

# Introduction to HTML, Javascript, PostgreSQL and PostGIS

Presented By

Siddhartha Bhuyan

[siddhartha.bhuyan1@nesac.gov.in](mailto:siddhartha.bhuyan1@nesac.gov.in)

Contact No.: 03642308734

Mobile: 9085995599

**HTML**

# What is HTML?

- Stands for Hyper Text Markup Language
- Standard markup language for creating Web pages
- Describes the structure of a Web page
- Consists of a series of elements
- Elements tell the browser how to display the content
- Elements label pieces of content such as "this is a heading", "this is a paragraph", "this is a link", etc.

# History of HTML

1989	Tim Berners-Lee invented www
1991	Tim Berners-Lee invented HTML
1993	Dave Raggett drafted HTML+
1995	HTML Working Group defined HTML 2.0
1997	W3C Recommendation: HTML 3.2
1999	W3C Recommendation: HTML 4.01
2000	W3C Recommendation: XHTML 1.0
2008	WHATWG HTML5 First Public Draft
2012	<u>WHATWG HTML5 Living Standard</u>
2014	<u>W3C Recommendation: HTML5</u>
2016	W3C Candidate Recommendation: HTML 5.1
2017	<u>W3C Recommendation: HTML5.1 2nd Edition</u>
2017	<u>W3C Recommendation: HTML5.2</u>

# A Simple HTML Document

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

# Example Explained

- The `<!DOCTYPE html>` declaration defines that this document is an HTML5 document
- The `<html>` element is the root element of an HTML page
- The `<head>` element contains meta information about the HTML page
- The `<title>` element specifies a title for the HTML page (which is shown in the browser's title bar or in the page's tab)
- The `<body>` element defines the document's body, and is a container for all the visible contents, such as headings, paragraphs, images, hyperlinks, tables, lists, etc.
- The `<h1>` element defines a large heading
- The `<p>` element defines a paragraph

# The HTML Element

An HTML element is defined by a start tag, some content, and an end tag:

`<tagname> Content goes here... </tagname>`

The HTML element is everything from the start tag to the end tag:

`<h1>My First Heading</h1>`

`<p>My first paragraph.</p>`

# Nesting Elements

Elements can be placed within other elements. This is called *nesting*. If we wanted to state that a cat is **very** grumpy, we could wrap the word *very* in a <strong> element, which means that the word is to have strong(er) text formatting:

```
<p>My cat is <strong>very</strong> grumpy.</p>
```

My cat is **very** grumpy.

\*IMP: The tags have to open and close in a way that they are inside or outside one another.

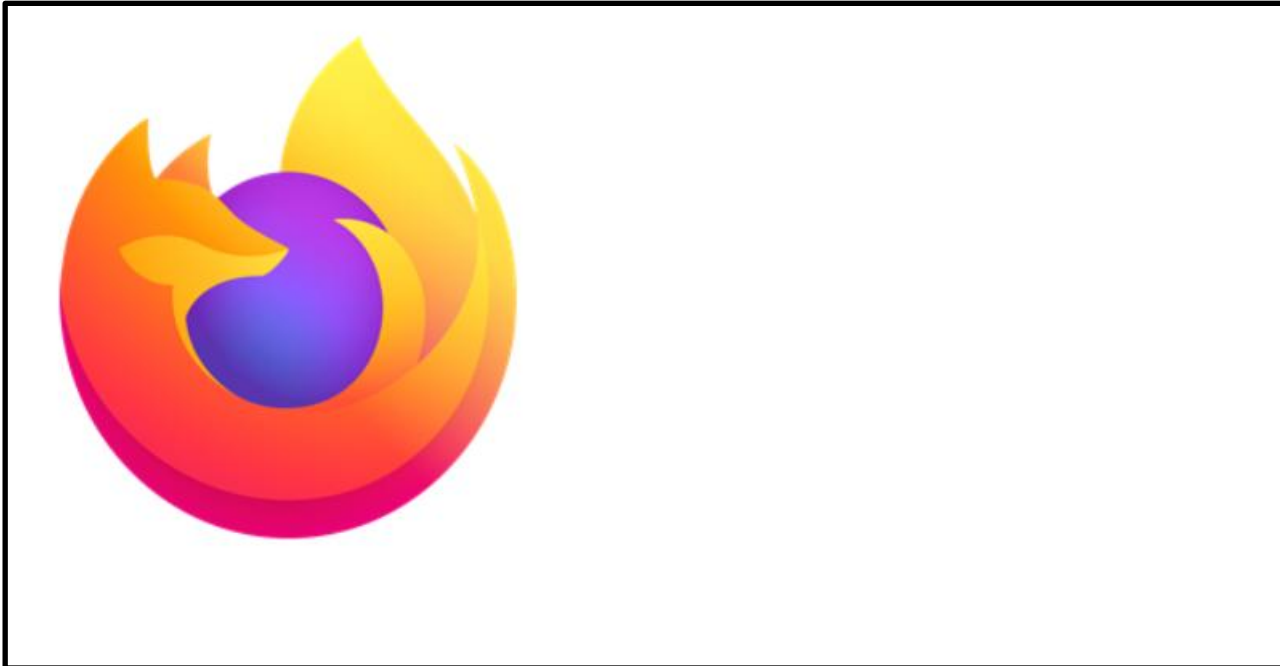


# Void Elements

Some elements consist of a single tag, which is typically used to insert/embed something in the document. Such elements are called [void elements](#). For example, the [<img>](#) element embeds an image file onto a page:

```

```



# Attributes

Elements can also have attributes. Attributes look like this:



A diagram illustrating an HTML element with an attribute. The text `<p class="editor-note">My cat is very grumpy</p>` is shown. A bracket above the `class="editor-note"` portion is labeled "Attribute".

