but be prepared to stretch

Week 5 Lecture 2

Class	BSCCS2003
Created	@October 4, 2021 12:41 AM
Materials	
Module #	28
Type	Lecture
Week #	5

Requests and Responses

Example dynamic web page

- View: page
- Links: clickable to select various options
- Clicking a link triggers different behaviors

Request - Response

- Web is based completely on requests and responses
  - Client makes requests
  - Server sends responses
- Basic requests: clicking on link / URL
  - HTTP GET
- More complex requests: form submissions
  - HTTP POST

Constraints?

- Any "page" can be requested
- Assignments, quizzes, lectures, general information

  Are there common threads?

Example: Gradebook

- Students: ID, name, address, ...
- Courses: ID, name, department, year, ...
- StudentCourse Relationship: which students are registered for which courses

	A	B
1	**Name**	**IDNumber**
2	Sunil Shashi	MAD001
3	Chetana Anantha	MAD002
4	Madhur Prakash	MAD003
5	Nihal Surya	MAD004
6	Shweta Lalita	MAD005
7	Raghu Balwinder	MAD006
8	Gulshan Kuldeep	MAD007
9	Kishan Shrivatsa	MAD008
10	Purnima Sunil	MAD009
11	Nikitha Madhavi	MAD010
12	Lilavati Prabhakar	MAD011
13	Rama Yamuna	MAD012

	A	B
1	**CourseID**	**Name**
2	EE1001	Introduction to Electrical Engineering
3	AM1100	Engineering Mechanics
4	MA1020	Functions of Several Variables
5	ME1100	Thermodynamics
6	BT1010	Life Sciences

Common Operations

- Create a new student - add name, roll number, date of birth, ...
- Create a new course
- Assign a student to course
- Enter marks of student / Update marks of a student
- View summaries / charts / histograms
- Archive an old course
- Remove graduated students

Some essential functions can be distilled

Week 5 Lecture 3

Class	BSCCS2003
Created	@October 4, 2021 12:51 AM
Materials	
Module #	29
Type	Lecture
Week #	5

CRUD

Types of operations - Create

- Create a new entry
- Must not already exist
- Check within database to avoid conflicts
- Mention mandatory vs optional fields (name, address, mobile number) ...

Types of operations - Read

- Get a list of students
- Summarize number of students, age distribution, geographic locations
- Plot histograms of marks

Types of operations - Update

- Change of address
- Update marks
- Change start date of course

Type of operations - Delete

- Remove graduated students
- Delete mistaken entries

- Un-enroll students from course

CRUD

Create - Read - Update - Delete

- Originally in context of database operations: nothing to do with the Web
- Reflects cycle of data models
- Databases optimized for various combinations of operations
  - Read-heavy: lots of reading, very little writing or creating
  - Write-heavy: security archive logs

API - Application Programming Interface

- Standardized way to communicate with a server
- Client only needs to know the API - now how the server implements the database, for example
- CRUD is good set of functionality for a basic API
  - Usually considered the first level API to implement a web application
  - Deals only with the data model life cycle - other control aspects possible

Week 5 Lecture 4

Class	BSCCS2003
Created	@October 4, 2021 12:59 AM
Materials	
Module #	30
Type	Lecture
Week #	5

Group Actions by Controller

Actions vs Controllers?

- CRUD etc. are a set of actions
- Other actions:
  - send email
  - update logs
  - send alert on WhatsApp / Telegram
- Can actions be grouped together logically?

  Controller!

Example: Laravel PHP framework

Actions Handled By Resource Controller

Verb	URI	Action	Route Name
GET	`/photos`	index	photos.index
GET	`/photos/create`	create	photos.create
POST	`/photos`	store	photos.store
GET	`/photos/{photo}`	show	photos.show
GET	`/photos/{photo}/edit`	edit	photos.edit
PUT/PATCH	`/photos/{photo}`	update	photos.update
DELETE	`/photos/{photo}`	destroy	photos.destroy

Summary

- Actions: Interaction between view and model
- Controller: Group actions together logically
- API: Complex set of capabilities of server
- Interaction through HTTP requests
- HTTP verbs used to convey meanings

Rules of Thumb

- Should be possible to change views without the model ever knowing
- Should be possible to change the underlying storage of model without views ever knowing
- Controllers / Actions should generally NEVER talk to a database directly

In practice:

- Views and controllers tend to be more closely inter-linked than with Models
    - More about a way of thinking that a specific rule of design

Week 5 Lecture 5

Class	BSCCS2003
Created	@October 4, 2021 1:13 AM
Materials	
Module #	31
Type	Lecture
Week #	5

Routes and Controllers

Web applications

- Client-Server model
- Stateless: Server does not know the present state of the client
  - Must be ready to respond to whatever the client requests without assuming anything about the client
- Requests sent through HTTP protocol
  - Use variants of the GET, POST (verbs) to convey meaning
  - Use URL (Uniform Resource Locator) structure to convey context

Routing: mapping URLs to actions

Python Decorators

- Add extra functionality on top of a function
- "@" - decorators before function name
- Effectively, it is a function of a function
  - Take the new inner function as an argument
  - Return a function that does something before calling the inner function

Basic routing in Flask

```
from flask import Flask
app = Flask(__name__)
```

```
@app.route('/')
def home():
  return 'Hello, World!'
```

HTTP verbs

```
@app.route('/'. methods=['GET'])
def index():
  ...

@ppa.route('/create', methods=['POST'])
def store():
  ...
```

CRUD-like functionality

```
# Assume 'index', 'store' etc. are functions

@app.route('/', method=['GET'])(index)
@app.route('/create', method=['POST'])(store)
@app.route('/<int:user_id>', method=['GET'])(show)
@app.route('/<int:user_id>/edit', method=['POST'])(update)
@app.route('/<int:user_id>', method=['DELETE'])(destroy)
```

Summary

- Flask is not natively MVC
  - But MVC is more a way of thought than a framework
- Simple URL routing
- Helpers to do large scale routing of common functions

Structure the application with separation of concerns in mind

- MVC is just one way to achieve clean design

Week 6 Lecture 1

Class	BSCCS2003
Created	@October 11, 2021 12:36 PM
Materials	
Module #	32
Type	Lecture
Week #	6

API Design

Distributed Software Architecture

- Server - Clients
- Standard "protocols" needed for communication
- Assumptions?
    - Server always on?
    - Server knows what client is doing?
    - Client authentication
    - Network latency?

The Web

- Client - Server may be far apart
- Different network, latencies, quality
- Authentication? Not core part of protocol
- State?
    - Server does not know what state client is in
    - Client cannot be sure what state server is in

Architecture for the Web

- Roy Fielding, PhD thesis 2000 UC Irvine

- "REpresentational State Transfer" - REST
  - Take into account the limitations of the web
  - Provide guidelines or constraints
- Software Architecture Style
  - Not a set of rules

Constraint #1: Client-Server

Constraint #2: Stateless

- Server cannot assume state of the client:
  - Which page are you looking at
  - Is a request coming from an already logged in user just because of the address?
- Client cannot assume the state of the server
  - Did the server reboot since the last request?
  - Is this request being answered by the same server?

Constraint #3: Layered system

Constraint #4: Cacheability

Constraint #5: Uniform interface

- Client and State interact in a uniform and predictable manner
- Server exposes "resources"

Hypertext/media used to convey the available resources and functionality - can be discovered by client through hypertext information from the server

Constraint #6: Code on Demand (Optional)

- Server can extend client functionality
  - JavaScript
  - Java applets (are they still relevant?)

