



Web Programming

CSc 337



Welcome to CSc 337

Web programming

- Prerequisite: CSC 120, ISTA 130, ECE 175 (equivalent experience in a high level programming language that includes variables, strings, selection, loops, methods, parameters, and arrays.)
- Looks at
 - The basics of how the web works (HTTP, HTTPS, web browsers)
 - Client-side web programming (HTML, CSS, and Javascript)
 - Server-side web programming (Javascript, Nodejs, Database)



Instructional staff

Instructor: Reyan Ahmed

- Office: Gould-Simpson room 831
- Email: abureyanahmed@arizona.edu
- Office hours (preliminary):
 - Tuesdays 3:30PM - 5:00PM
 - Thursdays 3:00PM - 4:30PM
 - or by appointment (send email; put CSc 337 in Subject)
 - any changes will be posted in D2L/Discord



Instructional staff

Teaching assistant:

- Jesse Chen (jessechen@arizona.edu)
- Bronson Housmans (bhousmans@arizona.edu)
- Jeziel Banos Gonzalez (jezielbgon@arizona.edu)
- Zachary Hansen (zacharysehansen@arizona.edu)
- Office: TBD



Course communications

- Lecture sessions
- Office hours
- D2L website
- Discord: <https://discord.gg/nsgsuppq>



Course materials

- Lecture slides (D2L)
- Lecture recording (Panopto)
- Other resources:
 - <https://www.w3schools.com/>
 - Node.js, MongoDB and Angular Web Development (Developer's Library) by Brad Dayley



In class activities (10 points)

- There will be in class activities (ICA) almost in every lecture
- A couple of related questions will be asked during the class
- Take pictures and combine a pdf if you are writing in paper
- Submit the pdf at Gradescope by 11 pm
- Will be graded for completion
- Will help me to understand how fast I should proceed
- There will be at least 40 ICAs
- Each worth of around $\frac{1}{4}$ points
- You can collaborate with others, but you have to submit individually



Weekly Assignments (32 points)

- Approximately one homework per week (first 12 weeks)
- Will be posted on Tuesdays
- You will get around a week to complete
- Submit at Gradescope
- Will be graded for accuracy
- Assignment 1 will be posted on Tuesday, January 21st
- Will contain problems from topics covered recently
- 9 assignments in total
- Best 8 will be counted ($8 \times 4 = 32$)
- Must be done and submitted individually



Weekly Assignments (32 points)

- You should not take direct help from external sources
- However, if a code does not work due to version mismatch then you can take help
- For example, consider the following code:

```
const fs = require('fs');  
const file = new fs.SyncWriteStream('example.txt', { flags: 'w' });  
file.write('Hello, world!');  
file.end();
```



Weekly Assignments (32 points)

- Now consider another piece of code:

```
const fs = require('fs');  
const file = fs.createWriteStream('example.txt');  
file.write('Hello, world!');  
file.end();
```



Weekly Assignments (32 points)

- In some exceptional cases, you can take help from external resources:
 - Software installation
 - A code shown in class does not work in your machine due to version mismatch

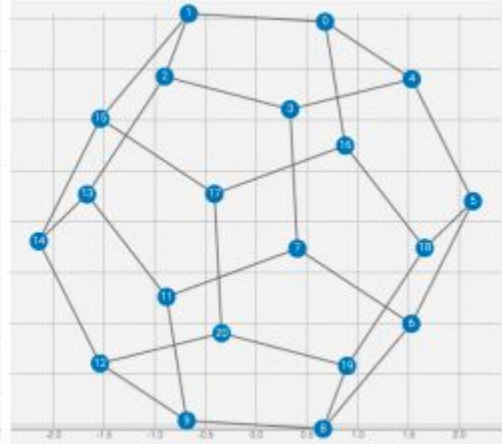
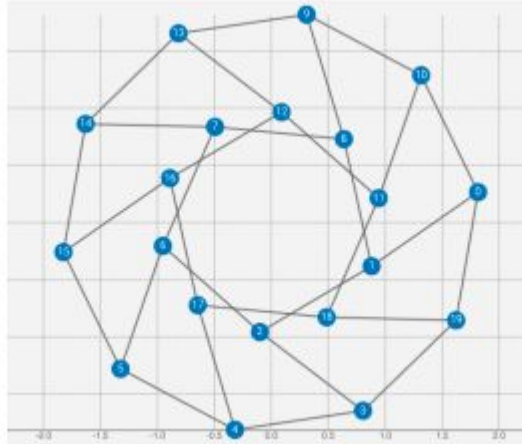
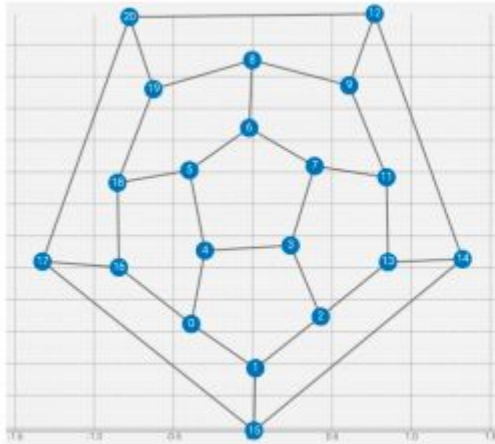


Project (8 points)

- Will give an idea how real life web applications look like
- You will get around 3-4 weeks at the end of the semester
- Will be a group project consisting of at most 3 members
- You have to combine different techniques:
 - client side programming
 - server side programming
 - databases

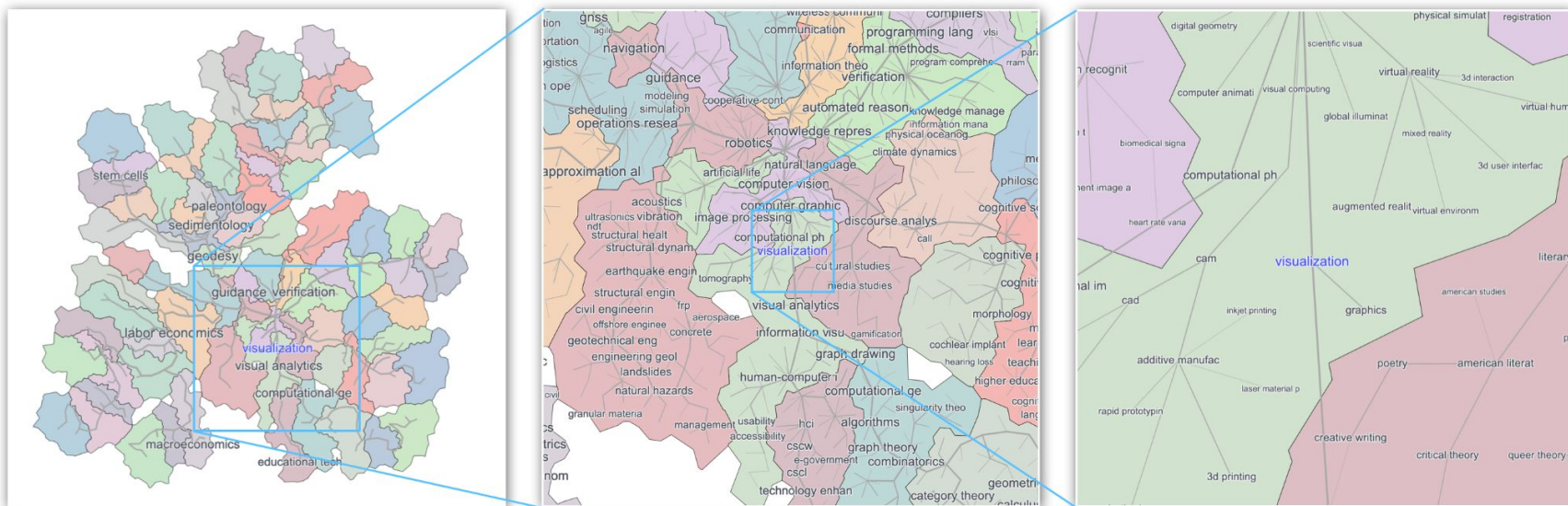
Web programming examples

- Stress optimization
 - Example: <http://hdc.cs.arizona.edu/~mwli/graph-drawing/>
 - Paper: <https://arxiv.org/pdf/2008.05584.pdf>



Web programming examples

- Using map to visualize networks
 - Example: <https://tiga1231.github.io/zmlt/demo/overview.html>
 - Paper: <http://www2.cs.arizona.edu/~kobourov/pacvis10.pdf>



Web programming examples

- Dynamic network visualization
 - Example: <https://ryngray.github.io/dynamic-trees/>
 - Paper: <https://arxiv.org/pdf/2106.08843.pdf>



Midterms (30 points)

- Midterm 1 (15 points)
 - Thursday, February 27 , 2025
- Midterm 2 (15 points)
 - Tuesday, April 8, 2025
- These are in-person exams
- Given during class
- Closed book exams
- There will be no make up
- No replacement



Final (20 points)

- Date: Tuesday, 05/13/2025
- Time: 8:00am - 10:00am
- Location: regular classroom
- I will remind you again before the exam
- Closed book exam
- There will be no make up
- No replacement



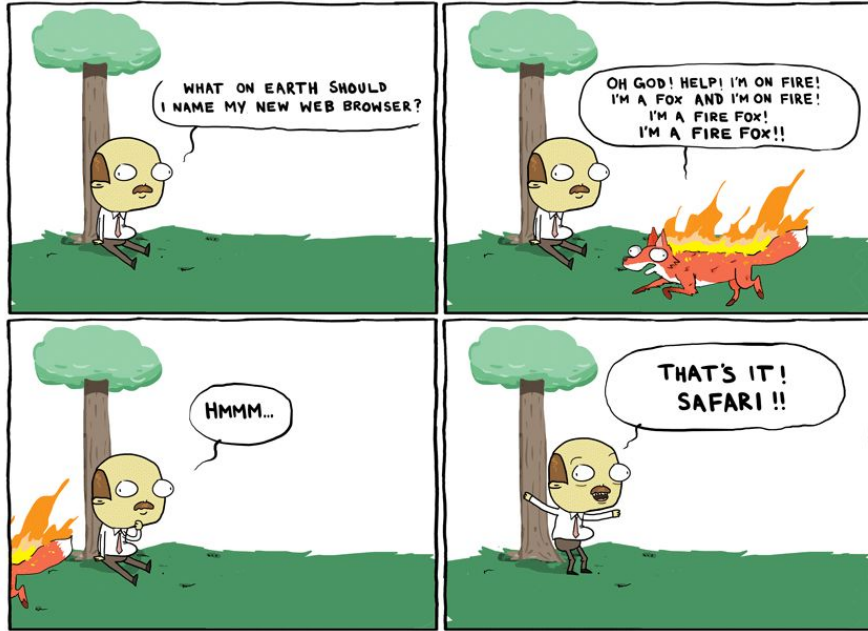
Point distribution

In class activities	10%
Weekly assignments	32%
Project	8%
Midterm 1	15%
Midterm 2	15%
Final	20%



Others

- Grading will be completed in a week
- You will get 7 days to request regrades
- Take a look at the syllabus for other details
- Feel free to ask questions



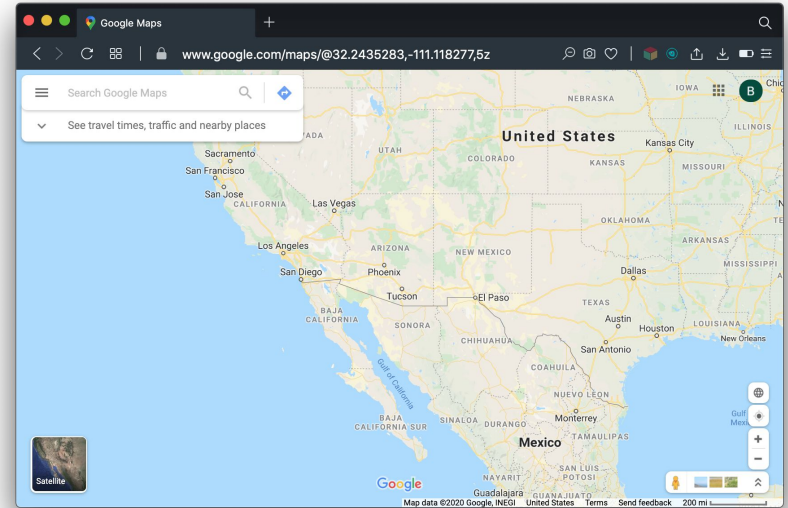
CSc 337 - Browsers, URLs, and HTTP

Reyan Ahmed

What is a web browser?

A web browser ... is a software application for accessing information on the World Wide Web. When a user requests a web page from a particular website, the web browser retrieves the necessary content from a web server and then displays the page on the screen.