Attributes contain extra information about the element that won't appear in the content. In this example, the **class** attribute is an identifying name used to target the element with style information.

# Attributes

An attribute should have:

- A space between it and the element name.  
(For an element with more than one attribute, the attributes should be separated by spaces too.)
- The attribute name, followed by an equal sign.
- An attribute value, wrapped with opening and closing quote marks.

# Boolean Attributes

Attributes written without values are called Boolean attributes. They can only have one value, which is generally the same as the attribute name. For example, the [disabled](#) attribute:

```
<input type="text" disabled="disabled" />
```

OR

```
<input type="text" disabled />
```

# Single or Double Quotes?

Both are equivalent for example:

```
<a href='https://www.example.com'>A link to my example.</a>
```

```
<a href="https://www.example.com">A link to my example.</a>
```

\*IMP: Don't mix quotes for e.g.

```
href="https://www.example.com'
```

But the following is alright

```
<a href="https://www.example.com" title="Isn't this fun?"> A  
link to my example. </a>
```

What if you wanted to use double quotes inside double quotes?

Use HTML Entities

```
<a href="https://www.example.com" title="An  
&quot;interesting&quot; reference">A link to my example.</a>
```

# Whitespaces in HTML

These two code snippets are equivalent:

```
<p id="noWhitespace">Dogs are silly.</p>
```

```
<p id="whitespace">Dogs
```

```
are                                silly.</p>
```

No matter how much whitespace you use inside HTML element content (which can include one or more space characters, but also line breaks), the HTML parser reduces each sequence of whitespace to a single space when rendering the code. So why have this facility? The answer is readability. Formatting your HTML code is also very important for debugging and code intelligibility

# Entity References: Including Special Characters in HTML

In HTML, the characters `<`, `>`, `"`, `'` and `&` are special characters. They are parts of the HTML syntax itself. So how do you include one of these special characters in your text?

Using Character References – special codes that represent characters, to be used in these exact circumstances. Each character reference starts with an ampersand (`&`), and ends with a semicolon (`;`)

Literal character	Character reference equivalent
<code>&lt;</code>	<code>&amp;lt;</code>
<code>&gt;</code>	<code>&amp;gt;</code>
<code>"</code>	<code>&amp;quot;</code>
<code>'</code>	<code>&amp;apos;</code>
<code>&amp;</code>	<code>&amp;amp;</code>

`<p>In HTML, you define a paragraph using the <p> element.</p>`

`<p>In HTML, you define a paragraph using the &lt;p&gt; element.</p>`

# HTML Comments

To write an HTML comment, wrap it in the special markers `<!--` and `-->`. For example:

```
<p>I'm not inside a comment</p>
```

```
<!-- <p>I am!</p> -->
```





# Other Important Things

- Metadata such as charset, name:content, http-equiv:content
- Script and Link elements
- Style tags
- Primary language and span elements
- Headings and hierarchy (and SEO and screen readers)
- Ordered and unordered lists, nesting of lists
- Emphasis and importance using the `<em>` and `<strong>` tags
- Hyperlinks using the `<a>` tag
- Title attribute in anchor tags
- File hierarchy and anchor tags
- Document fragments
- Absolute and relative URLs
- Download attribute in `<a>` tags

# Other Important Things

- mailto attribute
- Description lists
- Block quotes and inline quotations
- Citations
- Abbreviations
- Address tags
- Superscripts and subscripts
- <code>, <pre>, <var>, <kbd>, <samp> tags
- <time> tag and datetime attribute
- HTML templates and page design
- Markup validation service (W3C) and debugging
- Multimedia and embedding – img, figure, video, audio, iframe, embed, object, SVG
- HTML tables and divs