Part of the overall structure - these are not hard rules

Week 6 Lecture 2

Class	BSCCS2003
Created	@October 11, 2021 7:15 PM
Materials	
Module #	33
Type	Lecture
Week #	6

REST

REST

- REpresentational State Transfer

What does that mean?

- State information between client and the server explicitly transferred with every communication

Sequence

- Client accesses Resource identifier from the server
  - Usually URI - superset of URL
  - Typically start from home page of the application
  - No initial state assumed
- Resource Operation specified as part of access
  - If HTTP, then GET,POST, etc.
  - Not fundamentally tied to the protocol
- Server responds with new Resource Identifier
  - New state of system; new links to follow, etc.

State of interaction transferred back and forth

HTTP

- One possible protocol to carry the REST messages

- Use the HTTP verbs to indicate action
- Standardize some types of functionality

- **GET:** Retrieve representation of target resource's state
- **POST:** Enclose data in request: target resource "processes" it
- **PUT:** Create a target resource with data enclosed
- **DELETE:** Delete the target resource

Idempotent Operations

- Repeated application of the operation is not a problem
- **Example:** GET is always safe - read-only operation
- **Example:**
  - **PUT:** will always create the same new resource. If already exists, may give error
  - **DELETE:** can delete only once. May error on repeated deletion, but won't change the data
  - **POST:** May NOT be idempotent
    - **Example:** Add comment to a blog - repeat will cause multiple copies

CRUD

- CRUD: Database operations
- Create, Read, Update, Delete
- Typically a common set of operations needed in most web applications
  - Good candidate of REST based functionality

REST $\neq$ CRUD

But, they do work together quite well

Data Encoding

- **Basic HTML:** for simple reasons
- **XML:** Structured data response
- **JSON:** Simpler form of structured data

Data serialization for transferring complex data types over text based format

JSON

- JavaScript Object Notation
- Nested arrays:
  - Serialize complex data structures like dictionaries, arrays, etc.

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

API data transfer format

- Input to API: text - HTTP
- Output: complex data types - JSON, XML, YAML, etc
  - JSON most commonly used
- Different from internal server representation
- Different from final view presentation

YAML

- **Yet Another Markup Language:** common alternative, especially for documentation and configuration

Week 6 Lecture 3

Class	BSCCS2003
Created	@October 11, 2021 7:17 PM
Materials	
Module #	34
Type	Lecture
Week #	6

REST APIs

- CRUD
- Variants of listing
- Specialized functions
  - Create a new virtual machine
  - Reboot an existing virtual machine
  - Turn off street lights on a given street
- Formal specifications help others to use

Example #1: Wikipedia

- Open API
- Search for pages
- History of a page
- JSON output

```
curl "https://en.wikipedia.org/w/rest.php/v1/search/page?q=earth&limit=1"
```

Response


```
{
  "pages": [
    {
      "id": 9228,
      "key": "Earth",
      "title": "Earth",
      "excerpt": "<span class=\"searchmatch\">Earth</span> is the third planet from the Sun and the only astronomical object known to harbour and support life. About 29.2% of <span class=\"searc
      "description": "Third planet from the Sun in the Solar System",
      "thumbnail": {
        "mimetype": "image/jpeg",
        "size": null,
        "width": 200,
        "height": 200,
        "duration": null,
        "url": "//upload.wikimedia.org/wikipedia/commons/thumb/9/97/The_Earth_seen_from_Apollo_17.jpg/200px-The_Earth_seen_from_Apollo_17.jpg"
      }
    }
  ]
}
```


Documentation

Schema [edit]

`id` required I integer	Page identifier
`key` required I string	Page title in URL-friendly format
`title` required I string	Page title in reading-friendly format
`excerpt` required I string	*For search pages endpoint:* A few lines giving a sample of page content with search terms highlighted with `<span class=\"searchmatch\">` tags *For autocomplete page title endpoint:* Page title in reading-friendly format
`description` required I string	Short summary of the page topic based on the corresponding entry on Wikidata or `null` if no entry exists

Detailed Documentation

Search pages [edit]

Route	`/search/page?q=search terms`	**Content type**	`application/json`
Method	`GET`	**Returns**	`pages` object containing array of search results

Searches wiki page titles and contents for the provided search terms, and returns matching pages.

> **When using this endpoint on your wiki**
> This endpoint uses the search engine configured in the $wgSearchType configuration setting and returns results in the namespaces configured by $wgNamespacesToBeSearchedDefault.

Examples [edit]

curl | Python | PHP | JavaScript

```
# Search English Wikipedia for up to 20 pages containing information about Jupiter
$ curl https://en.wikipedia.org/w/rest.php/v1/search/page?q=jupiter&limit=20
```

Parameters and Response Codes

Parameters [edit]

`q` required I query	Search terms
`limit` optional I query	Maximum number of search results to return, between 1 and 100. Default: 50

Responses [edit]

200	Success: Results found. Returns a `pages` object containing an array of search results.
200	Success: No results found. Returns a `pages` object containing an empty array.
400	Query parameter not set. Add `q` parameter.
400	Invalid limit requested. Set `limit` parameter to between 1 and 100.
500	Search error

Week 6 Lecture 4

Class	BSCCS2003
Created	@October 11, 2021 8:07 PM
Materials	
Module #	35
Type	Lecture
Week #	6

REST APIs

Example #2: CoWin public APIs

- For Co-Win app: Vaccine registration and information
- Unauthenticated APIs:
  - State-wise search, districts, etc.
- Authenticated APIs:
  - Book appointment

https://apisetu.gov.in/public/marketplace/api/cowin#/

General Information

Servers

https://cdn-api.co-vin.in/api - Production Server

Authorize

User Authentication APIs

Metadata APIs

Appointment Availability APIs

Certificate APIs

Schemas

CentersSchema

Example: Availability API

GET /v2/appointment/sessions/public/findByPin Get vaccination sessions by PIN

API to get planned vaccination sessions on a specific date in a given pin.

Parameters Try it out

Name	Description
Accept-Language string *(header)*	The locate code of the preferred language such as en_US. The text data will be returned in the preferred language along with default English text. *Example* : hi_IN hi_IN
pincode * required string *(query)*	110001
date * required string *(query)*	31-03-2021

Testing public API

WARNING: Do NOT overdo this (This is a public API related to health services, overloading this might have some serious implications especially on someone's life)

```
$ curl -X GET "https://cdn-api.co-vin.in/api/v2/appointment/sessions/public/findByPin"
-H "accept: application/json" -H "Accept-Language: en_US"
```

```
{
  "errorCode": "USRRES0001",
  "error": "Input parameter missing"
}
```

```
$ curl -X GET "https://cdn-api.co-vin.in/api/v2/appointment/sessions/public/findByPin?pincode=600020&date=04-08-2021"
-H "accept: application/json" -H "Accept-Language: en_US"
```


```
{"sessions":[
{"center_id":604384,
"name":"Fortis Malar Hospital",
"address":"Fortis Malar HospitalChennai TN.",
"state_name":"Tamil Nadu",
"district_name":"Chennai",
"block_name":"Adyar",
"pincode":600020,
"from":"13:30:00","to":"15:30:00","lat":12,"long":80,"fee_type":"Paid",
"session_id":"d40bd2c9-0f42-4948-b794-e3c31fa7c3cc",
"date":"04-08-2021",
"available_capacity":85,
"available_capacity_dose1":40,
"available_capacity_dose2":45,
"fee":"1250", . . .
```

```
curl -X GET
"https://cdn-api.co-vin.in/api/v2/registration/certificate/public/download?ben
eficiary_reference_id=1234567890123" -H "accept: application/json" -H
"Accept-Language: en_US" -H "User-Agent: Mozilla/5.0"
```

```
Unauthenticated access!
```

Authentication

- Many APIs must be protected:
  - Only meant for specific users
  - Avoid abuse by overloading servers

How to implement this?