(Wikipedia)

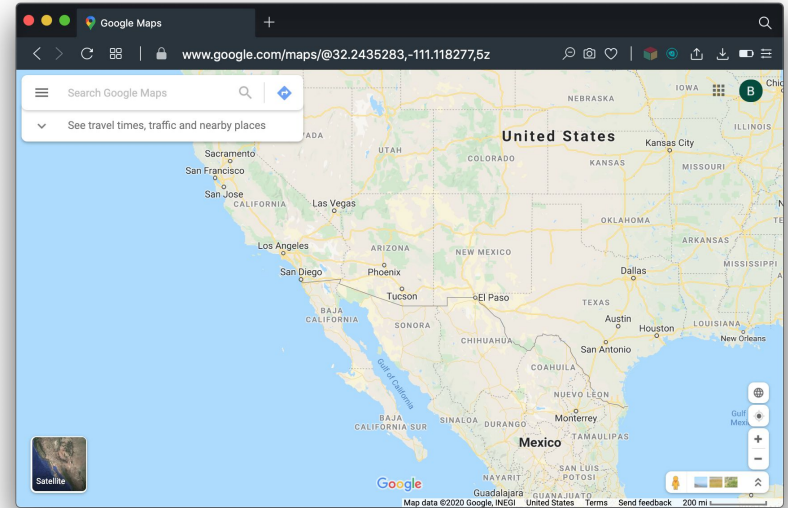


What is a web browser?

Content (pages) are accessed via the Uniform Resource Locator, URL

Can manually request pages via URL

Don't always have to memorize URLs (search engines, bookmarks, etc)



What do you think are the top three
Desktop browsers?

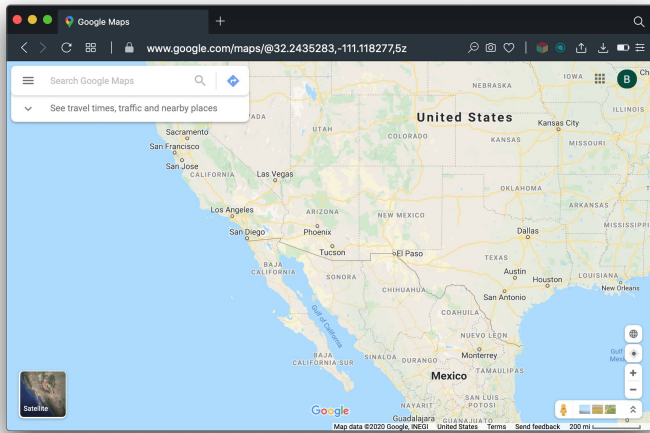
What about for mobile?

What do you think are the top three
Desktop browsers?

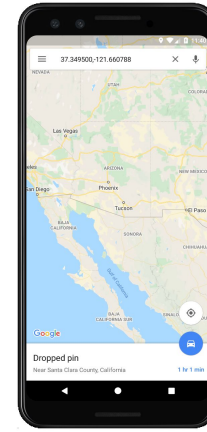
<https://gs.statcounter.com/browser-market-share>

What about for mobile?

<https://gs.statcounter.com/browser-market-share/mobile/worldwide>



- The screen is wider
- Can show more information in a row



- Smaller width
- Need to break down larger rows into multiple rows

Uniform Resource Locator

<https://www.reddit.com/r/UofArizona/>

Uniform Resource Locator

The **DOMAIN** - Used to identify the location (computer, server, or group of servers) to get the resource from

The **PATH** to the specific resource or file within the specified domain.

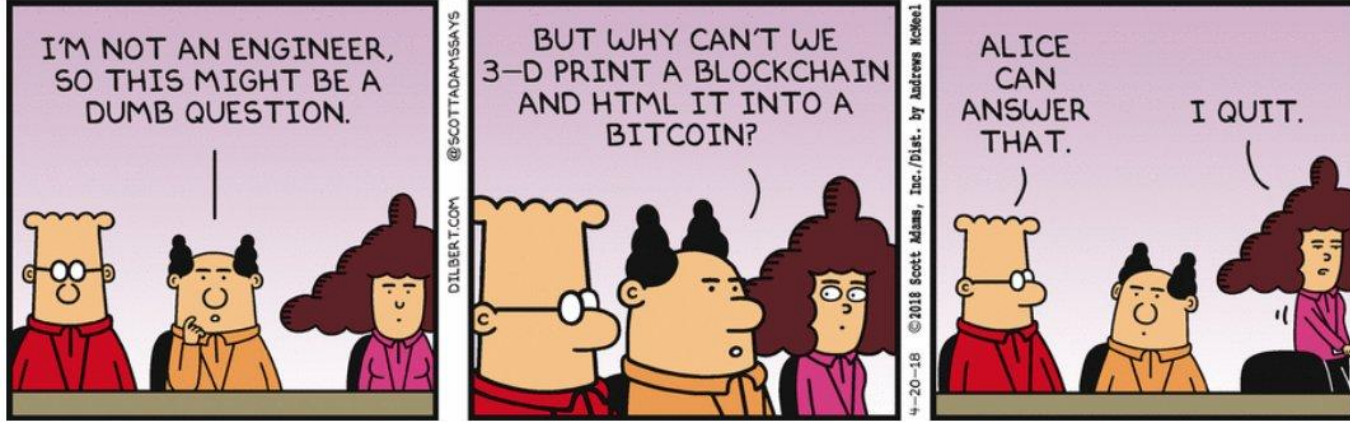
`https://www.reddit.com/r/UofArizona/`

The **SCHEME** (or **PROTOCOL**). Used to communicate between the browser, and the source of the page (server)

Domains can end with things other than .com

Starting domains with www isn't required, just convention

Other stuff can go here, more on that in the future



CSc 337 - HTML

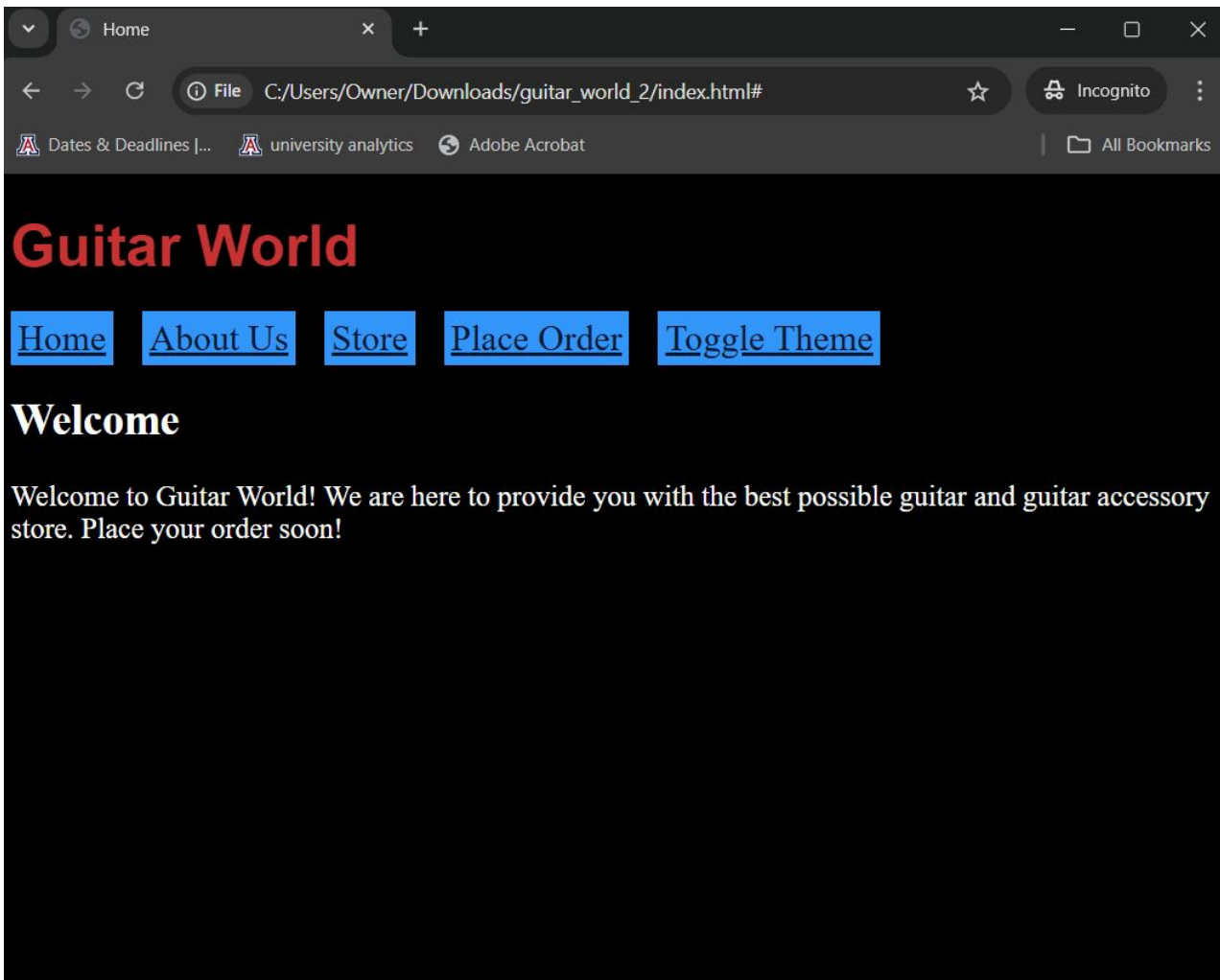
Reyan Ahmed

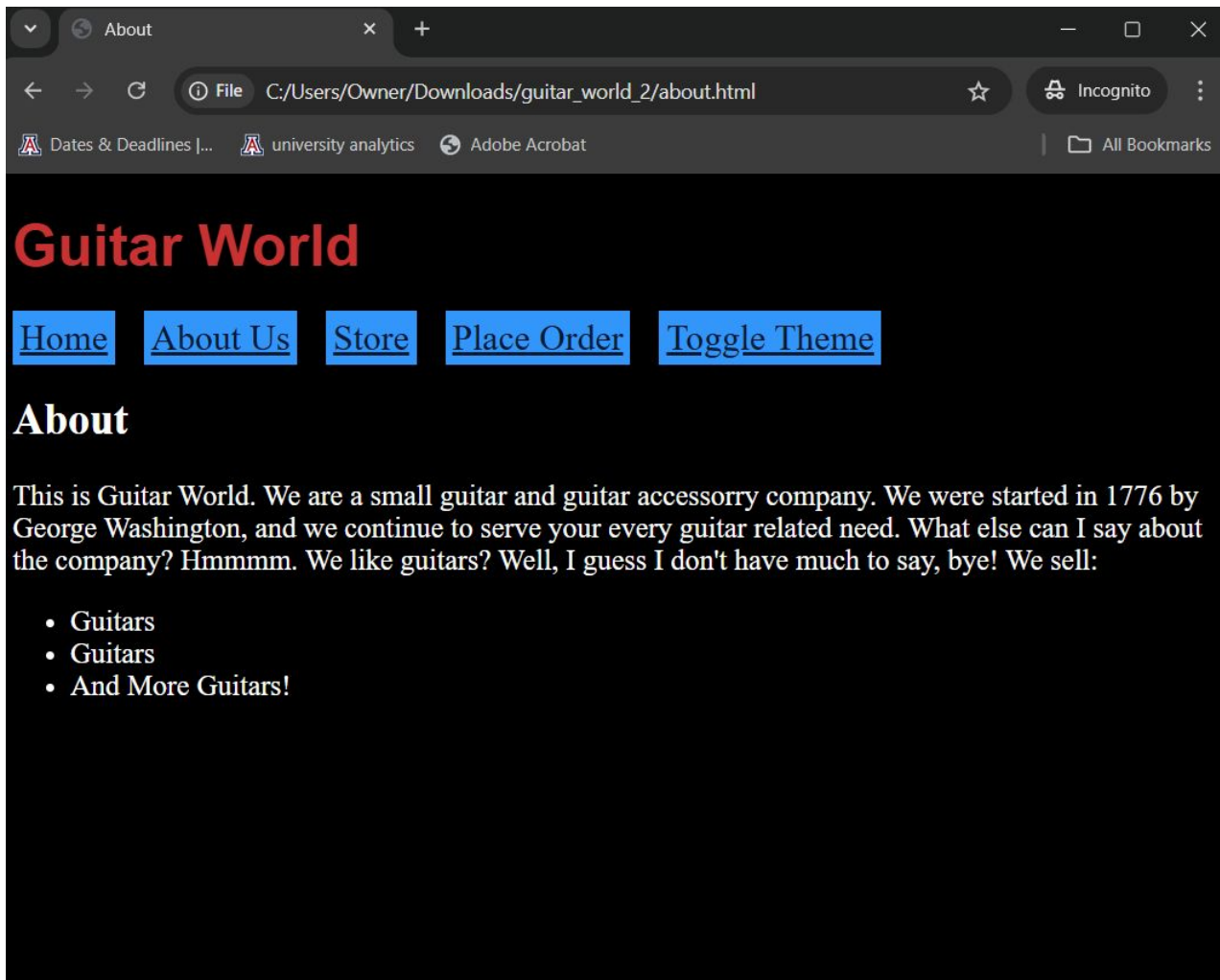
HyperText Markup Language

- Markup Language != Programming Language
- [HTML](#) specifies the ***content*** of the website
- Not responsible for
 - Style (colors, fonts, positioning, etc) **CSS**
 - Animation / Interaction **CSS / Js**
 - Functionality / Computations **Js**
 - Networking **Js**