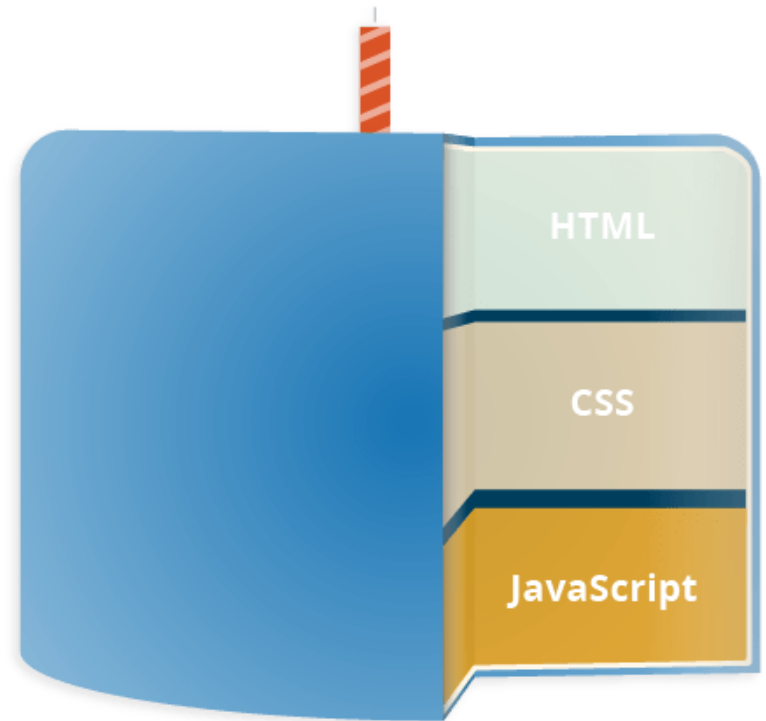
# JavaScript

# What is Javascript?

- A scripting or programming language that allows you to implement complex features on web pages
- Used for displaying timely content updates, interactive maps, animated 2D/3D graphics, scrolling video jukeboxes, etc.
- Can change HTML content, attributes, styles etc. that is brings dynamism to static HTML pages

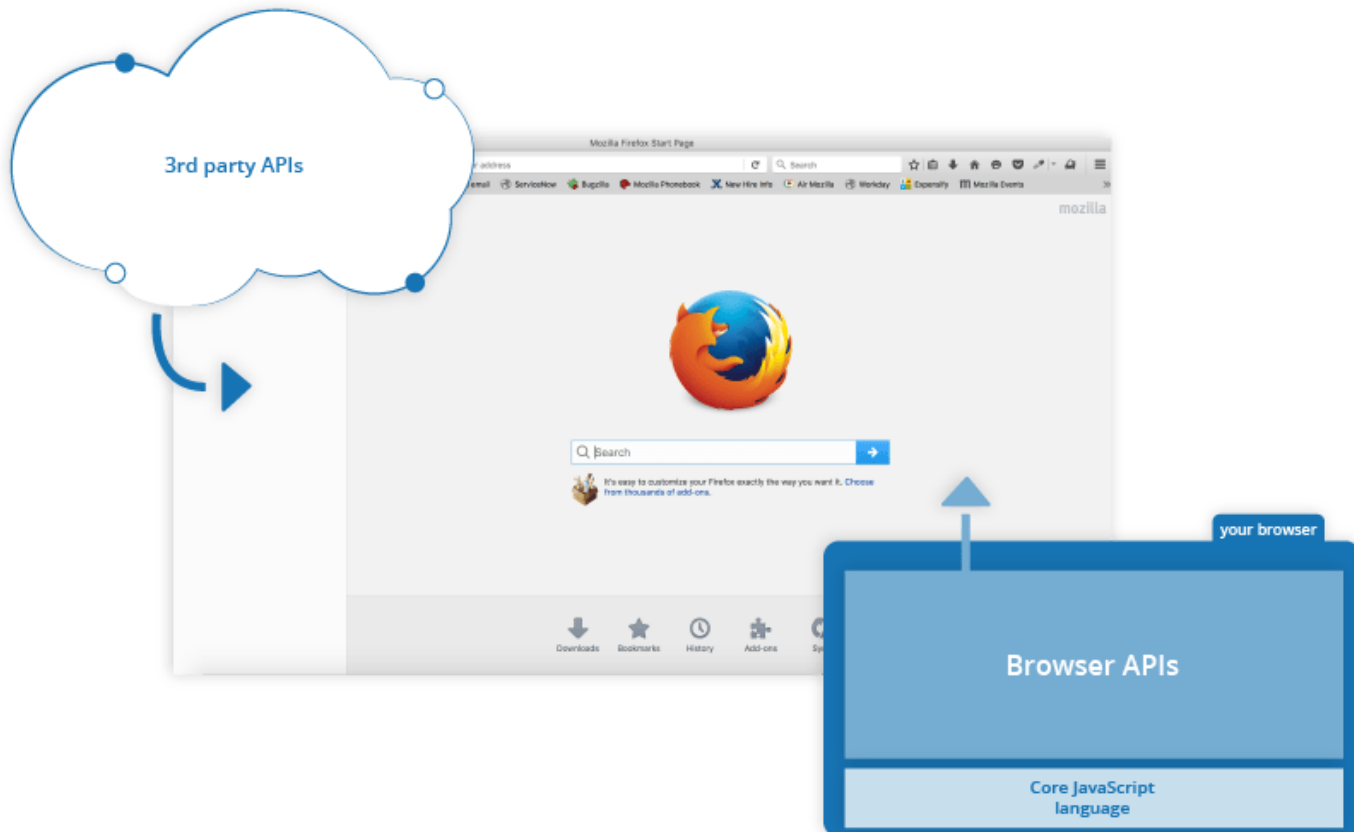
# Role of Javascript

- HTML is the markup language that we use to structure and give meaning to our web content, for example defining paragraphs, headings, and data tables, or embedding images and videos in the page.
- CSS is a language of style rules that we use to apply styling to our HTML content, for example setting background colors and fonts, and laying out our content in multiple columns.
- JavaScript is a scripting language that enables you to create dynamically updating content, control multimedia, animate images, and pretty much everything else.