Require a "token" that only a valid user can have

- Securely give token only when the user logs in - Google OAuth, Facebook, etc.
- **API key:** One time token that user downloads - can be copied, so potentially less secure unless combined with other methods

Summary

- **API examples:** CoWin, Google Cloud, Twitter, GitHub,... etc.
- Authentication may be enforced or optional on some parts
- Allows 3rd party integrations
- Equivalent of a "remote procedure call" — call a function on a remote system

Week 6 Lecture 5

Class	BSCCS2003
Created	@October 11, 2021 8:32 PM
Materials	
Module #	36
Type	Lecture
Week #	6

OpenAPI

APIs of interest for web apps

- **Purpose:** information hiding — neither server nor the client should know the details of the implementation on the other side
- **Unbreakable contract:** should not change — standardized
  - Versions may update with breaking changes

Documentation

- Highly subjective — some programmers better than others at documenting
- Incomplete — what one programmer finds important may not match others
- Outdated
- Human language specific

Description files

- Machine readable — has very specific structure
- Enable automated processing:
  - Boilerplate code
  - Mock servers
- **Example:** Assembly language is a version of the programming language of computers that is both machine and human readable

    - Structured, so that it can be compiled
- **Versus:** English language specification which needs someone to write code

OpenAPI Specification (OAS)

- **Vendor-neutral** format for **HTTP-based remote API** specification
- Does not aim to describe all possible APIs
- Efficiently describe the common use cases
- Originally developed as Swagger — evolved from Swagger 2.0

Current version: OAS3 - v3.1.0 as of Aug. 2021

Week 6 Lecture 6

Class	BSCCS2003
Created	@October 11, 2021 8:49 PM
Materials	
Module #	37
Type	Lecture
Week #	6

Important Concepts of an API

Concepts

- Describe in YAML (or possibly JSON)
- Specific structure to indicate overall information, paths, schemas, etc

eg:

```
openapi: 3.1.0
info:
  title: A minimal OpenAPI document
  version: 0.0.1
paths: {} # No endpoints defined
```

Endpoints List

Source: https://oai.github.io/Documentation/specification-paths.html

Paths

```
openapi: 3.1.0
info:
  title: Tic Tac Toe
  description: |
    This API allows writing down marks on a Tic Tac Toe board
    and requesting the state of the board or of individual squares
  version: 1.0.0
paths:
  /board:
  ...
```

Operations

```
paths:
  /board:
    get:
      ...
    put:
      ...
```

Operation object

```
paths:
  /board:
    get:
      summary: Get the whole board
      description: Retrieves the current state of the board and the winner.
      parameters:
        ...
      responses:
        ...
```

Responses

```
paths:
  /board:
    get:
      responses:
        "200":
          ...
        "404":
          ...
```

Response Objects

```
paths:
  /board:
    get:
      responses:
        "200":
          description: Everything went fine.
```

```
          content:
            ...
```

Content Specification

```
content:
  application/json:
    ...
  text/html:
    ...
  text/*:
    ...
```

Schema

```
content:
  application/json:
    schema:
      type: integer
      minimum: 1
      maximum: 100
```

Complex schema

```
content:
  application/json:
    schema:
      type: object
      properties:
        productName:
          type: string
        productPrice:
          type: number
```

Parameters

```
paths:
  /users/{id}:
    get:
      parameters:
      - name: id
        in: path
        required: true
```

Request body

```
requestBody:
  content:
    application/json:
      schema:
        type: integer
        minimum: 1
        maximum: 100
```

Best practices

- Design-first v/s Code-first
  - Always prefer the design-first
- Single source of truth
  - The structure of the code should be derived from the OAS - or -
  - Spec should be derived from the code
  - Minimize chances of code and documentation diverging
- Source code version control
- OpenAPI is ... Open — public documentation better to identify the problems

- Automated tools, editors — make use of them

Week 7 Lecture 1

Class	BSCCS2003
Created	@October 18, 2021 10:50 AM
Materials	
Module #	38
Type	Lecture
Week #	7

Memory Hierarchy

Types of Storage Elements

- On-chip registers: 10s-100s of bytes
- SRAM (Cache): 0.1-1MB # Static RAM
- DRAM (Memory/RAM of a PC): 0.1-16GB
- Solid-State Disk (SSD) - Flash: 1-128GB
- Magnetic Disk (HDD-Hard Disk Drive): 0.1-10TB

Storage Parameters

- **Latency:** Time to read the first value from a storage location (lower is better)
  - Register < SRAM < DRAM < SSD < HDD
  - So, the turnaround time when trying to read something from a ...
    - Register: in order of nanoseconds
    - SRAM: tens to hundreds of nanoseconds
    - DRAM: several microseconds
    - SSDs: hundreds of microseconds
    - HDDS: order of milliseconds
- **Throughput:** number of bytes/second that can be read (higher is better)
  - DRAM > SSD > HDD (registers, SRAM have limited capacity)

- **Density:** Number of bits stored per unit area/cost (higher is better)
  - Volume manufacture important
  - HDD > SSD > DRAM > SRAM > Registers

Computer Organization

- CPU has as many registers as possible
- Backed by L1, L2, L3 cache (SRAM)
- Backed by several Gigabytes (GBs) of DRAM working memory
- Backed by SSD for high throughput
- Backed by HDD for high capacity
- Backed by long-term storage, backup

Cold Storage

- Backups & archives:
  - Huge amounts of data
  - Not read very often
  - Can tolerate high read latency
- Amazon Glacier, Google, Azure Cold/Archive storage classes
- High latency of retrieval → up to 48 hours
- Very high durability
- Very low cost

Impact on Application Development

- Plan the storage needs based on the application growth
- Speed of app determined by types of data stored, how stored
- Some data stores are more efficient for some types of read/write operations

Developer must be aware of the choices and what kind of DB to choose for a given application

Week 7 Lecture 2

Class	BSCCS2003
Created	@October 18, 2021 2:13 PM
Materials	
Module #	39
Type	Lecture
Week #	7

Data Search

O() notation - The Big O notation

- Used in study of algorithmic complexity
- Rough approximation: "order of magnitude", "approximately", etc.
- Main concepts:
    - $O(1)$ - constant time, independent of input size: Excellent!
    - $O(log\ N)$ - logarithmic in input size - grows very slowly with input: Very good
    - $O(N)$ - linear in input size - often the baseline - Would like to do better
    - $O(N^k)$ - polynomial (quadratic, cubic, etc.) - Not good as the input size grows
    - $O(k^N)$ - exponential - **VERY BAD**: Won't even work for reasonably small inputs

Searching for an element in the memory

Unsorted data in a linked list (or an array)

- Start from the beginning
- Proceed stepwise, comparing each element
- Stop if found and return the LOCATION
- If end-of-list, return NOTFOUND

$O(N)$

Sorted data in an array

- Look at the middle element in an array:
  - Greater than the target?
    - Search in the lower half
  - Lesser than the target?
    - Search in the upper half
- Switch focus to the new array: Half the original size
  - Repeat