HyperText Markup Language

- Markup Language != Programming Language
- [HTML](#) specifies the ***content*** of the website
- Not responsible for
 - Style (colors, fonts, positioning, etc) **CSS**
 - Animation / Interaction **CSS / Js**
 - Functionality / Computations **Js**
 - Networking **Js**





Products

C:/Users/Owner/Downloads/guitar_world_2/store.html

Dates & Deadlines |... university analytics Adobe Acrobat




Incognito

All Bookmarks

Guitar World

[Home](#)[About Us](#)[Store](#)[Place Order](#)[Toggle Theme](#)

Products

Image	Name	Price	Description
	Taylor 110-GB	\$499.89	ABC 123
	Martin 00-X2E	\$799.00	ABC 123
	Ibanez AZ2402	\$1729.99	ABC 123

Order

FileC:/Users/Owner/Downloads/guitar_world_2/order.html

Dates & Deadlines [...]university analyticsAdobe AcrobatAll Bookmarks

Guitar World

[Home](#)[About Us](#)[Store](#)[Place Order](#)[Toggle Theme](#)

Order

Full Name

Jackie Chan

CC number

Age

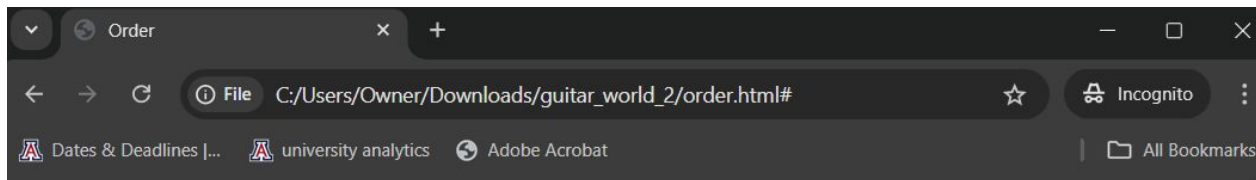
Email Address

Product to Order

Taylor 110 gb • Martin 00x2e • Ibanez az2402 •

Every guitar you purchase from Guitar world comes with a free case. Select your case color here.

Submit Info



Guitar World

[Home](#)[About Us](#)[Store](#)[Place Order](#)[Toggle Theme](#)

Order

Full Name

CC number

Age

Email Address

Product to Order Taylor 110 gb ☐ Martin 00x2e ☐ Ibanez az2402 ☐

Every guitar you purchase from Guitar world comes with a free case. Select your case color here.



```
<!DOCTYPE html>
```

```
<html>
```

```
  <head>
```

```
  </head>
```

```
  <body>
```

```
  </body>
```

```
</html>
```

```
<!DOCTYPE html>
```

```
<html>
```

```
  <head>
```

```
    <title>Guitar World</title>
```

```
  </head>
```

```
  <body>
```

```
    This website is all about guitars.
```

```
  </body>
```

```
</html>
```

```
<!DOCTYPE html>
<html>
  <head>
    <title>Guitar World</title>
  </head>
  <body>
    <h1>Guitar World</h1>
    <h2>Home Page</h2>
    <h3>Welcome!</h3>
    <p>Buy guitars here!</p>
  </body>
</html>
```

Display Type

- By default, browsers display many HTML elements in either **block** or **inline** mode
- Block means that each element is on its own row, and consumes the entire page width
- Inline means that elements can be displayed inline, do not consume entire width
- **** vs **<div>**

```
<!DOCTYPE html>
<html>
  <head>
    <title>Guitar World</title>
  </head>
  <body>
    <div>Taylor</div>
    <div>Ibanez</div>
    <div>Fender</div>
    <span>Martin</span>
    <span>Yamaha</span>
    <span>Ernie Ball</span>
  </body>
</html>
```


List Data

- For displaying, well, lists
- Use
 - `` to state the existence of an **unordered** list
 - `` to state the existence of an **ordered** list
 - `` represents a list element

```
<body>
  <ul>
    <li>Guitar</li>
    <li>Bass</li>
    <li>Drums</li>
  </ul>
  <ol>
    <li>Taylor</li>
    <li>Martin</li>
    <li>Ibanez</li>
  </ol>
</body>
```

```
<body>
  <ul>
    <li>Guitars
      <ul>
        <li>Acoustic</li>
        <li>Electric</li>
        <li>Air</li>
      </ul>
    </li>
    <li>Bass</li>
    <li>Drums</li>
  </ul>
</body>
```

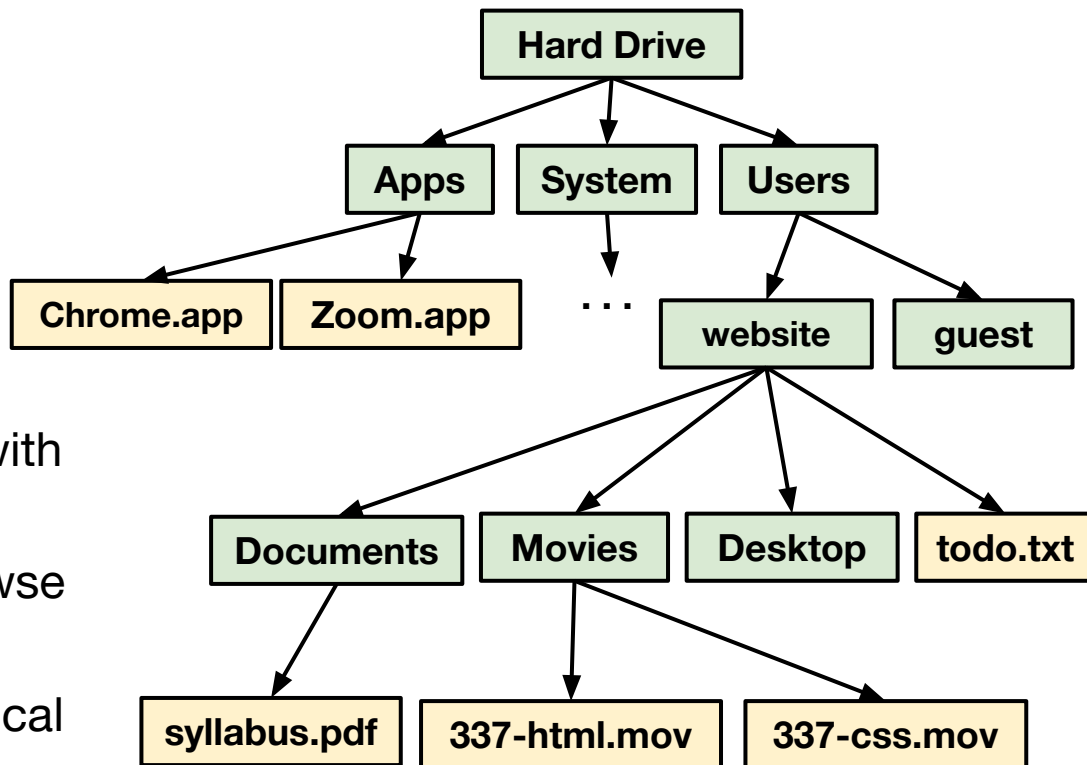
Tabular Data

- Often useful to display data in a grid-like format
 - ESPN stats, wikipedia data, gradebook
- Use
 - **<table>** to state the existence of the table, and within
 - **<tr>** represents a row in the table
 - **<th>** represents a header element within the table
 - **<td>** represents a data element within the table

```
<body>
  <table>
    <tr>
      <th>Guitar</th>
      <th>Bass</th>
    </tr>
    <tr>
      <td>Taylor</td>
      <td>Ernie Ball</td>
    </tr>
    <tr>
      <td>Martin</td>
      <td>Ibanez</td>
    </tr>
  </table>
</body>
```

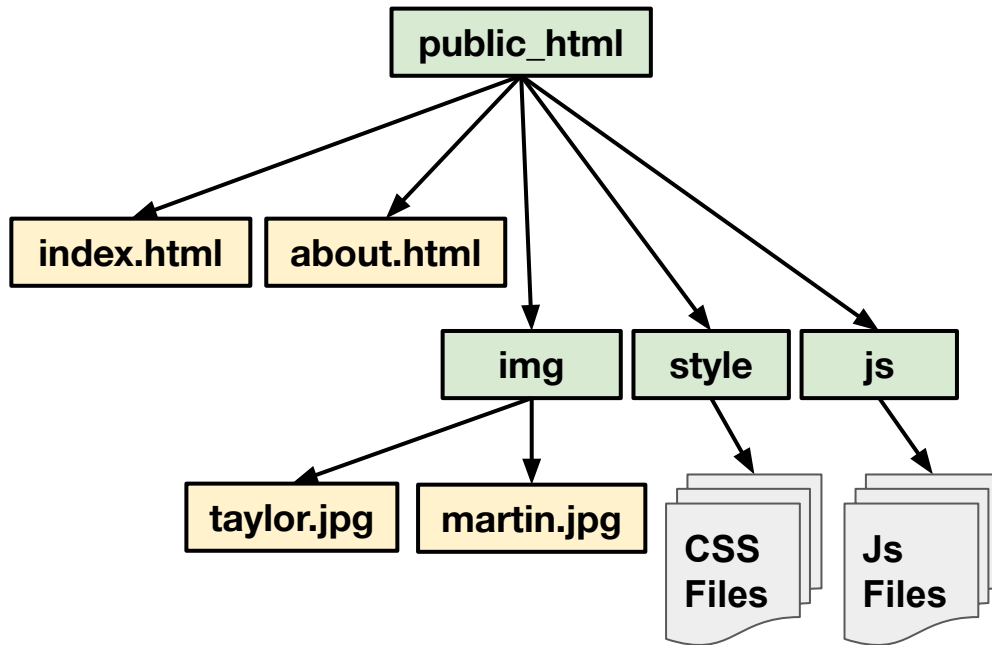
Files and File Systems

- On (at least most of) our computers, there is a **file system** via which we can create, save, modify, and remove files
 - On Mac: can browse with Finder
 - On Windows: can browse with windows explorer
- File systems often hierarchical



Website Structure

- All files go in public_html
- Always have an index.html
- Can have other “pages” too
- Subfolders for resources

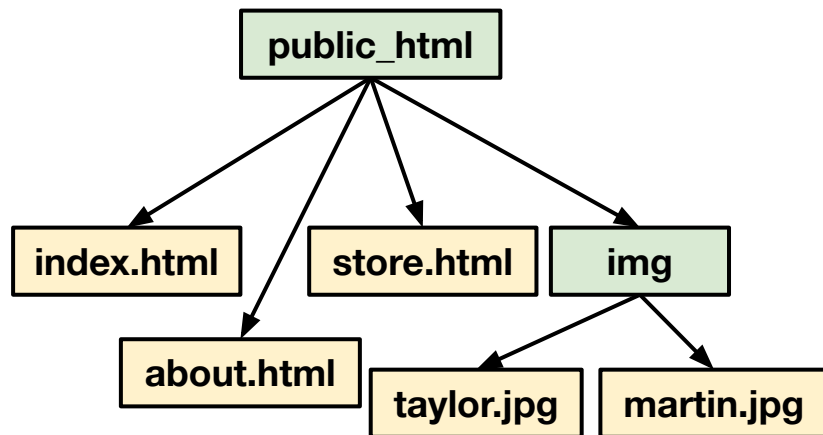


The a tag

`<a>` used to link from one page (html file) to another

In `index.html`:

```
<a href="./about.html">About Us</a>  
<a href="./store.html">Shop</a>
```



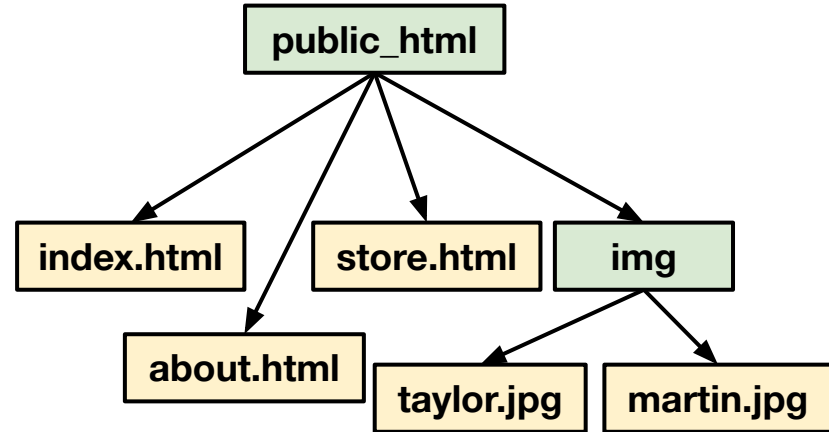
The img tag

`` used to link from one page (html file) to another, self-closing

In `store.html`:

```

```





CSc 337 - CSS

Reyan Ahmed

Cascading Style Sheets (CSS)