# APIs and Javascript Functionality

- Application Programming Interfaces or APIs are ready-made sets of code building blocks that allow a developer to implement programs that would otherwise be hard or impossible to implement and fall broadly into two categories



# Browser APIs

These are built into your web browser, and are able to expose data from the surrounding computer environment, or do useful complex things. For example:

- The DOM (Document Object Model) API allows you to manipulate HTML and CSS, creating, removing and changing HTML, dynamically applying new styles to your page, etc.
- The Geolocation API retrieves geographical information.
- The Canvas and WebGL APIs allow you to create animated 2D and 3D graphics.
- Audio and Video APIs like HTMLMediaElement and WebRTC allow you to do really interesting things with multimedia, such as play audio and video right in a web page, or grab video from your web camera and display it on someone else's computer

# Third Party APIs

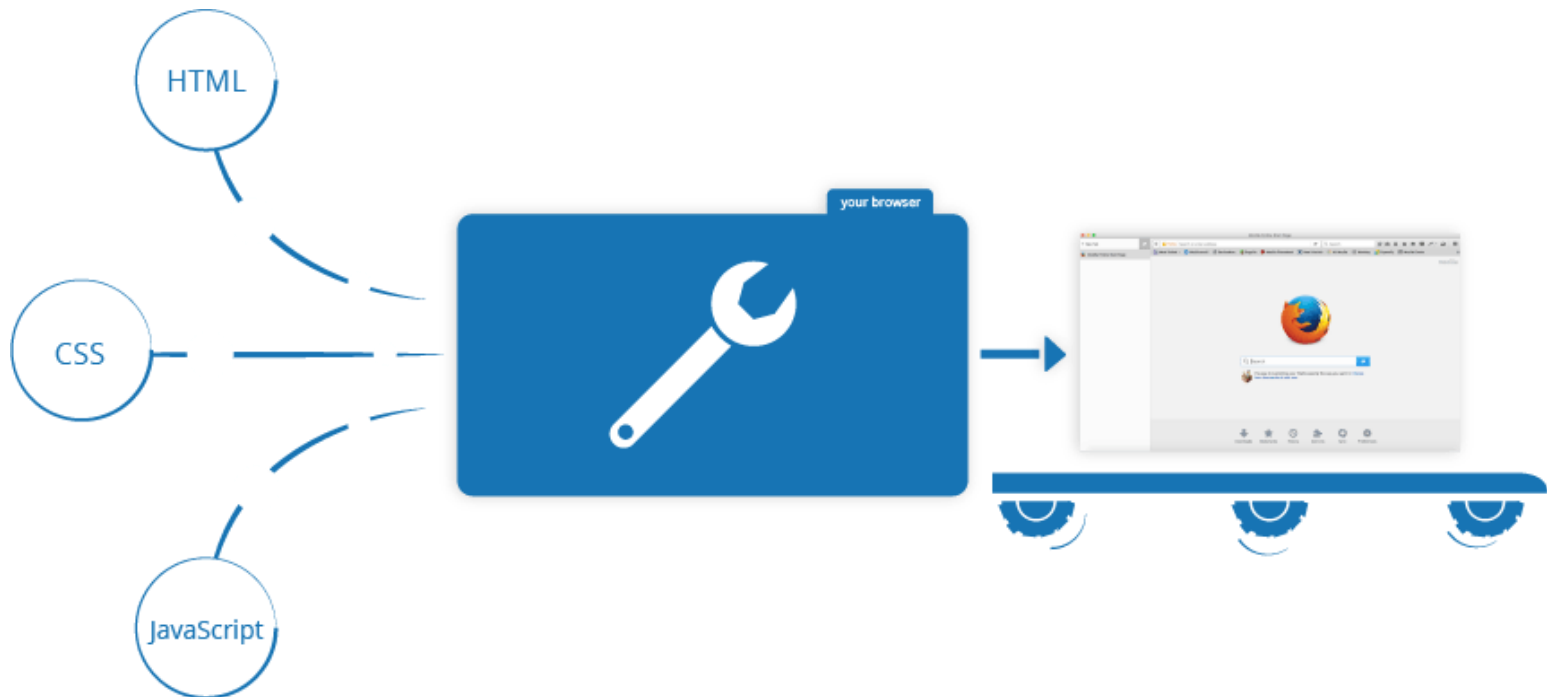
These are not built into the browser by default, and you generally have to grab their code and information from somewhere on the Web. For example:

- The [Twitter API](#) allows you to do things like displaying your latest tweets on your website.
- The [Google Maps API](#) and [OpenStreetMap API](#) allows you to embed custom maps into your website, and other such functionality.



# Loading a Page in the Browser

When you load a web page in your browser, you are running your code (the HTML, CSS, and JavaScript) inside an execution environment (the browser tab). Note that the code in your web documents is generally loaded and executed in the order it appears on the page.