$O(log\ N) : Binary\ Search$

Problems with arrays

- Size must be fixed ahead of time
- Adding new entries requires re-sizing - can try oversize, but eventually ...
- Maintaining sorted order $O(N)$:
  - Find the location to insert
  - Move all further elements by 1 to create a gap
  - Insert
- Deleting
  - Find the location, delete
  - Move all the entries down by 1 step

Alternatives

- Binary Search Tree
  - Maintaining sorted order is easier: growth of the tree
- Self-Balancing
  - BST can easily tilt to one side and grow downwards
  - Red-black, AVL, B-tree ... more complex, but still reasonable
- Hash tables
  - Compute an index for an element: $O(1)$
  - Hope the index for each element is unique!
    - Difficult but is doable in many cases

Week 7 Lecture 3

Class	BSCCS2003
Created	@October 18, 2021 2:42 PM
Materials	
Module #	40
Type	Lecture
Week #	7

Database Search

Databases (Tabular)

- Tabular with many columns
- Want to search quickly on some columns
- Maintain "INDEX" of columns to search on
    - Store a sorted version of column
    - Needs column to be "comparable": integer, short string, date/time, etc.
        - Long text fields are not good for index
        - Binary data not good as well

Example: MySQL

MySQL 8.0 Reference Manual / ... / Comparison of B-Tree and Hash Indexes

version 8.0

8.3.9 Comparison of B-Tree and Hash Indexes

Understanding the B-tree and hash data structures can help predict how different queries perform on different storage engines that use these data structures in their indexes, particularly for the `MEMORY` storage engine that lets you choose B-tree or hash indexes.

- B-Tree Index Characteristics
- Hash Index Characteristics

Source: https://dev.mysql.com/doc/refman/8.0/en/index-btree-hash.html

Index-friendly Query

```
SELECT * FROM tbl_name WHERE key_col LIKE 'Patrick%';
SELECT * FROM tbl_name WHERE key_col LIKE 'Pat%_ck%';
```

In the first statement, only rows with `'Patrick' <= key_col < 'Patricl'` are considered

In the second statement, only rows with `'Pat' <= key_col < 'Pau'` are considered

Index-unfriendly Query

```
SELECT * FROM tbl_name WHERE key_col LIKE '%Patrick%';
SELECT * FROM tbl_name WHERE key_col LIKE other_col;
```

In the first statement, the `LIKE` value begins with a wildcard character

In the second statement, the `LIKE` value is not a constant

Multi-column index

- (index_1, index_2, index_3): compound index on 3 columns:
  - first sorted on index_1, then on index_2, then on index_3
  - all values with same index_1 will be sorted on index_2
  - all values with same index_1 and index_2 will be sorted on index_3
- Example: (date-of-birth, city-of-birth, name)
  - can query for all the people born on the same date in the same city with the same name easily

Multi-index friendly

```
... WHERE index_part1=1 AND index_part2=2 AND other_column=3

/* index = 1 OR index = 2 */
... WHERE index=1 OR A=10 AND index=2

/* optimized like "index_part1='hello'" */
... WHERE index_part1='hello' AND index_part3=5

/* use index on index1 but not on index2 or index3 */
... WHERE index1=1 AND index2=2 OR index1=3 AND index3=3;
```

Multi-index unfriendly

```
/* index part1 is not used */
... WHERE index_part2=1 AND index_part3=2

/* Index not used in both parts of the WHERE clause */
... WHERE index=1 OR A=10

/* No index spans all rows */
... WHERE index_part1=1 OR index_part2=10
```

Hash-index

- Only used in in-memory tables
- Only for equality comparisons - cannot handle ranges
- Does not help with "ORDER BY"
- Partial key prefix cannot be used
- But VERY fast where applicable

Query Optimization

Database specific:

- https://dev.mysql.com/doc/refman/8.0/en/index-btree-hash.html
- https://www.sqlite.org/optoverview.html
- https://www.postgresql.org/docs/12/geqo.html

Summary

- Setting up queries properly impacts application performance
- Building proper indexes is crucial to good search
- Compound indexes, multiple indexes etc. possible
  - Too many can be a waste of space
- Make use of structure in data to organize it properly

Week 7 Lecture 4

Class	BSCCS2003
Created	@October 18, 2021 3:42 PM
Materials	
Module #	41
Type	Lecture
Week #	7

SQL v/s. NoSQL

SQL

- Structured **Query** Language
  - Used to query DB that have structure
  - Could also be used for CSV files, spreadsheets, etc.
- Closely tied to RDBMS - Relational DBs
  - Colums/Fields
  - Tables of data hold relationships
  - All entries in a table **must** have same set of columns
- Tabular DBs
  - Efficient indexing possible - use specified columns
  - Storage efficiency: prior knowledge of data size

Problems with tabular DBs

- Structure (good? bad?)
- All rows in a table must have same set of columns

Example:

- Student - hostel ⇒ mess
- Student - day-scholar ⇒ gate pass for vehicle

- Table? Column for mess, column for gate pass?

Alternate ways to store: Document DBs

- Free-form (unstructured) documents
  - Typically JSON encoded
  - Still structured, but each document has own structure
- **Examples:**
  - MongoDB
  - Amazon DocumentDB

```
[
  {
    "year": 2013,
    "title": "Turn It Down, Or Else!",
    "info": {
      "directors": ["Alice Smith", "Bob Jones"],
      "release_date": "2013-01-18T00:00:00Z",
      "rating": 6.2,
      "genres": ["Comedy", "Drama"],
      "image_url": "http://ia.media-imdb.com/images/N/09ERWAU7FS797AJ7LU8HN",
      "plot": "A rock band plays their music at high volumes, annoying the neighbours",
      "actors": ["David Matthewman", "Jonathan G. Neff"]
    }
  },
  {
    "year": 2015,
    "title": "The Big New Movie",
    "info": {
      "plot": "Nothing happens at all",
      "rating": 0
    }
  }
]
```

Alternate ways to store: Key-Value

- Python dictionary, C++ OrderedMap, etc. → dictionary/hash table
- Map a key to a value
- Store using search trees or hash tables
- Very efficient key lookup, not good for range type queries
- **Example:**
  - Redis
  - BerkeleyDB
  - memcached
- Often used alongside other DBs for "in-memory" fast queries

Alternate ways to store: Column store

- Traditional relational DBs store all values of a row together on the disk
  - Retrieving all entries of a given row is very fast
- Instead store all entries in a colum together
  - Retrieve all values of a given attribute (age, place of birth, ...) very fast
- **Example:**
  - Cassandra
  - HBase

Alternate ways to store: Graphs

- Friend-of-a-friend, social networks, maps: graph oriented relationships
- Different degrees (number of out-going edges), weights of edges, nodes, etc.

- Path-finding more important than just search
  - Connections, knowledge discovery
- Examples:
  - Neo4j
  - Amazon Neptune

Alternate ways to store: Time Series DBs

- Very application specific: Store some metric or values as a function of time
- Used for log analysis, performance analysis, monitoring
- Queries:
  - How many hits between T1 and T2?
  - Avg. number of requests per second?
  - Country from where maximum requests came in the past 7 days?
- Typical RDBMS completely unsuitable - same for most alternatives
- **Example:**
  - RRDTool
  - InfluxDB
  - Prometheus
  - Elasticsearch
  - grafana

NoSQL

- Started out as an "alternative" to SQL
- But SQL is just a query language - can be adapted for any kind of query, including from a document store or graph
- "Not-only-SQL"
- Additional query patterns for other types of data stores

ACID

- **Transaction:** core principle of DB
- **ACID:**
  - Atomic
  - Consistent
  - Isolated
  - Durable
- Many NoSQL DBs sacrifice some part of ACID
  - **Example:** Eventual consistency instead of consistency, for performance
- But, there can be ACID compliant NoSQL DB as well

Why not ACID?