- A style sheet language
- Style Sheet Language != Programming Language
- CSS used for specifying *how* the contents of the site (HTML) is displayed
 - Coloring, stylizing, positioning, and sizing the elements within the HTML

General Syntax

```
identifier {  
    property-a : style-value ;  
    property-b : style-value ;  
    . . .  
}
```

- **identifier** specifies which element(s) to style
- **property** specifies the property to set
- **style-value** is what to set that property to

An Example

```
<!DOCTYPE html>
```

```
<html>
```

```
  <head>
```

```
    <style>
```

```
      p {
```

```
        color: red;
```

```
        background-color: blue;
```

```
      }
```

```
    </style>
```

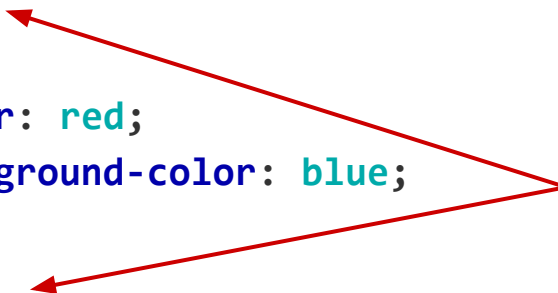
```
  </head>
```

```
  <body>
```

```
    <p>Taylor Guitars</p>
```

```
  </body>
```

```
</html>
```



Notice the <style>
tag - for putting CSS
inside of

Different ways, same result

Style tag

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      p {
        color: red;
        background-color: blue;
      }
    </style>
  </head>

  <body>
    <p>Taylor Guitars</p>
  </body>
</html>
```

index.html

Style attribute

```
<!DOCTYPE html>
<html>
  <head>
  </head>

  <body>
    <p style="color:red;
      background-color:blue;">
      Taylor Guitars
    </p>
  </body>
</html>
```

index.html

Separate Files

```
p {
  color: red;
  background-color: blue;
}
```

style.css

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet"
      type="text/css"
      href="style.css">
  </head>

  <body>
    <p>Taylor Guitars</p>
  </body>
</html>
```

index.html

Specific Styling

- What if we don't want to style elements based on their tag?
- For instance,
 - Want to color half of the paragraphs with one color, and half another?
 - Want to style just one, specific div amongst many?
 - Want to resize one image on the page?

Class and ID attributes

- Can give *any* element in the HTML a class name and an id name
- Multiple elements (even with different tags) can be in the same class
- IDs must be unique, not duplicate IDs on a single HTML page

```
<body>
  <p id="taylor" class="guitar">
    Taylor
  </p>
  <span id="martin" class="guitar">
    Martin
  </span>
  <h1 id="pearl" class="drums">
    Pearl
  </h1>
  <h2 id="gretsch" class="drums">
    Gretsch
  </h2>
</body>
```


Many, Many Properties

Some common ones . . .

color

font-size

height

margin

display

font-family

border

text-align

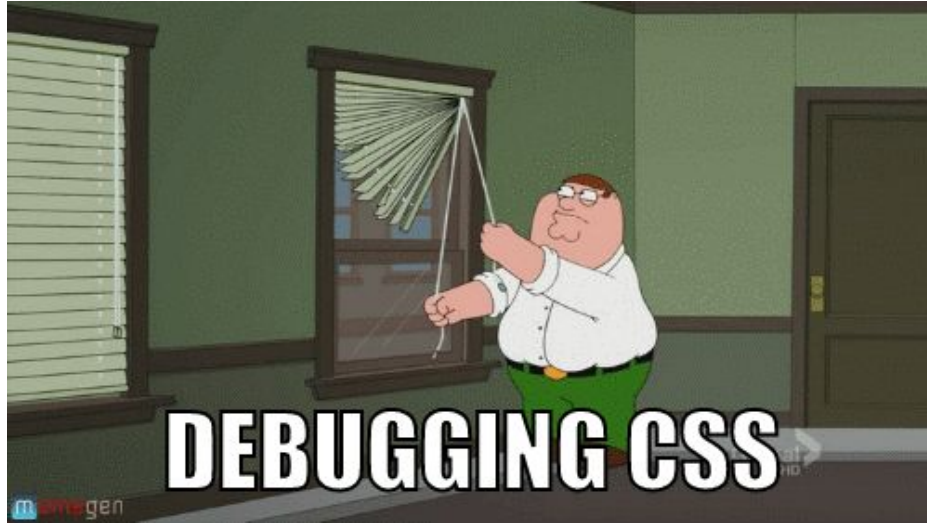
padding

background-color

width

visibility

position



CSc 337 - CSS Layout

Reyan Ahmed

CSS Layout

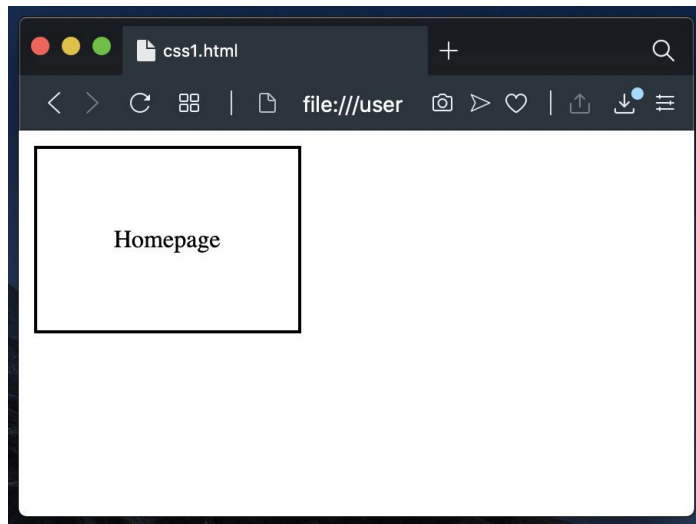
- In some (many?) cases, a web programmer wants to customize the way elements are positioned and laid out relative to the viewing window, or other elements
- Having a good understanding of CSS layout helps here
- Key CSS properties for this:
 - **margin** and **padding**
 - **display** and **position**
 - **width** and **height**
 - **top, bottom, left, right**
 - **float**

Margin and Padding (see dev console)

```
div {  
  border: solid 2px;  
  width: 70px;  
  margin: 10px;  
  padding: 50px;  
}
```

• • •

```
<div>Homepage</div>
```

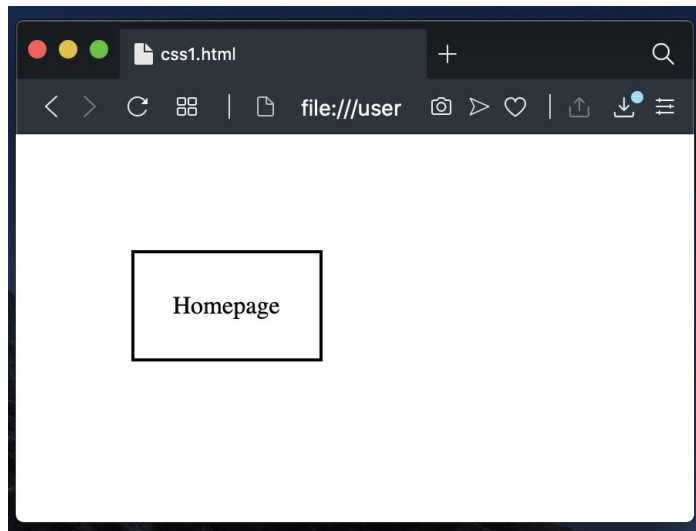


Margin and Padding (see dev console)

```
div {  
  border: solid 2px;  
  width: 70px;  
  margin: 75px;  
  padding: 25px;  
}
```

• • •

`<div>Homepage</div>`

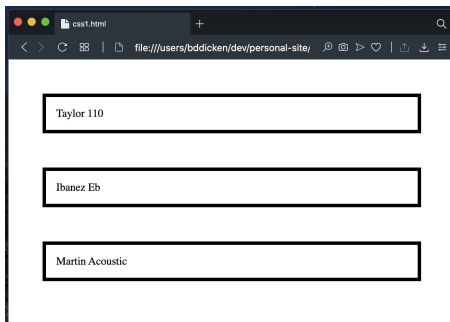


Display

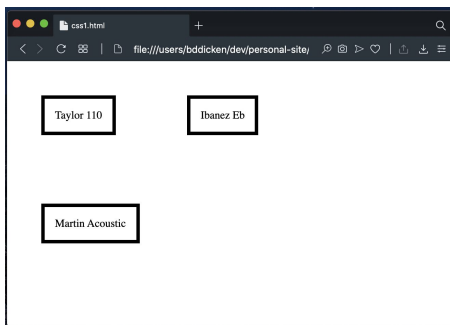
```
div {  
  display: ???;  
  border: solid 5px;  
  margin: 50px;  
  padding: 15px;  
}
```

• • •

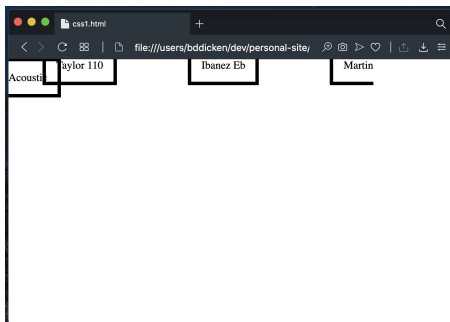
```
<div>Taylor 110</div>  
<div>Ibanez Eb</div>  
<div>Martin Acoustic</div>
```



display: block;



display: inline-block;



display: inline;

Position

`static` - not positioned

`relative` - not positioned, unless given extra properties

`fixed` - positioned relative to viewport

`absolute` - positioned relative to nearest positioned ancestor

Tough validation



CSc 337 HTML Forms

Reyan Ahmed

GETting vs POSTing

- Previously only wrote pages that display information, not much interaction other than `<a>` and resizing
- HTML forms allow us to get information from the user, receive it on our server

HTML Forms

- An HTML form consists of a `<form>` element with a number of `<label>` and `<input>` fields within
- The labels and inputs should be paired up 1 to 1

Full Name

Password

Age

Email Address

How tall are you? Tall ☒ About average ☐ Short ☐

Favorite Color!

Form Input types

- [Many, Many types](#)
- A few covered in class, but might be asked to use others in assignments and such

The diagram shows a web form with the following elements and their corresponding input types:

- Full Name** → **text**
- Password** → **password**
- Age** → **number**
- Email Address** → **email**
- How tall are you?** Tall ☒ About average ☐ Short ☐ → **radio**
- Favorite Color!** → **color**
- Submit Info** → **submit**

Form Element Attributes

- Every `<form>` element should have an **action** attribute
(can also specify **method**="..")
- Every `<input>` element should have a **type**, **id**, and **name** attribute
- Every `<label>` element should have a **for** attribute, which should match the **id** of the corresponding input



CommitStrip.com

CSc 337

Javascript

Reyan Ahmed

What is Javascript?

- Different language than Java, though some similarities
- Sometimes referred to as ECMAScript
- Has variables, for loops, objects, arrays (lists), functions
- [JsLint](#)



How Do I Run Javascript?

- The web console
- NodeJS
- Web runners, such as <https://playcode.io>



Numbers

```
var x = 3.5 // assign 3.5 to x
```

```
var y = 2 // assign 2 to y
```

```
var z = x*y // multiplication
```

```
console.log(z)
```

Similarly, we can add (+), subtract (-), divide (/).

If else

```
var body = document.getElementsByTagName("body")[0]
var bcolor = body.style["background-color"]
if(bcolor==" " || bcolor=="white"){
    body.style["background-color"]="black"
}
else if(bcolor=="black"){
    body.style["background-color"]="yellow"
}
else{
    body.style["background-color"]="white"
}
```

Average Numbers

```
var numbers = window.prompt('Give me numbers')
var numbers = numbers.split(',')
var total = 0
for(var count=0;count < numbers.length;count++) {
    total += parseInt(numbers[count])
}
console.log('The average is: ' +
(total / numbers.length))
```

parseFloat()

- parseFloat() is a built-in JavaScript function that parses a string argument and returns a floating point number.
- It is useful for converting a string that represents a decimal number into a float.

Math.round()

- Math.round() is a built-in JavaScript function that rounds a number to the nearest integer.
- It rounds up if the decimal part is 0.5 or higher, and rounds down if it is less than 0.5.