# Using Javascript

Two ways:

- Internal Javascript
- External Javascript

Inline javascript handlers – bad!

JS

```
function createParagraph() {  
  const para = document.createElement("p");  
  para.textContent = "You clicked the button!";  
  document.body.appendChild(para);  
}
```

HTML

```
<button onclick="createParagraph()">Click me!</button>
```

Instead use addEventListener as follows

```
const buttons = document.querySelectorAll("button");  
  
for (const button of buttons) {  
  button.addEventListener("click", createParagraph);  
}
```

# Script Loading

Consider the following

```
document.addEventListener("DOMContentLoaded", () => {  
    // javascript code block  
});
```

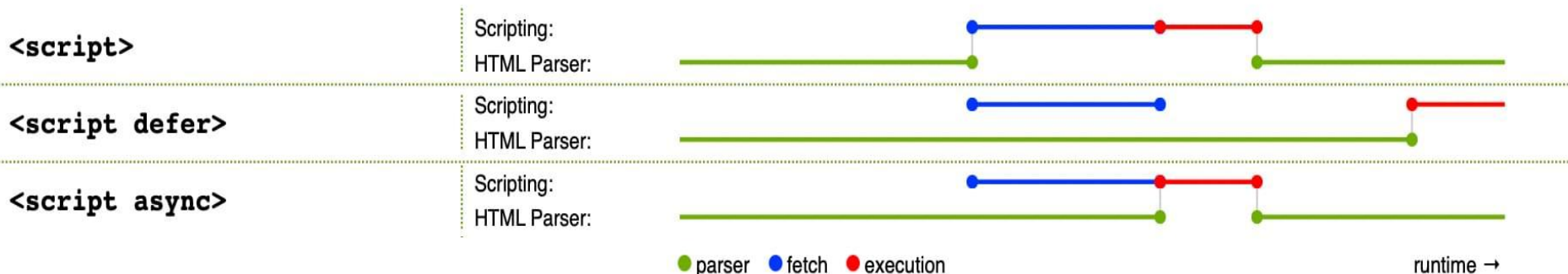
This is an event listener, which listens for the browser's **DOMContentLoaded** event, which signifies that the HTML body is completely loaded and parsed. The JavaScript inside this block will not run until after that event is fired. Can be put towards end of `<body>` but will cause slowness (why?)

# Async and Defer

The defer attribute, which tells the browser to continue downloading the HTML content once the `<script>` tag element has been reached

```
<script src="script.js" defer></script>
```

Using async attribute will download the script without blocking the page while the script is being fetched. However, once the download is complete, the script will execute, which blocks the page from rendering. Scripts won't run in any specific order. Use async when the scripts in the page run independently from each other and depend on no other script on the page.



# Async and Defer

- `async` and `defer` both instruct the browser to download the script(s) in a separate thread, while the rest of the page (the DOM, etc.) is downloading, so the page loading is not blocked during the fetch process.
- Scripts with an `async` attribute will execute as soon as the download is complete. This blocks the page and does not guarantee any specific execution order.
- Scripts with a `defer` attribute will load in the order they are in and will only execute once everything has finished loading.
- If your scripts should be run immediately and they don't have any dependencies, then use `async`.
- If your scripts need to wait for parsing and depend on other scripts and/or the DOM being in place, load them using `defer` and put their corresponding `<script>` elements in the order you want the browser to execute them.

# Comments in Javascript

- A single line comment is written after a double forward slash (//), e.g.

```
// I am a comment
```

- A multi-line comment is written between the strings /\* and \*/, e.g.

```
/* I am  
also a  
comment */
```

# Javascript Operators

Operator	Name	Purpose	Example
<code>+</code>	Addition	Adds two numbers together.	<code>6 + 9</code>
<code>-</code>	Subtraction	Subtracts the right number from the left.	<code>20 - 15</code>
<code>*</code>	Multiplication	Multiplies two numbers together.	<code>3 * 7</code>
<code>/</code>	Division	Divides the left number by the right.	<code>10 / 5</code>
<code>%</code>	Remainder (sometimes called modulo)	Returns the remainder left over after you've divided the left number into a number of integer portions equal to the right number.	<code>8 % 3</code> (returns 2, as three goes into 8 twice, leaving 2 left over).
<code>**</code>	Exponent	Raises a <code>base</code> number to the <code>exponent</code> power, that is, the <code>base</code> number multiplied by itself, <code>exponent</code> times.	<code>5 ** 2</code> (returns <code>25</code> , which is the same as <code>5 * 5</code> ).