- Consistency hard to meet: especially when scaling/distributing
- Eventual consistency easier to meet
- Example:
  - A (located in India) and B (located in USA) both add C as a friend on Facebook
  - Order of adding does not matter
  - Temporarily seeing C in A's list but not B, or B's list but not A - not a catastrophe

- Financial transactions **absolutely require ACID**
  - Consistency is paramount: even a split second of inconsistent data can cause problems

A word on storage

- In-memory:
  - Fast
  - Does not scale across machines
- Disk
  - Different data structures, organization needed

Week 7 Lecture 5

Class	BSCCS2003
Created	@October 18, 2021 6:50 PM
Materials	
Module #	42
Type	Lecture
Week #	7

Scaling

Replication and Redundancy

- Redundancy:
  - Multiple copies of same data
  - Often used in connection with backups - even if one fails, others survive
  - One copy is still the master
- Replication:
  - Usually in context of performance
  - May not be for purpose of backup
  - Multiple sources of same data - less chance of server overload
  - Live replication requires careful design

BASE vs ACID

- "Basically Available", "Soft state", "Eventually consistent"
  - Winner of worst acronym award, lol
- Eventual consistency instead of Consistency
  - Replicas can take time to reach a consistent state
- Stress on high availability of data

Replication in traditional DBs

- RDBMS replication is possible
- Usually server cluster in the same data center
  - Load balancing
- Geographically distributed replication harder
  - Latency constraints for consistency

Scale-up vs Scale-out

- **Scale-up:** Traditional approach
  - Larger machine
  - More RAM
  - Faster network, processor
  - Requires machine restart with each scale change
- **Scale-out:**
  - Multiple servers
  - Harder to enforce consistency etc. → better suited for NoSQL/non-ACID
  - Better suited to cloud model: Google, AWS etc. to provide automatic scale-out, cannot do auto scale-up

Application Specific

- Financial transactions:
  - Cannot afford even slightest inconsistency
  - Only scale-up possible
- Typical web application
  - Social networks, media: Eventual consistency is OK
  - e-Commerce: only the financial part needs to go to ACID

Week 7 Lecture 6

Class	BSCCS2003
Created	@October 18, 2021 7:05 PM
Materials	
Module #	43
Type	Lecture
Week #	7

Security of Databases

SQL in context of an application

- Non-MVC app: can have direct SQL queries anywhere
- MVC: only in controller, but any controller can trigger a DB query

So, what is dangerous about queries?

Typical HTML form

```
<form>
  Username:
  <input type="text" name="name">
  <br />
  Password:
  <input type="password" name="password">
  <br />
</form>
```

Code

```
name = form.request['name']
pswd = form.request['password']

sql = 'SELECT * FROM Users WHERE Name = "' + name + '" AND Pass = "' + pswd + '"'
```

Example Input vs SQL

Username: *abcd*

Password: *pass*

`SELECT * FROM Users WHERE Name = "abcd" AND Pass = "pass"`

Username: *" or ""="*

Password: *" or ""=""*

`SELECT * FROM Users WHERE Name = "" or "" AND Pass = "" or ""`

→ *The above query checks for, basically, nothing and returns everything from the table* `Users`

`sql = "SELECT * FROM Users WHERE Name = " + name`

Input:

`a; DROP TABLE Users;`

Query:

`SELECT * FROM Users WHERE Name = a; DROP TABLE Users;`

Problem

- Parameters from the HTML taken without validation
- Validation:
  - Are they valid text data (no special characters, other symbols)
  - No punctuation or other invalid input
  - Are they the right kind of input (text numbers, email, date)?
- Validation **MUST** be done before the DB query - even if you have validation in the HTML or JavaScript - not good enough
  - Direct HTTP requests can be made with junk data

Web Application Security

- SQL injection
  - Use known frameworks, best practices, validation
- Buffer overflows, input overflows
  - Length of inputs, queries
- Server level issues - protocol implementation?
  - Use known servers with good track record of security
  - Update all patches
- Possible outcomes:
  - loss of data - deletion
  - exposure of data - sensitive information leak
  - manipulation of data - change

HTTPS

- Secure sockets: Secure communication between client and server
- Server certificate:
  - based on DNS: Has been verified by some trusted 3rd party
  - difficult to spoof
  - based on Mathematical properties - ensure very low probability of mistakes match

- However:
  - Only secures the link for data transfer - does not perform validation or safety checks
  - Negative impact on "caching" of resources like static files
  - Some overhead on performance

Summary

- Internet and Web security are complex: Enough for a course in themselves
- Generally recommended to use known frameworks with trusted track records
- Code audits
- Patch updates on OS, server, network stack, etc. essential

App developers should be very careful of their code, but also aware of problems on other levels of stack.

Week 8 Lecture 1

Class	BSCCS2003
Created	@October 24, 2021 12:43 AM
Materials	
Module #	44
Type	Lecture
Week #	8

Application Frontends

What is an application frontend?

- User-facing interface
  - General GUI application on the desktop
  - Browser based client
  - Custom embedded interface
- Device/OS specific controls and interfaces
- Web browser standardization
  - Common conventions among multiple browsers on how to render, what to render
- Browser vs Native
  - Look and feel
  - APIs, interfaces and interactions

Web Applications

- Browser based → HTML + CSS + JavaScript
  - HTML → What to show
  - CSS → How to show it
  - JavaScript → Bonus interaction (not core UI but essential for dynamic experience)
- Frontend mechanisms?

    - How to generate the HTML, CSS and JS?
    - Functional re-use, common frameworks
    - Server/Client load implications
    - Security implications

Fully static pages

- All (or most) pages on the site are statically generated
    - Compiled ahead of time
    - Not generated at run-time
- Excellent for high performance
    - Server just picks up the file and delivers
- How do you adapt to the runtime conditions?
    - User login, user specific information, time of the day
    - JavaScript can help a lot
- Increasingly popular → Static site generators
    - Jekyll, Hugo, Next.js, Gatsby
    - JavaScript allows very interesting variants

Run-time HTML generation

- Traditional CGI/WSGI based apps
    - Python (Flask, Django, ...), Ruby (Ruby on Rails)
    - PHPs core concept → Server-side run-time generation of HTML
    - WordPress, Drupal, Joomla → traditional CMS applications
- Great flexibility
    - Common layouts, adaptation and theming easy
    - Run-time changes, user login, time of the day, etc
- Server load
    - Every page has to be generated dynamically
    - May involve databases hits
    - Cost
    - SPeed
- Caching and other technologies can help, but complex

Client Load

- Typical Web browser
    - issue requests, wait for responses
    - render HTML
    - wait for the user input → most time spent waiting here
- Why not let client do more?
    - Allows more fancy interactions
- Client-side scripting
    - JavaScript de-factor standard
    - Component frameworks allows reuse, complex interactions
    - Server side JavaScript Node.js

Tradeoffs

- Server-side rendering
  - Very flexible
  - May be easier to develop
  - Less security issues on client
  - Load on the server
  - More security issues on the server
- Static
  - Cache-friendly
  - Very fast
  - Interaction difficult/impossible?
  - Compilation phase → small changes require compilation
- Client-side
  - Can combine well with static pages
  - Less load on the server but still dynamic
  - More resources needed on client
  - Potential security issues, data leakage