Other Math functions

- `Math.abs()`
- `Math.pow()`
- `Math.max()`
- `Math.min()`
- `Math.random()`
- ...

Arrays (or lists)

// Javascript

```
groceryList = [];  
groceryList.push('egg');  
groceryList.push('bacon');  
  
console.log(groceryList[0]);  
  
for(var i=0;i<groceryList.length;i++)  
{  
    console.log(groceryList[i]);  
}
```

Python

```
grocery_list = []  
grocery_list.append('egg')  
grocery_list.append('bacon')  
  
print(grocery_list[0])  
  
for i in range(len(grocery_list)):  
    print(grocery_list[i])
```

Functions

// Javascript

```
function sum(a, b, c) {  
    return a + b + c;  
}
```

```
function sum(numbers) {  
    result = 0;  
    for (var i = 0; i < numbers.length; i++) {  
        result += numbers[i];  
    }  
    return result;  
}
```

Python

```
def sum(a, b, c):  
    return a + b + c
```

```
def sum(numbers):  
    result = 0  
    for element in numbers:  
        result += element  
    return result
```

Boolean variables

Values are true and false

- `var x = true`
- `var y = false`

Logical not

- `!x==false`
- `!y==true`

Logical or: `x || y`

Logical and: `x && y`

Loop features

- break: will break the loop
- continue: will ignore the current iteration and go to next iteration

Problem: Unique Numbers

Write a JavaScript function:

- that takes a list of numbers
- and returns a list of unique numbers in the input list.

Strings

- Consider it a list of characters
- Then everything we learned about list is applicable

```
var s = "Hello world!"  
for(var i=0;i<s.length;i++)  
{  
    console.log(s[i])  
}
```

Strings

- How can we reverse a string?
- Let's write a function!

Strings

Can we write a function to check palindrome (i.e. “madam”, “level” ...)?

- Reverse the string
- Check whether the reversed string is equal to the input string!
- Let's write the code!

Strings

Can we write a function to check palindrome (i.e. “madam”, “level” ...)?

- Reverse the string
- Check whether the reversed string is equal to the input string!
- Let's write the code!

`is_palindrome("madam")` is true.

But `is_palindrome("Madam")` is false!

Hmm, because ... javascript is case sensitive (most programming languages are)

Strings

- `"Madam".toLowerCase()` -> `"madam"`
- `"Madam".toUpperCase()` -> `"MADAM"`

Can we write a case insensitive palindrome checker?

Ofcourse, let's write it!

While loop

```
var s = "Hello world!"
```

```
var i = 0
```

```
while(i<s.length)
```

```
{
```

```
    console.log(s[i])
```

```
    i += 1
```

```
}
```


More data structures

So far we have seen:

- Numbers
- Strings
- Lists

What is the fanciest data structure you have seen?

- To me it is dictionary/hash map

Object (hashmap of javascript)

```
var x = {"a":97, "b":98, "c":99}  
console.log(x["a"])
```

Or

```
var x = {}  
x["a"]=97  
x["b"]=98  
x["c"]=99  
console.log(x["a"])
```

Object

`Object.keys()` is a built-in JavaScript method that returns an array of the own enumerable property names (keys) of an object.

```
var x = {"a":97, "b":98, "c":99}
var y = Object. keys(x)
for(var i=0;i<y.length;i++)
{
    console.log(y[i])
}
```

Object

Create an inventory object where:

- Each item name (e.g., "apple", "banana") is a key.
- The value of each item is an object containing:
 - price: The price of the item (number).
 - quantity: The number of items in stock (number).

Calculate the total value of the inventory by summing the price multiplied by quantity for all items.

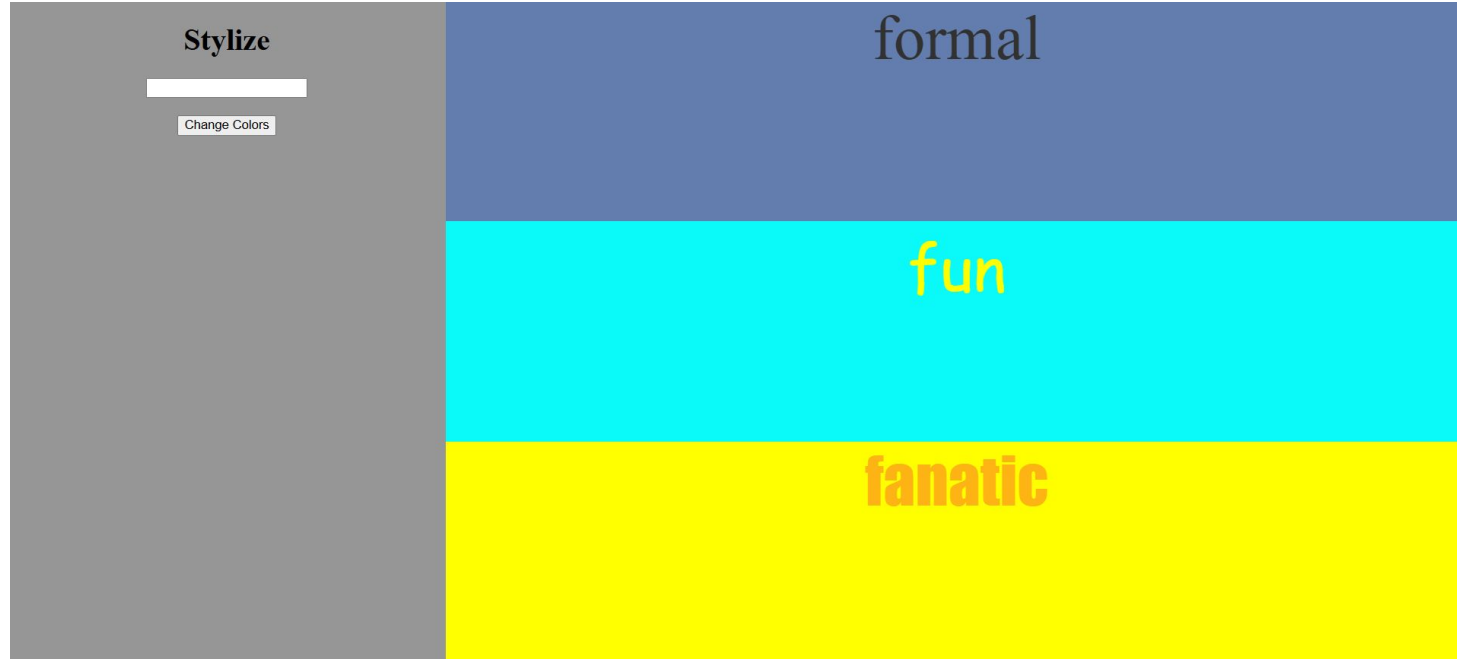


CSc 337

Building a Site with HTML, CSS, and Javascript

Reyan Ahmed

Let's build it!



Let's build it!

Requirements:

- The page should look similar to the image
- The colors of the right blocks will change if we hover or click the button
- The heading texts will change to the text written in the input text



CSc 337

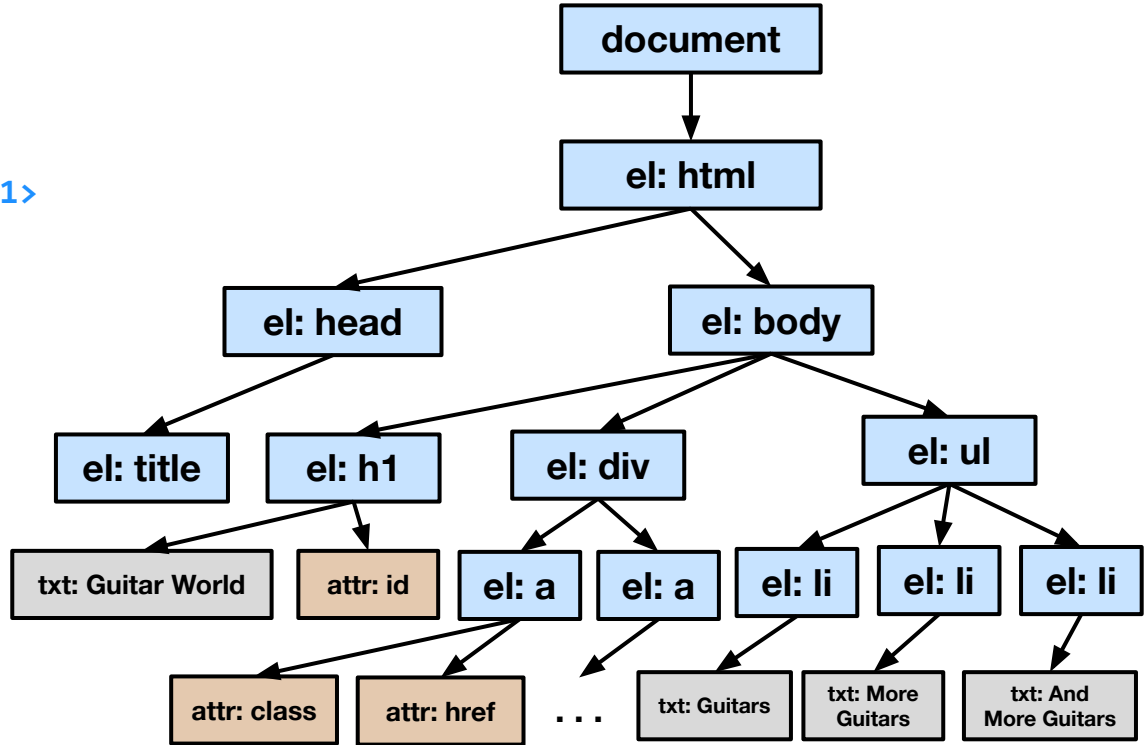
DOM

Reyan Ahmed

Document Object Model

- The DOM is the tree-structured, “behind the scenes” representation of the elements, attributes, etc of a webpage
- Can explore the dom via the inspector

```
<html>
<head>
  <title>About</title>
</head>
<body>
  <h1 id="mainTitle">Guitar World</h1>
  <div>
    <a class="mainLink"
      href="index.html">Home</a>
    <a class="mainLink"
      href="store.html">Store</a>
  </div>
  <ul>
    <li>Guitars</li>
    <li>More Guitars</li>
    <li>And More Guitars!</li>
  </ul>
</body>
</html>
```



Use Js to access DOM

- The DOM is the tree-structured, “behind the scenes” representation of the elements, attributes, etc of a webpage
 - Js Objects
- Can explore the dom via the developer console
- Access via the **document** variable

Use Js to access DOM

```
function printDOM(element) {  
  console.log(element.tagName);  
  if (element.children == undefined) { return; }  
  else {  
    for (c in element.children) {  
      printDOM(element.children[c]);  
    }  
  }  
}
```

```
printDOM(document);
```

```
<html>
<head>
  <title>Microsoft</title>
</head>
<body>
  <h1 id="aTitle">Microsoft</h1>
  
  <div id="mainContent">
    <span>
      Click
      <a href="buy.html">here</a>
      To buy a surface pro!
    </span>
  </div>
</body>
</html>
```

How many
DOM
elements?

What is the deepest DOM element level?

```
<html>
<head>
  <title>Microsoft</title>
</head>
<body>
  <h1 id="aTitle">Microsoft</h1>
  
  <div id="mainContent">
    <span>
      Click
      <a href="buy.html">here</a>
      To buy a surface pro!
    </span>
  </div>
</body>
</html>
```

Accessing specific elements

Methods in the document object:

getElementById(string)

getElementsByClassName(string)

getElementsByTagName(string)

Returns object(s) representing element(s)
in the DOM

Text vs HTML

One of the most basic things to do with a DOM element is set/get the text and HTML inside of it

`element.innerHTML`

`element.innerText`

What is the difference?

HTML symbols

- `<` for $<$
- `>` for $>$
- `∇` for ∇
- `∈` for \in
- `∉` for \notin
- ...

[Symbol list link](#)

```
<html>
<head>
  <title>Microsoft</title>
</head>
<body>
  <h1 id="aTitle">Microsoft</h1>
  
  <div id="mainContent">
    <span>
      Click
      <a href="buy.html">here</a>
      To buy a surface pro!
    </span>
  </div>
</body>
</html>
```

What does this print?

```
var c = document.getElementById('mainContent')
console.log(c.innerHTML)
console.log(c.innerText)
```

```
<html>
<head>
  <title>Microsoft</title>
</head>
<body>
  <h1 id="aTitle">Microsoft</h1>
  
  <div id="mainContent">
    <span>
      Click
      <a href="buy.html">here</a>
      To buy a surface pro!
    </span>
  </div>
</body>
</html>
```

What does this print?

```
var c = document.getElementById('mainContent')
c.innerHTML = 'hi'
console.log(c.innerText)
```

Changing Styles

Can change style (CSS) properties by updating the **style** information in the DOM

```
element.style[PROPERTY] = VALUE
```

or

```
element.style.PROPERTY = VALUE
```

JavaScript setInterval() Method

The setInterval() method in JavaScript is used to repeatedly execute a function or code snippet at specified time intervals (in milliseconds).