# Javascript Operators


Operator	Name	Purpose	Example	Shortcut for
<code>+=</code>	Addition assignment	Adds the value on the right to the variable value on the left, then returns the new variable value	<code>x += 4;</code>	<code>x = x + 4;</code>
<code>-=</code>	Subtraction assignment	Subtracts the value on the right from the variable value on the left, and returns the new variable value	<code>x -= 3;</code>	<code>x = x - 3;</code>
<code>*=</code>	Multiplication assignment	Multiplies the variable value on the left by the value on the right, and returns the new variable value	<code>x *= 3;</code>	<code>x = x * 3;</code>
<code>/=</code>	Division assignment	Divides the variable value on the left by the value on the right, and returns the new variable value	<code>x /= 5;</code>	<code>x = x / 5;</code>



# Javascript Operators

Operator	Name	Purpose	Example
<code>===</code>	Strict equality	Tests whether the left and right values are identical to one another	<code>5 === 2 + 4</code>
<code>!==</code>	Strict-non-equality	Tests whether the left and right values are <b>not</b> identical to one another	<code>5 !== 2 + 3</code>
<code>&lt;</code>	Less than	Tests whether the left value is smaller than the right one.	<code>10 &lt; 6</code>
<code>&gt;</code>	Greater than	Tests whether the left value is greater than the right one.	<code>10 &gt; 20</code>
<code>&lt;=</code>	Less than or equal to	Tests whether the left value is smaller than or equal to the right one.	<code>3 &lt;= 2</code>
<code>&gt;=</code>	Greater than or equal to	Tests whether the left value is greater than or equal to the right one.	<code>5 &gt;= 4</code>

# Javascript Operators

Operator	Name	Example
<code>===</code>	Strict equality (is it exactly the same?)	<div>JS </div> <pre>5 === 2 + 4 // false 'Chris' === 'Bob' // false 5 === 2 + 3 // true 2 === '2' // false; number versus string</pre>
<code>!==</code>	Non-equality (is it not the same?)	<div>JS </div> <pre>5 !== 2 + 4 // true 'Chris' !== 'Bob' // true 5 !== 2 + 3 // false 2 !== '2' // true; number versus string</pre>
<code>&lt;</code>	Less than	<div>JS </div> <pre>6 &lt; 10 // true 20 &lt; 10 // false</pre>
<code>&gt;</code>	Greater than	<div>JS </div> <pre>6 &gt; 10 // false 20 &gt; 10 // true</pre>

# Loops in Javascript

```
const fruits = ["apples", "bananas", "cherries"];  
for (const fruit of fruits) {  
    console.log(fruit);  
}
```

Can also use a while loop (look it up)

# Variable Types

- Numbers
- Strings
- Booleans
- Arrays
- Objects

JavaScript is a "dynamically typed language", which means that, unlike some other languages, you don't need to specify what data type a variable will contain (numbers, strings, arrays, etc.). Just use the keywords `var` or `let` to declare / initialize the values.

Constants: like variables, except that:

- you must initialize them when you declare them
- you can't assign them a new value after you've initialized them

# Strings

- Creation
- Escaping characters
- Concatenation
- Multiline strings
- String methods

# Objects

An object is a collection of related data and/or functionality

```
const person = {  
  name: ["Bob", "Smith"],  
  age: 32,  
  bio: function () {  
    console.log(`${this.name[0]} ${this.name[1]} is  
    ${this.age} years old.`); },  
  introduceSelf: function () {  
    console.log(`Hi! I'm ${this.name[0]}.`);  
  },  
};
```

# Objects

`person.name;`

`person.name[0];`

`person.age;`

`person.bio(); // "Bob Smith is 32 years old."`

`person.introduceSelf(); // "Hi! I'm Bob."`

Dot and Bracket notation – for accessing object properties

```
const person = { name: { first: "Bob", last: "Smith", }, age: 32, // ... };
```

`person.name.first; person.name.last;`

`person["age"]; person["name"]["first"];`

Constructors

# JSON Objects

- JSON is purely a string with a specified data format — it contains only properties, no methods.
- JSON requires double quotes to be used around strings and property names. Single quotes are not valid other than surrounding the entire JSON string.
- Even a single misplaced comma or colon can cause a JSON file to go wrong, and not work. You should be careful to validate any data you are attempting to use (although computer-generated JSON is less likely to include errors, as long as the generator program is working correctly). You can validate JSON using an application like [JSONLint](#).
- JSON can actually take the form of any data type that is valid for inclusion inside JSON, not just arrays or objects. So for example, a single string or number would be valid JSON.
- Unlike in JavaScript code in which object properties may be unquoted, in JSON only quoted strings may be used as properties.



# An Example With Javascript + HTML

```
<!DOCTYPE html>
<html lang="en-us">
<head>
  <meta charset="utf-8">

  <title>Number guessing game</title>
  <script src="javascript_code.js">
  <style>
    html {
      font-family: sans-serif;
    }

    body {
      width: 50%;
      max-width: 800px;
      min-width: 480px;
      margin: 0 auto;
    }

    .form input[type="number"] {
      width: 200px;
    }

    .lastResult {
      color: white;
      padding: 3px;
    }
  </style>
</head>
```

```
<body>
  <h1>Number guessing game</h1>

  <p>We have selected a random number between 1 and 100.
  See if you can guess it in 10 turns or fewer. We'll tell you if your
  guess was too high or too low.</p>

  <div class="form">
    <label for="guessField">Enter a guess: </label>
    <input type="number" min="1" max="100" required
    id="guessField" class="guessField">
    <input type="submit" value="Submit" class="guessSubmit"
    </div>

    <div class="resultParas">
      <p class="guesses"></p>
      <p class="lastResult"></p>
      <p class="lowOrHi"></p>
    </div>
  </body>
</html>
```