Estimating performance

https://server.guy.com/comparison/apache-vs-nginx/

- Static pages:
  - Apache ~ 10,000 req/s - 512 parallel requests
  - Nginx ~ 20,000 req/s - 512 parallel requests
- Dynamic (call out to PHP - limited by page rendering in PHP)
  - Both ~ 100 req/s @ 16 parallel
- Dynamic occupies more resources for longer → harder to scale
- Severe impact on the server

Week 8 Lecture 2

Class	BSCCS2003
Created	@October 24, 2021 10:58 AM
Materials	
Module #	45
Type	Lecture
Week #	8

Asynchronous Updates

Original Web

- Client send request → Server responds → Client displays
- For any update of the page
  - A new request sent from the client to the server
  - Server has to respond with complete page, HTML, styling, etc
  - Client has to render the page again from scratch
- Potential issues
  - Server load → lots of redundant data to be sent each time
  - Server-rendering → More work
  - Slow updates → Load full page, re-render

Asynchronous Updates

- Update only part of the page
  - Load extra data in the background after the main page has been loaded and rendered
- Quick response on the main page → Better user experience
- Request for update can ask for just minimal data to refresh part of the page
  - Example → Show user a form to select animal
  - Request data about animal alone from the server → No need for HTML or other styling

    - Refresh only one `<div>` in the page with text about the animal
- Originally seen as AJAX, now many variants

Core idea → Refresh part of the document based on asynchronous (background) queries to the server

DOM

Document Object Model

- Programming interface for web documents
- What is a web-page?
    - HTML source? Rendered image?
- DOM is an abstract model (tree structure) of the document
- Object-oriented allows manipulation like known objects
- Tightly coupled with JavaScript in most cases
    - Can also be manipulated from other languages (Python has XML DOM interface for example)

Example Usage

https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

```
const paragraphs = document.querySelectorAll("p");
// paragraphs[0] is the first <p> element
// paragraphs[1] is the second <p> element, etc.
alert(paragraphs[0].nodeName);
```

Manipulating the DOM

```
<html>
  <head>
    <script>
      // Run this function when the document is loaded
      window.onload = function() {
        // Create a couple of elements in an otherwise empty HTML page
        const heading = document.createElement("h1");
        const heading_text = document.createTextNode("Big Head!");
        heading.appendChild(heading_text);
        document.body.appendChild(heading);
      }
    </script>
  </head>
  <body>
  </body>
</html>
```

Component building

- DOM is manipulatable through programs
- Bring concepts of programming into DOM manipulation:
    - Objects
    - Composition of objects, inheritance
    - Loops, iterators, programmable placement
- Lot more flexibility in front-end
- Lot more complexity in the front-end

Summary

- Asynchronous updates opened up front-end development
- Many new frameworks, technologies

Week 8 Lecture 3

Class	BSCCS2003
Created	@October 24, 2021 11:18 AM
Materials	
Module #	46
Type	Lecture
Week #	8

Browser/Client Operations

Minimal requirements

- Render HTML
- Cookie interaction → accept, return cookies from the server to allow sessions
- Text-mode browsers (lynx, elinks, etc) may not do anything more

Text-mode and Accessibility

- Browse from the command-line → only text displayed
- No images, limited styling

Accessibility:

- Page should not rely on colours or font size/styles to convey meaning
- W3C accessibility guidelines

Page styling

- Cascading Style Sheets (CSS) most popular now
- Difficult in text, accessible browsers
  - But has many features to help even with those!
- Proper separation of HTML and styling gives best freedom to browser, user

Interactivity

- Some form of client-side programming needed
- JavaScript most popular → de-facto standard
- Can interact with basic HTML elements (buttons, links, forms, etc)
- Can also be used independently to create more complex forms

Performance of JS depends on browser and choice of scripting engine

JavaScript engines

- Chrome/Chromium/Brave/Edge → V8
- Firefox → SpiderMonkey
- Safari, older versions of IE use their own

Impact

- Performance → V8 generally best at present
- JS standardization means differences in engines less important

Client load

- JS engines also use client CPU power
  - Complex page layouts require computation
- Can also use GPU → Extensive graphics support
  - Images
  - Video
- Potential to load CPU
  - Wasteful → Block useful computations
  - Energy drain! → https://www.websitecarbon.com

Machine clients

- Client may not always be a human
- Machine end-points → Typically access APIs
- Embedded devices → Post sensor information to data collection sites
  - Especially for monitoring, time series analysis, etc
- Typically cannot handle JS → only HTTP endpoints

Alternative scripting languages

Python inside a browser? Brython

https://brython.info

Problems with alternatives

- JS already included with browsers → why alternative?
- Usual approach → **transpilation**
  - Translation - Compilation
- Some older browsers tried directly including custom languages → Now mostly all convert

WASM

- WebAssembly
- Binary instruction format
- Targets a stack based virtual machine (similar to Java)
- Sandboxed with controlled access to APIs
- "Executable format for the Web"

- Handles high performance execution → can translate graphics to OpenGL, etc

Emscripten

- Compiler framework → compile C or C++ (or any other language that can target LLVM) to WebAssembly
- Potential for creating high performance code that runs inside the browser
- Limited usage so far

https://emscripten.org/index.html

Native mode

- File system
- Phone, SMS
- Camera object detection
- Web payments

Functionality can be exposed through suitable APIs → requires platform support

- Adds additional security concerns

Week 8 Lecture 4

Class	BSCCS2003
Created	@October 24, 2021 11:38 AM
Materials	
Module #	47
Type	Lecture
Week #	8

Client side computations and Security Implications

Validation

- Server-side validation essential
  - No guarantees that request actually came from a given front-end
- But some client-side validation can reduce hits on the server
- Example → E-mail, date range, sanitization (no invalid characters) etc.
- Similar validation to backend, but now in front-end script
- Extra work, but better user experience

https://developer.mozilla.org/en-US/docs/Learn/Forms/Form_validation

Inbuilt HTML5 form controls

- Partial validation added by HTML5 standard
- `required` → mandatory field
- `minlength, maxlength` → for text fields
- `min, max` → for numeric values
- `type` → for some specific predefined types
- `pattern` → regular expression pattern match

Important → older browsers may not all these features

Is backward compatibility essential for your app?

JavaScript validation

Constraint Validation API

https://developer.mozilla.org/en-US/docs/Web/API/Constraint_validation

- Supported by most browsers
- Much more complex validation possible

Remember → not a substitute for server-side validation

```
<form>
  <label for="mail">I would like you to provide me with an e-mail address:</label>
  <input type="email" id="mail" name="mail">
  <button>Submit</button>
</form>
```

```
const email = document.getElementById("mail");

email.addEventListener("input", function (event) {
  if (email.validity.typeMismatch) {
    email.setCustomValidity("I am expecting an e-mail address");
  } else {
    email.setCustomValidity("");
  }
});
```

Captcha

- **Problem** → scripts that try to automate web-pages
- Can generate large number of reports in short time - server load
- Railway Tatkal, CoWin appointments, etc

Solution

- Prove that you are a human

- Limited number of clicks per possible unit time
- Script on the page will generate some token - server will reject requests without that token

Crypto-mining?