Syntax: `let intervalId = setInterval(function, milliseconds, arg1, arg2, ...)`

- **function:** The function to execute.
- **milliseconds:** The delay between each call to the function (in milliseconds).
- **arg1, arg2, ... (Optional):** Arguments to pass to the function when it is executed.

JavaScript clearInterval() Method

The `clearInterval()` method in JavaScript is used to stop the execution of a function that was previously set up with `setInterval()`. It prevents the function from being called repeatedly after a certain interval.

Syntax: `clearInterval(intervalId)`

- `intervalId`: The ID of the interval to be cleared, which is returned by `setInterval()`.

```
function getColor() {  
    let randomValue = Math.random();  
    if (randomValue < 0.3) { return 'red'; }  
    else if (randomValue < 0.7) { return 'blue'; }  
    return 'white';  
}
```

```
function patriotify() {  
    var elements = document.getElementsByTagName("*");  
    for(var i = 0; i < elements.length; i++) {  
        elements[i].style.backgroundColor = getColor();  
    }  
}
```

```
window.setInterval( patriotify, 1000 );
```

How would
this affect a
webpage?

Creating and adding elements

Method to create element:

```
document.createElement(element_name)
```

Method to create text:

```
document.createTextNode("Some text")
```

Method to add element:

```
parent_element.appendChild(child_element)
```

Method to remove element:

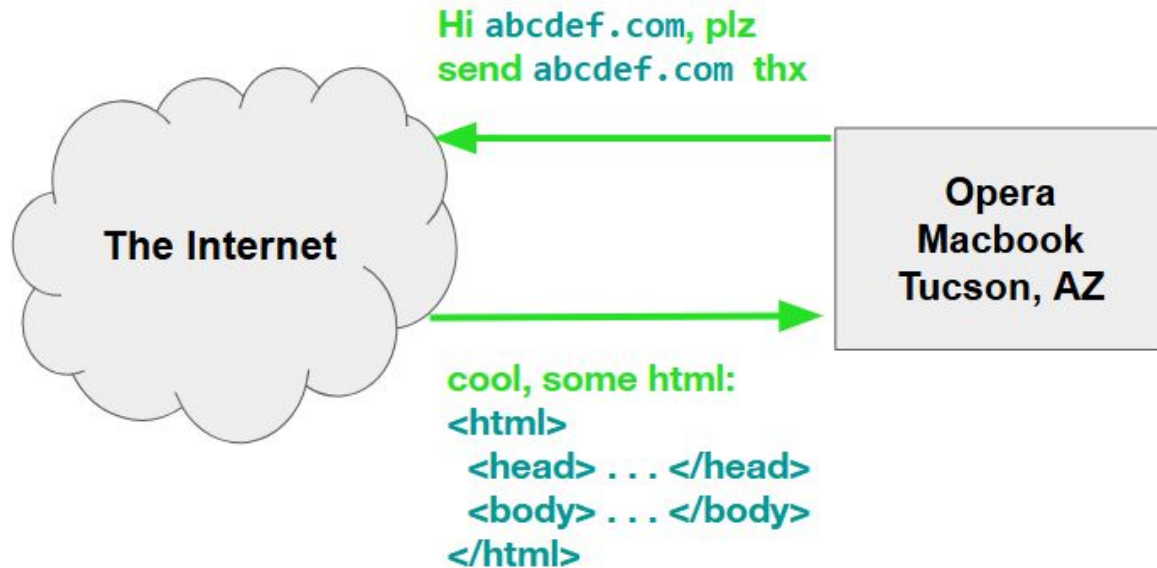
```
element.remove()
```




Introduction to Node.js

CSc 337

HTTP



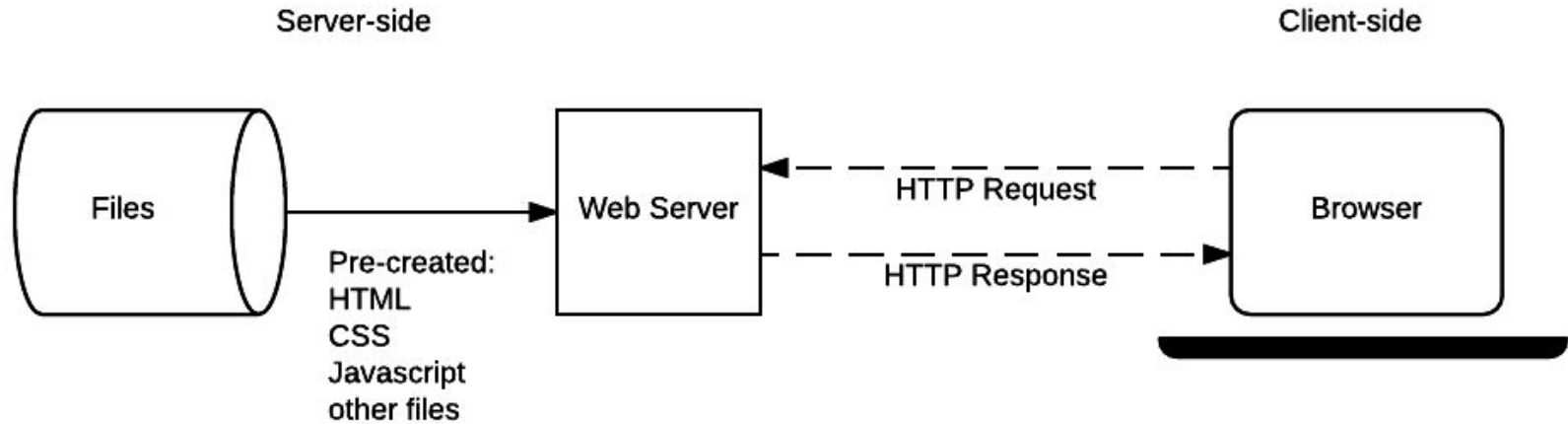
Web servers



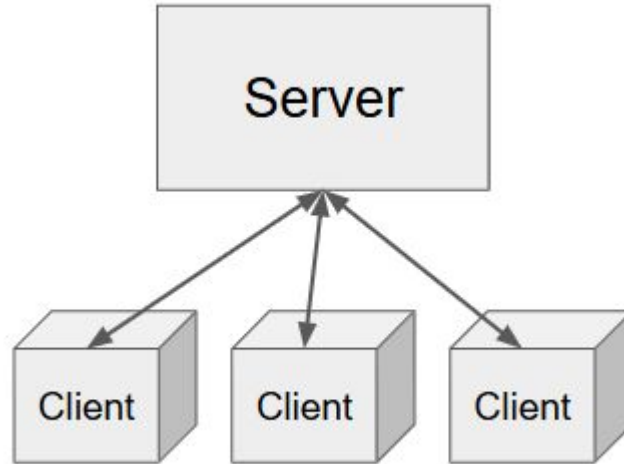
Web servers



Web servers



Web servers





Node.js

- Language: JavaScript
- developed in 2009 by Ryan Dahl
- scalable server-side environment
- Who Uses Node.js?
 - Yahoo!, LinkedIn, eBay, New York Times, Dow Jones, Microsoft



Installing Node.js

- <https://nodejs.org/>
- Download the installer
- Run the installer
- Click the next buttons
- Click the install button
- It should finish in a couple of minutes



Discover TypeScript in Node.js →

Run JavaScript Everywhere

Node.js® is a free, open-source, cross-platform JavaScript runtime environment that lets developers create servers, web apps, command line tools and scripts.

Download Node.js (LTS) 

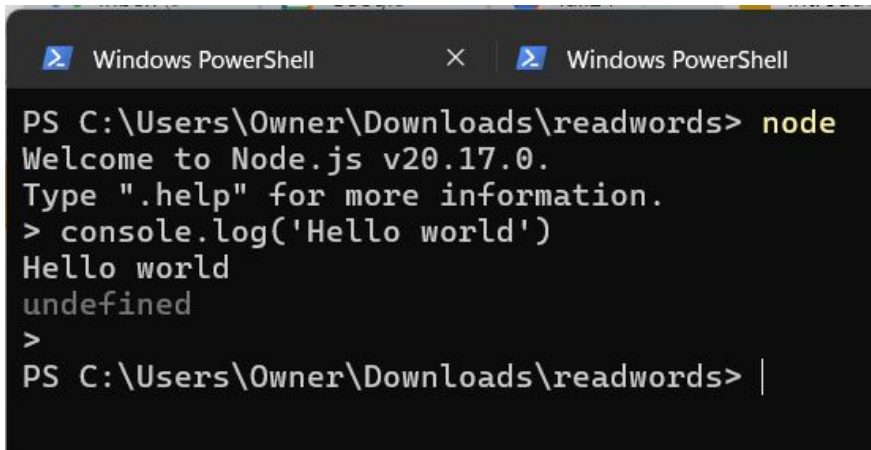
Downloads Node.js **v20.17.0**¹ with long-term support. Node.js can also be installed via package managers.

Want new features sooner? Get **Node.js v22.9.0**¹ instead.



Verify installation

Go to command prompt/terminal, and run 'node'



```
Windows PowerShell X Windows PowerShell
PS C:\Users\Owner\Downloads\readwords> node
Welcome to Node.js v20.17.0.
Type ".help" for more information.
> console.log('Hello world')
Hello world
undefined
>
PS C:\Users\Owner\Downloads\readwords> |
```



server.js

```
var http = require('http');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type':
    'text/plain'});
  res.end('Hello World!');
}).listen(8080);
```



HTML server

The content type attribute:

- “text/plain” will send the information as plain text
- “text/html” for HTML contents



HTML server

Create a html page in server with:

- A heading
- A paragraph
- A division



Getting the URL

```
var http = require("http")

http.createServer(function (req, res){
  res.writeHead(200, {"Content-Type": "text/plain"})
  if(req.url === "/index.html")
  {
    res.end("You are looking for the index page")
  }
  else if(req.url === "/about.html")
  {
    res.end("You are looking for the about page")
  }
}
```

```
  else
  {
    res.end("You are looking for an unknown page")
  }
}).listen(8080)

console.log("Server started ...")
```



Getting the URL

Sometimes it is helpful to check for a substring:

- We can use the includes method
- `"index.html".includes("index")` returns true
- `"index.html".includes("about")` returns false



Parsing the URL

We can use the the url module

- We can use the `url.parse((urlString, [parseQueryString])`
 - `urlString`: The URL string that you want to parse.
 - `parseQueryString`: A boolean that determines whether the query component of the URL should be parsed into an object



Parsing the URL

```
var http = require("http")
var url = require("url")

http.createServer(function (req, res){
  var parsedUrl = url.parse(req.url)
  var keys = Object.keys(parsedUrl)
  res.writeHead(200, {"Content-Type":"text/plain"})
  res.end("keys:" + keys + "\n" + parsedUrl.query)
}).listen(8080)

console.log("Server started ...")
```




Get method

```
var http = require("http")
var url = require("url")

http.createServer(function (req, res){
  if(req.url.includes("index")){
    res.writeHead(200, {"Content-Type":"text/html"})
    res.end(`
<!DOCTYPE html>
<html>
  <head>
    <title>A simple html</title>
  </head>
```

```
<body>
  <form method="GET" action="action.html">
    Name: <input type="text" name="name"><br>
    Email: <input type="email" name="email"><br>
    <input type="submit">
  </form>
</body>
</html>
  `)
}
```



Get method

```
else{
  var parsedUrl = url.parse(req.url, true)
  var query = parsedUrl.query
  res.writeHead(200, {"Content-Type": "text/plain"})
  res.end(`You have submitted
    Name = ${query.name}
    Email = ${query.email}`)
}
}).listen(8080)

console.log("Server started ...")
```



Post method

Don't use get method when:

- You want to submit sensitive information such as:
 - Password
 - credit card number
 - personal details
- In those cases use the post method



Post method

Most of the time the server will know what method is being used.

Also, the `req.method` attribute provides that information:

- It contains “GET” if get method is used
- Similarly, it contains “POST” if post method is used



Post method

```
var http = require("http")
var url = require("url")
var qs = require("querystring")

http.createServer(function (req, res){
  if(req.url.includes("index")){
    res.writeHead(200, {"Content-Type":"text/html"})
    res.end(`
<!DOCTYPE html>
<html>
  <head>
    <title>A simple html</title>
  </head>
```

```
<body>
  <form method="POST" action="action.html">
    Name: <input type="text" name="name"><br>
    Email: <input type="email" name="email"><br>
    Password: <input type="password"
name="password"><br>
    <input type="submit">
  </form>
</body>
</html>
`)
}
```



Post method

```
else if(req.url.includes("action")){
  if(req.method=="POST"){
    var body = ""
    req.on("data", function (chunk){
      body += chunk
    })

    req.on("end", function (){
      res.writeHead(200,
{"Content-Type":"text/plain"})
      var query = qs.parse(body)
```

```
      res.end(`
The data is parsed using querystring module.
Name : ${query.name}
Email : ${query.email}
      `)
    })
  }
}
}).listen(8080)

console.log("Server started ...")
```



University Course Enrollment System

- A simple web-based system where admins and students interact with each other for course management.
- Two main parts:
 - User Management: Create and manage users (admin, students).
 - Course Management: Add, view, and enroll in courses for a semester.