# An Example With Javascript + HTML

```
let randomNumber = Math.floor(Math.random() * 100) + 1;
const guesses = document.querySelector('.guesses');
const lastResult = document.querySelector('.lastResult');
const lowOrHi = document.querySelector('.lowOrHi');
const guessSubmit = document.querySelector('.guessSubmit');
const guessField = document.querySelector('.guessField');
let guessCount = 1;
let resetButton;
```

```
function checkGuess() {
    const userGuess = Number(guessField.value);
    if (guessCount === 1) {
        guesses.textContent = 'Previous guesses: ';
    }

    guesses.textContent += userGuess + ' ';

    if (userGuess === randomNumber) {
        lastResult.textContent = 'Congratulations! You got it right!';
        lastResult.style.backgroundColor = 'green';
        lowOrHi.textContent = '';
        setGameOver();
    } else if (guessCount === 10) {
        lastResult.textContent = '!!!GAME OVER!!!';
        lowOrHi.textContent = '';
        setGameOver();
    } else {
        lastResult.textContent = 'Wrong!';
        lastResult.style.backgroundColor = 'red';
        if (userGuess < randomNumber) {
            lowOrHi.textContent = 'Last guess was too low!';
        } else if (userGuess > randomNumber) {
            lowOrHi.textContent = 'Last guess was too high!';
        }
    }

    guessCount++;
    guessField.value = '';
    guessField.focus();
}
```

# An Example With Javascript + HTML

```
guessSubmit.addEventListener('click', checkGuess);
```

```
function setGameOver() {  
  guessField.disabled = true;  
  guessSubmit.disabled = true;  
  resetButton = document.createElement('button');  
  resetButton.textContent = 'Start new game';  
  document.body.appendChild(resetButton);  
  resetButton.addEventListener('click', resetGame);  
}
```

```
function resetGame() {  
  guessCount = 1;  
  const resetParas = document.querySelectorAll('.resultParas p');  
  for (const resetPara of resetParas) {  
    resetPara.textContent = "";  
  }  
}
```

```
resetButton.parentNode.removeChild(resetButton);  
guessField.disabled = false;  
guessSubmit.disabled = false;  
guessField.value = "";  
guessField.focus();  
lastResult.style.backgroundColor = 'white';  
randomNumber = Math.floor(Math.random() * 100) + 1;  
}
```

# **PostgreSQL and PostGIS**

# What is PostgreSQL?

An open source relational database management system (DBMS) developed by a worldwide team of volunteers. PostgreSQL is not controlled by any corporation or other private entity and the source code is available free of charge.

# History of PostgreSQL

- A Brief History of PostgreSQL
- PostgreSQL, originally called Postgres, was created at UCB by a computer science professor named Michael Stonebraker. Stonebraker started Postgres in 1986 as a follow-up project to its predecessor, Ingres, now owned by Computer Associates.
- **1977-1985** – A project called INGRES was developed.
  - Proof-of-concept for relational databases
  - Established the company Ingres in 1980
  - Bought by Computer Associates in 1994
- **1986-1994** – POSTGRES
  - Development of the concepts in INGRES with a focus on object orientation and the query language - Quel
  - The code base of INGRES was not used as a basis for POSTGRES
  - Commercialized as Illustra (bought by Informix, bought by IBM)
- **1994-1995** – Postgres95
  - Support for SQL was added in 1994
  - Released as Postgres95 in 1995
  - Re-released as PostgreSQL 6.0 in 1996
  - Establishment of the PostgreSQL Global Development Team

# Database Operations

- Creation

CREATE DATABASE db\_name

OWNER = role\_name

TEMPLATE = template

ENCODING = encoding

LC\_COLLATE = collate

LC\_CTYPE = ctype

TABLESPACE = tablespace\_name

CONNECTION LIMIT = max\_concurrent\_connection

- **db\_name:** It is the name of the new database that you want to create. It must always be a unique name.
- **role\_name:** It is the role name of the user who will own the new database.
- **template:** It is the name of the database template from which the new database gets created.
- **encoding:** It specifies the character set encoding for the new database. By default, it is the encoding of the template database.
- **collate:** It specifies a collation for the new database.
- **ctype:** It specifies the character classification for the new database like digit, lower and upper.
- **tablespace\_name:** It specifies the tablespace name for the new database.
- **max\_concurrent\_connection:** It specifies the maximum concurrent connections to the new database.