- JavaScript is a "complete" language
- Can implement any computation with JavaScript
- Modern JS engines are very powerful and fast
  - Can even access system graphics processor (GPU) for rendering, etc.
- Run a simple page that loads and runs a JS script

- Script will send results back to the server through async calls
- Client may not even be aware

Security Implications

Sandboxing

- Should JS be run automatically on every page
  - Yes → provides significant capabilites
  - No → what if the page tries to load local files and send them to the server?
- Sandbox → secure area that JS engine has access to
- Cannot access files, network resources, local storage
- Similar to Virtual Machines, but at a higher level (JS interpreter)

Overload and DoS

- DoS → Denial of Service
- Run a script that takes over the browser engine and runs at high load
- Difficult to even navigate away from the page, or close the page
- Potential exploit bugs in the browser
- Server attack:
  - Replace some popular JS file with a bad version
  - Will be loaded by a large number of sites, users → can write script to access some other site
  - Target site will be hit by a large number of requests from several sources, very difficult to control

Access to native resources

- Can JS be used to write fully native applications?
- Access to resources like local storage, sensors (tilt, magneto, camera)
- Can be permitted explicitly through the browser

Can also be compiled directly to native resources

- Reduce browser overheads
- Smoother interaction with the system

Summary

- Frontend experience determined by browser capabilities
  - Basic HTML + CSS rendering → styling
  - JavaScript/client-side scripting for user interaction, smoother intergration
- Native clients possible
- Potentially serious security implications
- Always validate data again at the server, do not assume client validation (Never trust the client)
  - HTTP is stateless → server cannot assume client was in a particular state

Week 9 Lecture 1

Class	BSCCS2003
Created	@November 1, 2021 10:55 AM
Materials	
Module #	48
Type	Lecture
Week #	9

Access Control

What is Access Control?

- Access → Being able to read/write/modify information
- Not all parts of an application is for public access
  - Personal, Financial Company, Grades
- Types of access →
  - read-only
  - read-write (CRUD)
  - modify but not create
  - ...

Examples

- Linux files →
  - owner, group → access your own files, cannot modify (or even read?) others
  - can be changed by the owner
  - `"root"` or `"admin"` or `"superuser"` has the power to change permissions
- Email →
  - you can read your own email
  - can forward an email to someone else → this is also access!

- E-commerce login →
  - shopping cart etc. visible only to the user
  - financial information (credit card, etc.) must be secure

Discretionary vs Mandatory

- **Discretionary**
  - you have control over who you share with
  - forwarding emails, changing file access modes etc. possible
- **Mandatory**
  - decisions made by centralized management → users cannot even share information without permission
  - Typically only in military or high security scenarios

Role based access control

- Access associated with "role" instead of "username"
- Example →
  - Head of the department has access to student records
  - What happens when HoD changes?
- Single user can have multiple roles
  - HoD, Teacher, Cultural advisor, sports club member, ...
- Hierarchies, Groups
  - HoD > Teacher > Student
  - HpD vs sports club member? → No hierarchy here

Attribute based access control

- Attribute
  - time of the day
  - some attribute of user (citizenship, age, ...)
- Can add extra capability over role-based

Policies vs Permissions

- Permissions
  - Static rules usually based on simple checks (does the user belong to a group)?
- Policies
  - More complex conditions possible
  - Combine multiple policies
  - Example →
    - Bank employees can view ledger entries
    - Ledger access only after 8AM on working days

Principle of least privilege

- Entity should have minimal access required to do the job
- Example → Linux file system
  - users can read system libraries but not write
  - some files like `/etc/shadow` are not even readable
  - you can install Python to local files using `"venv"` but not to system path
- Benefits

    - better security → fewer people with access to sensitive files
    - better stability → user cannot accidentally delete important files
    - ease of deployment → can create template filesystems to copy

Privilege Escalation

- Change user or gain an attribute
    - `"sudo"` or `"su"`
- Usually, combined with explicit logging and extra safety measures
- Recommended
    - do **not** `sudo` unless absolutely necessary
    - never operate as `root` in a Linux/Unix environment unless absolutely necessary

Context: Web apps

- Admin dashboards, user access, etc.
- Gradebook example:
    - only admin should be able to add/delete/modify
    - users should have read permissions only to their own data

Enforcing

- Hardware level
    - Security key, hardware token for access, locked doors, etc.
- Operating Systems
    - filesystem access, memory segmentation
- Application level
    - DB server can restrict access to specific databases
- Web application
    - Controllers enforce restrictions
    - Decorators in Python used in frameworks like Flask

Week 9 Lecture 2

Class	BSCCS2003
Created	@November 1, 2021 12:03 PM
Materials	
Module #	49
Type	Lecture
Week #	9

Security Mechanisms

Types of security checks

- Obscurity (generally a very bad idea)
  - application listens on non-standard port known only to specific people
- Address →
  - where are you coming from? host based access/deny controls
- Login →
  - username/password provided to each person needing access
- Tokens →
  - access tokens that are difficult/impossible to duplicate
  - can be used for machine-to-machine authentication without passwords

HTTP authentication

Basic HTTP auth →

- Enforced by the server
- Server returns `"401/Unauthorized"` code to the client
- Contrast with →
  - 404 → not found
  - 403 → forbidden (no option to authenticate)

- Client must respond with access token as an extra "Header" in the next request

Source: https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication

Problems with HTTP Basic Auth

- Username, Password effectively sent as plain text (base64 encoding)
  - Some minimal security if HTTPS is used (wiretap is difficult)
- Password will be seen in cleartext at the server
  - Should not be needed → better mechanisms possible
- No standard process for `"logout"`

Digest Authentication

- Message digest → cryptographic function
  - eg. MD5, SHA1, SHA256 etc.
- One-way function
  - $f(A) = B$
  - Easy to compute B given A
  - Very difficult (nearly impossible) to compute A given B
- Can define such one-way functions on strings
  - String → Binary number

HTTP Digest Authentication

- Server provides a "nonce" (Number used once) to prevent snooping
- Client must create a secret value including nonce
- Example
  - HA1 = MD5 (username:realm:password)
  - HA2 = MD5 (method:URI)
  - response = MD5 (HA1:nonce:HA2)
- Server and client know all the parameters above, so both will compute the same
- Any third party snooping will see only final response
  - cannot extract original values (username, password, nonce, etc)
  - nonce only used once to prevent replay

Client certificates

- Cryptographically secure credentials provided to each client
- Client does not handshake with the server to exchange information, prove knowledge
- Keep cert secure on client end
  - Impossible to reverse and find the key

Form Input

- Username, Password entered into the form
- Transmitted over link to server
  - link must be kept secure (HTTPS)
- GET requests
  - URL encoded data → very insecure, open to spoofing
- POST requests
  - form multipart data → slightly more secure
  - still needs secure link to avoid data leakage

Request level security

- One TCP connection
  - One security check may be sufficient
  - other network level issues to consider for TCP security
- Without connection `KeepAlive` →
  - each request needs new TCP connection
  - each request needs new authentication

Cookies

- Server checks some client credentials, then "sets a cookie"
- Header
  - `Set-Cookie: <cookie-name>=<cookie-value>; Domain=<domain-value>; Secure; HttpOnly`
- Client must send back the cookie with each request
- Server maintains "sessions" for clients
  - Remember cookies
  - Can set timeouts
  - Delete cookie records to "logout"
- Client
  - must send cookies with each request

API security

- Cookies etc. requires interactive use (browser)
- Basic auth pop-up window

APIs →

- Typically accessed by the machine client or other applications
- Command-line etc. possible
- Use "token" or "API key" for access
  - subject to same restrictions → HTTPS, not part of the URL, etc