User Types: Admin and Student

- Admin:
 - Single Admin: There will be only one admin for the entire system.
 - Role: The admin can manage the courses and view the list of students enrolled in each course.
- Student:
 - Multiple Students: Many students can be registered.
 - Role: Students can view the available courses and enroll in them (if they haven't already enrolled).



Admin Can Manage Courses

- View Courses: Admin will be able to see all the courses available for the current semester.
- Add New Courses: Admin can create and add new courses to the system for the semester.
 - Course details might include:
 - Course Name
 - Course Code
 - Course Description
 - Credits
 - Semester
- Edit/Remove Courses (optional): Admin may have the ability to modify or delete existing courses.



Student Can Enroll in Courses

- View Courses: Students can view a list of all available courses.
- Enroll in Courses: Students can enroll in any course that is not already filled or that they haven't already enrolled in.
 - Once enrolled, students will see their name listed under the course.
 - A student can only enroll in a course once and cannot enroll in the same course multiple times.



JavaScript Fetch API

CSc 337



Overview

- The Fetch API provides a modern, promise-based way to make asynchronous HTTP requests.
- It allows you to fetch resources like data from a server, and it's a more flexible and powerful alternative to the older XMLHttpRequest.



Key Features

- Promise-based: Simplifies asynchronous code with `.then()` and `.catch()` or `async/await`.
- Supports various methods: GET, POST, PUT, DELETE, etc.
- Handles JSON, text, and other response types.



Basic Syntax:

```
fetch(url, options)
  .then(response => response.json()) // or .text(), .blob(), etc.
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```



Example:

```
// Fetching data from an API
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.error('There was a problem with the fetch operation:', error));
```



Options:

- method: HTTP request method (GET, POST, etc.)
- headers: Headers to include in the request.
- body: Data to send with the request (usually for POST or PUT).



File system

CSc 337

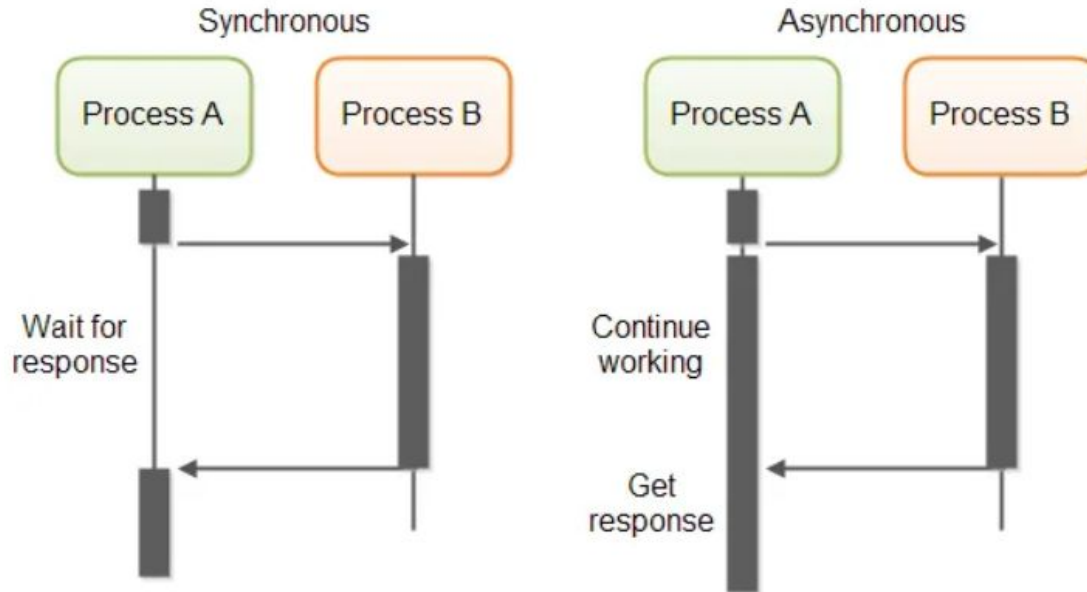


File system

For all the file system calls discussed in this chapter, you need to have loaded the fs module, for example:

```
var fs = require('fs');
```

File system





File system

The fs module provided in Node.js makes almost all functionality available in two forms: asynchronous and synchronous. For example, there is the asynchronous form:

```
write()
```

and the synchronous form:

```
writeSync()
```



File system

Asynchronous	Synchronous
Require a callback function	No callback
Exceptions are automatically handled	handled by your own try/catch blocks
placed on the event queue	run immediately



Opening and Closing Files

To open files in a Node.js app, use one of the following statements for asynchronous or synchronous:

```
fs.open(path, flags, [mode], callback)
```

```
fs.openSync(path, flags, [mode])
```



Opening and Closing Files

Flag	Description
r	Open file for reading.
r+	Open file for reading and writing.
w	Open file for writing. The file is created if it does not exist or truncated if it does exist.
wx	Same as w but fails if the path exists.



Opening and Closing Files

Once a file has been opened, you need to close it to force flushing changes to disk and release the OS lock.

```
fs.close(fd, callback)
```

```
fs.closeSync(fd)
```




Opening and Closing Files

```
fs.open("myFile", 'w', function(err, fd){  
  if (!err){  
    fs.close(fd);  
  }  
});
```



Opening and Closing Files

```
var fd = fs.openSync("myFile", 'w');  
fs.closeSync(fd);
```



Opening and Closing Files

```
try {  
  var fd = fs.openSync("myFile", 'w');  
  fs.closeSync(fd);  
} catch (error) {  
  console.error('An error occurred:', error);  
}
```



Writing Files

The following shows the syntax for the writeFile() methods:

```
fs.writeFile(path, data, [options], callback)
```

```
fs.writeFileSync(path, data, [options])
```



file_write.js

```
var fs = require('fs');
var config = {
  maxFiles: 20,
  maxConnections: 15,
  rootPath: "/webroot"
};
var configTxt = JSON.stringify(config);
var options = {encoding:'utf8', flag:'w'};
fs.writeFile('config.txt', configTxt, options,
function(err){
  if (err){
    console.log("Config Write Failed.");
  } else {
    console.log("Config Saved.");
  }
});
```



file_read.js

```
var fs = require('fs');
var options = { encoding: 'utf8', flag: 'r' };
fs.readFile('config.txt', options, function(err,
data) {
  if (err) {
    console.log("Config Read Failed.");
  } else {
    try {
      var config = JSON.parse(data);
      console.log("Config Loaded:", config);
    } catch (parseErr) {
      console.log("Failed to parse config.");
    }
  }
});
```



file_append.js

```
var fs = require('fs')

fs.appendFile('fruits.txt', 'apple',
{encoding:'utf8'}, err => {
  if(err){
    console.log(err)
  }
  else{
    console.log('Modified the file.')
  }
})
```



file_write_sync.js

```
var fs = require('fs');
var config = {
  maxFiles: 20,
  maxConnections: 15,
  rootPath: "/webroot"
};
var configTxt = JSON.stringify(config);
var options = { encoding: 'utf8', flag: 'w' };
try {
  fs.writeFileSync('config.txt', configTxt,
    options);
  console.log("Config Saved.");
} catch (err) {
  console.log("Config Write Failed.");
}
```




file_read_sync.js

```
var fs = require('fs');

var options = { encoding: 'utf8', flag: 'r' };

try {
  var data = fs.readFileSync('config.txt',
options);
  var config = JSON.parse(data);
  console.log("Config Loaded:", config);
} catch (err) {
  console.log("Config Read Failed.");
}
```



file_append_sync.js

```
var fs = require('fs')  
  
fs.appendFileSync('fruits.txt', '\nfigs', {encoding:'utf8'})
```



University Course Enrollment System

- A simple web-based system where admins and students interact with each other for course management.
- Two main parts:
 - User Management: Create and manage users (admin, students).
 - Course Management: Add, view, and enroll in courses for a semester.



User Types: Admin and Student

- Admin:
 - Single Admin: There will be only one admin for the entire system.
 - Role: The admin can manage the courses and view the list of students enrolled in each course.
- Student:
 - Multiple Students: Many students can be registered.
 - Role: Students can view the available courses and enroll in them (if they haven't already enrolled).



Admin Can Manage Courses

- View Courses: Admin will be able to see all the courses available for the current semester.
- Add New Courses: Admin can create and add new courses to the system for the semester.
 - Course details might include:
 - Course Name
 - Course Code
 - Course Description
 - Credits
 - Semester
- Edit/Remove Courses (optional): Admin may have the ability to modify or delete existing courses.



Student Can Enroll in Courses

- View Courses: Students can view a list of all available courses.
- Enroll in Courses: Students can enroll in any course that is not already filled or that they haven't already enrolled in.
 - Once enrolled, students will see their name listed under the course.
 - A student can only enroll in a course once and cannot enroll in the same course multiple times.



University Course Enrollment System

We have built this website, but there were several drawbacks:

- There were html codes written in the server
- Once the server terminates, then all information are gone

We will now fix these issues.



Introduction to Crypto

CSc 337



Crypto Module in Node.js

- The crypto module provides cryptographic functionality that includes a set of wrappers for OpenSSL's hash, HMAC, cipher, decipher, and signature functions.
- Commonly used for hashing, encryption, and generating secure random values.



createHash()

- createHash() is a method in the crypto module used to create a hash object.
- It allows you to specify a hash algorithm (e.g., sha256, md5).
- Example usage: `const hash = crypto.createHash('sha256');`



SHA-256

- SHA-256 (Secure Hash Algorithm 256-bit) is a cryptographic hash function that produces a 256-bit hash value (64 characters in hexadecimal).
- It is widely used for generating unique fingerprints of data, such as password hashing or data integrity checks.



digest()

- After calling `update()` on the hash object with data (e.g., a password), `digest()` is used to finalize the hash and return the result.
- `digest()` returns the hash in the specified encoding (e.g., hex, base64).
- Example usage: `const hashedPassword = hash.update('password').digest('hex');`

CSc 337

Node and Express

Reyan Ahmed

What is express ?



- Node package that can be used to build web servers
- Utilizes callbacks
 - (as with node itself)
- Other modules too:
<https://nodejs.dev/en/learn/>

Routes



- Routes are a fundamental concept of express
- Basically, gives the developer easy way to control the server response based on the PATH portion of the requested URL

Responding differently to:

```
http://domainname.com/applications  
http://domainname.com/projects  
http://domainname.com/contracts  
... etc ...
```

A basic server



```
const express = require('express')
```

```
const app = express()
```

```
const port = 3000
```

```
app.get('/', (req, res) => res.send('Some response'))
```

```
app.listen(port, () =>
```

```
  console.log(`Example app listening at http://localhost:${port}`))
```


Specifying Routes



The way to specify routes in code is:

```
app.METHOD(PATH, ACTION);
```

- **METHOD** - the http method used (**get**, **post**, etc)
- **PATH** - string (or regex!) of path(s) to match from URL
- **ACTION** - the function to call when there is a route match. Function should have two params, request info and response

Math server

- Implement a simple express server that can do basic math calculations (add, subtract, divide, multiply)
- Should accept URLs of the form:
`/calculate/:operation/:num1/:num2`
- Should perform the **operation** using **num1** and **num2** and respond with the resulting value
- For example:
`/calculate/add/50/30 -> 80`
`/calculate/subtract/50/50 -> 0`



Introduction to MongoDB

CSc 337

What is a database?

An organized collection of structured data, typically stored electronically, that allows for efficient storage, retrieval, and management of information.





Benefits of database

- Reduced data redundancy
- Data security
- Data sharing
- Backup and recovery facilities
- Improved data access
- Access for multiple users
- Scalability
- ...



What is special about MongoDB?

Flexible: it uses a document-oriented data model, allowing documents within a collection to have varying structures and data types, without requiring a predefined schema.

Scalable: has a full range of vertical, horizontal, and elastic scaling options available.

Powerful NoSQL document database: can handle big data faster.



Who uses MongoDB?





Installing MongoDB

<https://www.mongodb.com/>

www.mongodb.com/try/download/community



MongoDB

<https://www.mongodb.com>

MongoDB: The World's Leading Modern Database | MongoDB

MongoDB Atlas integrates operational and vector databases in a single, unified platform. Use vector representations of your data to perform semantic search, ...

Log in to your account

Log in to your account. Don't have an account? Sign Up. Google ...



Community Server

MongoDB Community Server Download. The Community ...



Atlas Database

MongoDB Atlas is a fully managed cloud database service that ...



Install MongoDB

Install MongoDB ... MongoDB is available in two server editions ...



MongoDB Documentation

Welcome to the official MongoDB Documentation. Whether you're ...



[More results from mongodb.com »](#)



Installing MongoDB

<https://www.mongodb.com/>

www.mongodb.com/try/download/community

Version

8.0.6 (current)



Platform

Windows x64



Package

msi



Download



Copy link

More Options





Installing MongoDB Shell

The MongoDB Shell (mongosh) may not get installed with MongoDB Server.

How to check?

- Open a command-line interface or terminal.
- Run the mongo command to open the MongoDB shell.
- Once in the MongoDB shell, run the `db.version()` command. This will display the version of the MongoDB server you are connected to.



MongoDB Shell

mongodb.com/try/download/shell

Note: MongoDB Shell is an open source (Apache 2.0), standalone product developed separately from the MongoDB Server.

[Learn more](#)

Version

2.4.2



Platform

Windows x64 (10+)



Package

zip



Download



Copy link

More Options





User accounts

Overview

- Purpose: Manage database access and ensure security.
- Importance: Helps in controlling who can access or modify data.



User accounts

User Roles

- Built-in Roles:
 - read: Allows read access to the database.
 - readWrite: Allows read and write access.
 - dbAdmin: Grants administrative privileges on the database.
 - userAdmin: Allows management of users and roles.



Listing Users

The following commands show changing to the admin database and listing users:

```
use admin
```

```
show users
```



Listing Users

Understanding the Cursor Object in MongoDB

- Definition: A cursor is an object that allows you to iterate over the result set of a query.
- Functionality: Provides a way to access documents returned by a database query one at a time.