# Datatypes

- **Boolean**
- **Character** Types [ *such as char, varchar, and text*]
- **Numeric** Types [ *such as integer and floating-point number*]
- **Temporal** Types [ *such as date, time, timestamp, and interval*]
- **UUID** [ *for storing UUID (Universally Unique Identifiers)* ]
- **Array** [ *for storing array strings, numbers, etc.*]
- **JSON** [ *stores JSON data*]
- **hstore** [ *stores key-value pair*]
- **Special** Types [ *such as network address and geometric data*]



# Querying Data

The various clauses that can be used with the SELECT statement are listed below:

- **DISTINCT operator:** It is used to select distinct rows from a table.
- **ORDER BY clause:** It is used to sort table rows.
- **WHERE clause:** It is used to filter rows from a table. **LIMIT clause:** It is used to select a subset of rows from the table
- **FETCH clause:** It is also used to select subset of rows from the table.
- **GROUP BY clause:** It is used to group different rows into a single group.
- **HAVING clause:** It is used to filter rows from a table with specified attributes or features.
- **FROM clause:** It is used to specify a column in a table.
- **joins:** It is used to join two or more tables together using joins such as **INNER JOIN, LEFT JOIN, FULL OUTER JOIN, CROSS JOIN** clauses.
- **Set operators:** These operators such as **UNION, INTERSECT, and EXCEPT** are used to manipulate the different sets of data.

# Modifying Data

- INSERT INTO table(column1, column2, ...) VALUES (value1, value2, ...);
- UPDATE table SET column1 = value1, column2 = value2, ... WHERE condition;
- DELETE FROM table WHERE condition;
- INSERT INTO table\_name(column\_list) VALUES(value\_list) ON CONFLICT target action;

–The target can be :

- (column\_name) – any column name.
- ON CONSTRAINT constraint\_name – where the constraint name could be the name of the UNIQUE constraint.
- WHERE predicate – a WHERE clause with a boolean condition.

–The action can be :

- DO NOTHING – If the row already exists in the table, then do nothing.
- DO UPDATE SET columnA = valueA, ... WHERE condition – update some fields in the table depending upon the condition.

# Keys and Constraints

- Primary Key:** A primary key is a column or a group of columns used to identify a row uniquely in a table. Technically speaking a primary key constraint is the blend of a not-null constraint and a UNIQUE constraint. Only one primary key must exist in a table.
- Foreign Key:** A foreign key is a column or a group of columns used to identify a row uniquely of a different table. The table that comprises the foreign key is called the referencing table or child table. And the table to that the foreign key references is known as the referenced table or parent table. A table can possess multiple foreign keys according to its relationships with other tables.
- Check Constraint:** The CHECK constraint is primarily used to specify if a value in a column necessarily meets a specific requirement. The CHECK constraint utilizes a Boolean expression to assess the values before performing an insert or update operation to the column. If the values pass the check, PostgreSQL allows the insertion or update of those values into the column. It is primarily used while creating a table. Syntax is: `variable_name Data-type CHECK(condition);`
- Unique Constraint:** Used to make sure that values stored in a column or a group of columns are unique across rows in a table. Every time the user inserts a new row, PostgreSQL checks if the value already exists in the table if UNIQUE constraints are used. If it discovers that the new value is already present, it denies the change and issues an error. A similar process is carried out for updating existing data.
- Not NULL Constraint:** Not-Null constraint as the name suggests is used to ensure that any value in the respective column is not null. In the world of database, NULL is unknown or missing information. The NULL value is separate from an empty string or the number zero.

# Joins

There are 4 basic types of joins supported by PostgreSQL, namely:

- Inner Join
- Left Join
- Right Join
- Full Outer Join

Some special PostgreSQL joins are below:

- Natural Join
- Cross Join
- Self Join

# Operators

- **UNION:** Used to combine result sets of multiple queries into a single set of result. It is used to combine result sets of two or more [SELECT](#) statements into a single result set. **Syntax:** SELECT column\_1, column\_2 FROM table\_name\_1 UNION SELECT column\_1, column\_2 FROM table\_name\_2;
- **INTERSECT:** used to combine two or more result sets returned by the [SELECT](#) statement and provide with the common data among the tables into a single result set. **Syntax:** SELECT column\_list FROM A INTERSECT SELECT column\_list FROM B; The below rules must be followed while using the INTERSECT operator with the SELECT statement:
  - The number of columns and their order in the SELECT clauses must be the same.
  - The data types of the columns must be compatible.
- **EXCEPT:** used to return distinct rows from the first (left) query that are not in the output of the second (right) query while comparing result sets of two or more queries. **Syntax:** SELECT column\_list FROM A WHERE condition\_a EXCEPT SELECT column\_list FROM B WHERE condition\_b;

# Operators

- **ANY:** used to compare a scalar value with a set of values returned by a subquery. **Syntax:** expression operator ANY(subquery)The below rules must be followed while using PostgreSQL ANY operator:
  - The subquery must return exactly one column.
  - The ANY operator must be preceded by one of the following comparison operator =, <=, >, <, > and <>
  - The ANY operator returns true if any value of the subquery meets the condition, otherwise, it returns false.
- **ALL:** used for comparing a value with a list of values returned by a subquery. **Syntax:** comparison\_operator ALL (subquery)The below rules need to be followed while using the ALL operator:
  - The ALL operator always needs to be preceded by a comparison operator(=, !=, <, >, >=, <=).
  - It must always be followed by a subquery surrounded by parentheses.
- **EXISTS:** used to test for the existence of rows in a subquery.It is generally used with correlated subqueries. If the subquery returns at least one row, the result of EXISTS is true. In case the subquery returns no row, the result is of EXISTS is false. **Syntax:** EXISTS (subquery)