Week 9 Lecture 3

Class	BSCCS2003
Created	@November 1, 2021 6:40 PM
Materials	
Module #	50
Type	Lecture
Week #	9

Sessions

Session management

- Client sends multiple requests to server
- Save some "state" information
  - logged in
  - choice of background colour
  - ...
- Server customizes responses based on client session information

Storage →

- Client-side session → completely stored in cookie
- Server-side session → stored on server, looked up from cookie

Cookies

- Set by server with Set-Cookie header
- Must be returned by the client with each request
- Can be used to store information →
  - theme, background colour, font size → simple no security issues
  - user permissions, username → can also be set in cookie
    - must not be possible to alter

Example → Flask

```
from flask import session

# Set the session key to some random bytes. Keep this secret!
app.secret_key = b'_5#y2L"F4Q8z\n\xec]/'

@app.route('/')
def index():
  if 'username' in session:
    return f'Logged in as {session["username"]}'
  return 'You are not logged in'
```

```
@app.route('/login', methods=['GET', 'POST'])
def login():
  if request.method == 'POST':
    session['username'] = request.form['username']
    return redirect(url_for('index'))
  return '''
    <form method="post">
      <p><input type="text" name="username"></p>
      <p><input type="submit" value="Login"></p>
    </form>
  '''

@app.route('/logout')
def logout():
  # remove the username from the session if it's there
  session.pop('username', None)
  return redirect(url_for('index'))
```

Security Issue

- Can user modify a Cookie?
  - Can set any username
- If someone else gets Cookie, can they log in as user?
  - Timeout
  - Source IP
- Cross-site requests
  - Attackers can create page to automatically submit request to another site
  - If user is logged in on other site when they visit the attack page, it will automatically invoke action
  - Verify on the server that the request came from legitimate starting point

Server side information

- Maintain client information at the server
- Cookie only provides minimal lookup information
- Not easy to alter
- Requires persistent storage at the server
- Multiple backends possible
  - File storage
  - Database
  - Redis, other caching key-value stores

Enforce authentication

- Some parts of site must be protected
- How?
  - Enforce existence of specific token to access tho those views
- Views
  - determined by the controller

- Protect access to the controller
  - Flask controller → Python function
  - Protect function → add wrapper around it to check auth status
    - Decorator

Example → `flask_login`

```
from flask_login import login_required, current_user
...

@main.route('/profile')
@login_required
def profile():
  return render_template('profile.html', name=current_user.name)
```

```
from flask_login import login_user, logout_user, login_required
...

@auth.route('/logout')
@login_required
def logout():
  logout_user()
  return redirect(url_for('main.index'))
```

Transmitted data security

- Assume connection can be "tapped"
- Attacker should not be able to read the data
- HTTP GET URLs not good →
  - logged on firewalls, proxies, etc
- HTTP POST, Cookies etc →
  - if wire can be made safe, then good enough

How to make the wire safe?

Week 9 Lecture 4

Class	BSCCS2003
Created	@November 2, 2021 12:03 PM
Materials	
Module #	51
Type	Lecture
Week #	9

HTTPS

Normal HTTP process

- Open connection to server on a fixed network port (default 80)
- Transmit HTTP request
- Receive HTTP response

Safety of transmitted data?

- Can be tapped
- Can be altered

Secure sockets

- Set up an "encrypted" channel between the client and the server
- How?
  - Need a shared secret → eg. long binary string → this is the "key"
  - XOR all the input data with key to generate new binary data
  - Attacker without the key cannot derive the actual data
- How to set up the shared secret?
  - Must assume anything on the wire can be tapped
  - What about pre-existing key?
  - Secure side channel → send a token by a post, SMS

Types of security

- Channel (wire) security
  - Ensure that no one can tap the channel → most basic need for other auth mechanisms, etc.
- Server authentication
  - How do we know we are actually connection to mail.google.com and not some other server?
  - DNS hijacking is possible → redirect to another server
  - Server certificates
  - Common root of trust needed → someone who "vouches for" mail.google.com
- Client certificate
  - Rare but useful → server can require client certificate
  - Used especially in corporate intranets etc

Chain of Trust

- Chain of trust
  - mail.google.com issued certificate by
  - GTS CA1C3 issued certificate by
  - GTS Root R1
- GTS Root R1 certificate stored in Operating System or Browser
  - Do you trust your OS? Do you trust your browser?
    - interesting question, ngl
- From there on a secure (crypto) chain

Potential problems

- Old browsers
  - Not updated with new chains of trust
- Stolen certificates at root of trust
  - Certificate revocation, invalidation possible
  - Need to ensure OS, browser can update their trust stores
- DNS hijacking
  - Give false IPs for server as well as entries along chain of trust
  - But certification in OS will fail against eventual root of trust

Impact of HTTPS

- Security against wiretapping
- Better in public WiFi networks

Negative →

- Affects caching of resources (proxies cannot see content)
- Performance impact due to run-time encryption

Week 9 Lecture 5

Class	BSCCS2003
Created	@November 2, 2021 1:10 PM
Materials	
Module #	52
Type	Lecture
Week #	9

Logging

What is logging?

- Record all accesses to app
- Why?
  - Record bugs
  - Number of visits, usage patterns
  - Most popular links
  - Site optimization
  - Security checks
- How?
  - Built into the app → output to a log file
  - Direct output to analysis pipeline

Server logging

- Built in to Apache, Nginx, ...
- Just accesses and URL accessed
- Can indicate possible security attacks →
  - Large number of requests in short duration
  - Requests with "malformed" URLs

- Repeated requests to unusual endpoints

Application level logging

- Python logging framework
  - Output to a file, other "stream" handlers
- Details of application access
  - Which controllers
  - What data models
  - Possible security issues
- All server errors

```
! base ~/g/m/gradebook flask run
 * Serving Flask app 'application:app' (lazy loading)
 * Environment: development
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 674-210-362
127.0.0.1 - - [06/Sep/2021 21:04:21] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [06/Sep/2021 21:04:21] "GET /static/css/style.css HTTP/1.1" 304 -
127.0.0.1 - - [06/Sep/2021 21:04:21] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [06/Sep/2021 21:04:27] "GET /user/ HTTP/1.1" 200 -
127.0.0.1 - - [06/Sep/2021 21:04:27] "GET /static/css/style.css HTTP/1.1" 304 -
127.0.0.1 - - [06/Sep/2021 21:04:34] "GET /user/1 HTTP/1.1" 200 -
127.0.0.1 - - [06/Sep/2021 21:04:34] "GET /static/css/style.css HTTP/1.1" 304 -
```

Log rotation

- High volume logs → mostly written, less analysis
- Cannot store indefinitely
  - Delete old entries
- Rotation →
  - Keep last N files
  - Delete oldest file
  - Rename log.i to log.i+1
  - Fixed space used on server

Logs on custom app engines

- Google App Engine
  - custom logs
  - custom reports
- Automatic security analysis

Time series analysis

- Logs are usually associated with timestamps
- Time series analysis →
  - How many events per unit time
  - Time of specific incident(s)
  - Detect patterns (periodic spikes, sudden increase in load)
- Time-series databases
  - RRDTool, InfluxDB, Prometheus, ...

    - Analysis and visualization engines

Summary

Security is the key to successful applications

- Requires good understanding of principles
    - Crypto
    - SQL, OS Vulnerabilities, ...
- Good frameworks to be preferred
- Analyze, Identify, Fix