```
use admin
```

```
cur = db.system.users.find()
```

```
cur.count()
```



Creating user accounts

To create a user you should switch to that database and then use the `createUser()` method to create the user object:

```
use testDB
```

```
db.createUser( { user: "testUser",
```

```
  pwd: "test",
```

```
  roles: [ "readWrite", "dbAdmin" ] })
```




Removing user accounts

To remove the testUser user from the testDB database, use the following commands from the MongoDB shell:

```
use admin
```

```
db.dropUser("testUser")
```



Creating a User Administrator Account

The User Administrator account should be created with `userAdminAnyDatabase` as the only role. This gives the User Administrator the ability to create new user accounts but not to manipulate the database beyond that.

```
use admin
```

```
db.createUser( { user: "useradmin",
```

```
  pwd: "test",
```

```
  roles: [ "userAdminAnyDatabase" ] } )
```



Creating a Database Administrator Account

Creating a Database Administrator account is done by executing the `createuser` method in the MongoDB shell to access the admin database and then adding a user with `readWriteAnyDatabase`, `dbAdminAnyDatabase`, and `clusterAdmin` rights.

```
use admin
```

```
db.createUser( { user: "dbadmin",
```

```
  pwd: "test",
```

```
  roles: [ "readWriteAnyDatabase", "dbAdminAnyDatabase", "clusterAdmin" ] } )
```



Creating a Database Administrator Account

Once you have created the new administrator account, you can authenticate as that user using the following commands:

```
use admin
```

```
db.auth("dbadmin", "test")
```



Creating a Database Administrator Account

You can also authenticate to the admin database as the Database Administrator when starting the MongoDB shell using the `--username` and `--password` options, for example:

```
mongosh --username "dbadmin" --password "test"
```



Administering Databases

Displaying a List of Databases:

```
show dbs
```



Administering Databases

Changing the Current Database:

```
db = db.getSiblingDB('testDB')
```

Or use the following command:

```
use testDB
```



Administering Databases

Creating Databases:

```
use newDB
```

```
db.createCollection("newCollection")
```




Administering Databases

To delete a database from the MongoDB shell, use the `dropDatabase()` method.

```
use newDB
```

```
db.dropDatabase()
```



Managing Collections

Displaying a list of collections in a database:

```
use testDB
```

```
show collections
```



Managing Collections

Creating collections:

```
use testDB
```

```
db.createCollection("newCollection")
```



Managing Collections

Deleting collections:

```
use testDB
```

```
coll = db.getCollection("newCollection")
```

```
coll.drop()
```



Managing Collections

Finding documents in a collection:

```
db.users.find()
```

Or

```
db.getCollection("users").find()
```

Or

```
coll = db.getCollection("users");coll.find()
```



Managing Collections

Adding documents to a collection:

```
coll = db.getCollection("newCollection")
```

```
coll.insert({ vehicle: "plane", speed: "480mph" })
```

```
coll.insert({ vehicle: "car", speed: "120mph" })
```

```
coll.insert({ vehicle: "train", speed: "120mph" })
```



Managing Collections

Finding documents in a collection with specific field value:

```
coll.find({speed:'120mph'})
```



Managing Collections

Similarly, we can remove documents in a collection with specific field value:

```
coll.remove({'vehicle':'car'})
```




Managing Collections

If you want to update the first matched document, then use updateOne:

```
coll.updateOne(  
  { speed: "150mph" },  
  { $set: { speed: "120mph", updated: true } }  
);
```



Managing Collections

If you want to update all documents, then use updateMany:

```
coll.updateMany(  
  { speed: "120mph" },  
  { $set: { speed: "150mph", updated: true } }  
);
```



MongoDB and Node.js

CSc 337



Adding the MongoDB Driver to Node.js

From your project root directory, execute the following command using a console prompt:

```
npm install mongodb
```



Using MongoClient Class in MongoDB with Node.js

MongoClient Class Overview:

- MongoClient is the primary class used to connect to a MongoDB server in Node.js.
- It is part of the official mongodb Node.js driver.



Using MongoClient Class in MongoDB with Node.js

Key Methods:

- `connect()`
 - Purpose: Establishes a connection to the MongoDB server.
 - Usage:
 - Connects to a MongoDB server and returns a Promise.
 - Can be used with `async/await` or `.then()`.



Using MongoClient Class in MongoDB with Node.js

```
const { MongoClient } = require('mongodb');
const uri = 'mongodb://localhost:27017';
const client = new MongoClient(uri);

client.connect()
  .then(() => {
    console.log('Connected to MongoDB');
  })
  .catch((error) => {
    console.error('Connection failed', error);
  });
```



Using MongoClient Class in MongoDB with Node.js

Key Methods:

- `close()`
 - Purpose: Closes the connection to the MongoDB server.
 - Usage:
 - Called after all database operations are complete.
 - Returns a Promise, ensuring that the connection closes properly.



Using MongoClient Class in MongoDB with Node.js

```
client.connect()
  .then(() => {
    // Do some work with the database
    console.log('Connected to MongoDB');
    return client.close();
  })
  .then(() => {
    console.log('Connection closed');
  })
  .catch((error) => {
    console.error('Error during connection or
closing', error);
  });
```



Using MongoClient Class in MongoDB with Node.js

Key Methods:

- Connection Lifecycle:
 - `connect()`: Establish a connection.
 - `close()`: Cleanly disconnect when done.
 - Promises: Both methods return promises, which allow for chaining with `.then()` or handling errors with `.catch()`.
 - async/await Alternative:
 - You can also use `async/await` to work with these methods in an asynchronous function.



Using MongoClient Class in MongoDB with Node.js

```
async function connect() {  
  const client = new MongoClient(uri);  
  try {  
    await client.connect();  
    console.log('Connected to MongoDB');  
  } catch (error) {  
    console.error('Error connecting to MongoDB', error);  
  } finally {  
    await client.close();  
    console.log('Connection closed');  
  }  
}  
connect();
```



MongoClient.db() Method in MongoDB

Overview:

- MongoClient.db() is a method used to select a database from a connected MongoDB client instance.
- It's essential to access collections and perform operations on the MongoDB database.
- The method does not create the database immediately. The database is only created when you first insert data into it.



MongoClient.db() Method in MongoDB

Syntax:

```
db = client.db([dbName], [options]);
```

- dbName (optional): The name of the database you want to select.
- options (optional): An object containing additional options like readPreference, authSource, etc.



MongoClient.db() Method in MongoDB

Basic Usage:

- Without Database Name: If you don't pass a database name, it will return the database to which you've already connected.
 - `const db = client.db(); // Uses the default database`
- With Database Name: If you specify the database name, it will select that particular database.
 - `const db = client.db('mydatabase'); // Selects 'mydatabase'`



MongoClient.db() Method in MongoDB

```
async function connectToDatabase() {  
  try {  
    await client.connect();  
    console.log('Connected to MongoDB');  
    const db = client.db('testDB');  
    console.log('Database selected:', db.databaseName);  
    const collection = db.collection('users');  
    console.log('Collection selected:', collection.collectionName);  
  } catch (error) {  
    console.error('Error:', error);  
  } finally {  
    await client.close();  
    console.log('Connection closed');  
  }  
}
```



MongoClient.db() Method in MongoDB

Database Selection:

- Use `client.db('databaseName')` to specify the database you want to work with.
- If no name is passed, MongoDB uses the default database connection.

Database Creation:

- The database is not physically created until you insert data or perform an operation that requires it.

Collection Access:

- After selecting the database, you can access collections using `db.collection('collectionName')`.



MongoClient.db() Method in MongoDB

After selecting the database using `db()`, you can perform CRUD operations on collections:

```
const usersCollection = db.collection('users');
await usersCollection.insertOne({ name: 'Alice', age: 30 });
console.log('User inserted');

// Find a document in the collection
const user = await usersCollection.findOne({ name: 'Alice' });
console.log('User found:', user);
```



collection.insertOne()

```
async function dbInsertOne(){
  try{
    await client.connect()
    var db = client.db('testDB')
    var coll = db.collection('newCollection')
    await coll.insertOne({'courseID':'CSc 337', 'courseName':'Web
programming', 'desc':'In this course we will learn HTML, CSS,
javascript and node.js.'})
    await client.close()
  }
  catch(err){
    console.log(err)
  }
}
```



collection.insertOne()

```
client.connect()
.then(function(){
  var db = client.db('testDB')
  var coll = db.collection('newCollection')
  return coll.insertOne({'courseID':'CSc 337', 'courseName':'Web
programming', 'desc':'In this course we will learn HTML, CSS,
javascript and node.js.})
})
.then(function(){
  console.log('Insertion complete.')
})
.catch(function(err)
{
  console.log(err)
})
```



collection.insertMany()

```
client.connect()
.then(function(){
  var db = client.db('testDB')
  var coll = db.collection('newCollection')
  var courses = [
    {'courseID':'CSc 337', 'courseName':'Web programming',
    'desc':'In this course we will learn HTML, CSS, javascript and
    node.js.'},
    {'courseID':'CSc 110', 'courseName':'intro prog 1',
    'desc':'variable, list, dict, for, while ...'},
    {'courseID':'CSc 120', 'courseName':'intro prog 2', 'desc':'class,
    references, linked list, compexity ...'}
  ]
  return coll.insertMany(courses)
})
```



collection.insertMany()

```
async function dbInsertMany(){
  await client.connect()
  var db = client.db('testDB')
  var coll = db.collection('newCollection')
  var courses = [
    {'courseID':'CSc 337', 'courseName':'Web programming',
    'desc':'In this course we will learn HTML, CSS, javascript and
    node.js.'},
    ...
    {'courseID':'CSc 120', 'courseName':'intro prog 2', 'desc':'class,
    references, linked list, complexity ...'}
  ]
  await coll.insertMany(courses)
  await client.close()
}
```



collection.findOne()

```
client.connect()
.then(function(){
  var db = client.db('testDB')
  var coll = db.collection('newCollection')
  return coll.findOne({'vehicle':'plane'})
})
.then(function(document){
  console.log(document)
})
.catch(function(err){
  console.log(err)
})
.finally(function(){
  client.close()
})
```



collection.findOne()

```
async function dbFindOne() {  
  await client.connect()  
  var db = client.db('testDB')  
  var coll = db.collection('newCollection')  
  var doc = await coll.findOne({'vehicle':'plane'})  
  console.log(doc)  
  await client.close()  
}
```



collection.find()

```
client.connect()
.then(function(){
  var db = client.db('testDB')
  var coll = db.collection('newCollection')
  return coll.find({}).toArray()
})
.then(function(documents){
  console.log(documents)
})
.catch(function(err){
  console.log(err)
})
.finally(function(){
  client.close()
})
```




collection.find()

```
async function dbFunction()
{
  try{
    await client.connect()
    console.log('Connection open now.')
    var db = client.db('testDB')
    var coll = db.collection('newCollection')
    var docs = await coll.find({}).toArray()
    console.log(docs)
    await client.close()
  }catch(err)
  {
    console.log(err)
  }
}
```