

FULL STACK DEVELOPMENT
[R22A0513]
LECTURE NOTES
B. TECH III YEAR – I SEMESTER (R22)
(2025-26)



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)

Recognized under 2(f) and 12 (B) of UGC ACT 1956

(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC –
‘A’ Grade - ISO 9001:2015 Certified)

Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100,
Telangana State, India

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDEX

SNO	UNIT	TOPIC	PAGE NO
1	I	Understanding the Basic Web Development Framework-	5
2	I	HTML Basics	9
3	I	Cascading Style Sheets	38
4	I	Version Control	47
5	II	JavaScript basics, Functions, form validation	62
6	II	OOPS Aspects of JavaScript	93
7	II	JQuery Framework, jQuery events	106
8	II	AJAX for data exchange with server, JSON data format	114
9	III	Angular: importance of Angular, Understanding Angular	120
10	III	creating a Basic Angular Application, Angular Components, Expressions	124
11	III	Data Binding, Built-in Directives, Custom Directives	127
12	III	Implementing AngularServices in Web Applications	145
13	III	Need of React, Simple React Structure, The Virtual DOM, React Components	162
14	III	Introducing React Components, Creating Components in React, Data and Data Flow in React	175
15	III	Rendering and Life Cycle Methods in React, Working with forms in React	192
16	III	integrating third party libraries, Routing in React	207
17	IV	Node js: Getting Started with Node.js, Using Events, Listeners, Timers	214
18	IV	Callbacks in Node.js, Handling Data I/O in Node.js	225
19	IV	Accessing the File System from Node.js	235
20	V	Understanding NoSQL and MongoDB, Getting Started with MongoDB	248
21	V	Getting Started with MongoDB and Node.js	250
22	V	Accessing MongoDB from Node.js	318
23	V	Using Mongoose for Structured Schema and Validation	323
24	V	Advanced MongoDB Concepts	325

(R22A0513) Full Stack Development

COURSE OBJECTIVES:

1. To become knowledgeable about the most recent web development technologies.
2. Idea for creating two tier and three tier architectural web applications.
3. Students will become familiar to implement fast, efficient, interactive and scalable web applications using run time environment provided by the full stack components
4. Design and Analyze real time web applications and Constructing suitable client and server side applications.
5. To learn core concept of both front end and back end programming.

UNIT - I

Web Development Basics: Understanding the Basic Web Development Framework- User, Browser, Webserver, Backend Services, **HTML Basics:** Headings, Paragraphs, Links, Images, Lists, Tables, Div Element, Forms, **Cascading Style Sheets:** Syntax, Types, Selectors, Background, Border, Font, Text, Table, box model, **Version Control:** Getting Started with Git, Git Basics, Git Branching and Merging, working with remote repositories.

UNIT - II

JavaScript and jQuery: JavaScript basics, Functions, form validation, OOPS Aspects of JavaScript, JQuery Framework, jQuery events, AJAX for data exchange with server, JSON data format.

UNIT - III

Angular: importance of Angular, Understanding Angular, creating a Basic Angular Application, Angular Components, Expressions, Data Binding, Built-in Directives, Custom Directives, Implementing AngularServices in Web Applications.

React:

Need of React, Simple React Structure, The Virtual DOM, React Components, Introducing React Components, Creating Components in React, Data and Data Flow in React, Rendering and Life Cycle Methods in React, Working with forms in React, integrating third party libraries, Routing in React.

UNIT – IV

Node js: Getting Started with Node.js, Using Events, Listeners, Timers, and Callbacks in Node.js, Handling Data I/O in Node.js, Accessing the File System from Node.js, Implementing Socket Services in Node.js.

UNIT – V

MongoDB:

Understanding NoSQL and MongoDB, Getting Started with MongoDB, Getting Started with MongoDB and Node.js, Manipulating MongoDB Documents from Node.js, Accessing MongoDB from Node.js, Using Mongoose for Structured Schema and Validation, Advanced MongoDB Concepts.

TEXT BOOKS:

1. Web Design with HTML, CSS, JavaScript and JQuery Set Book by Jon Duckett
Professional JavaScript for Web Developers Book by Nicholas C. Zakas. **(Unit-I, II).**
2. ProGit, 2nd Edition, Apress publication by Scott Chacon and Straub. **(Unit I).**
3. Mark Tielens Thomas, React in Action, 1st Edition, Manning Publications. **(Unit-III).**

4. Brad Dayley, Brendan Dayley, Caleb Dayley., Node.js, MongoDB and Angular Web Development, 2nd Edition, Addison-Wesley, 2019. **(Unit-III, Unit-IV, Unit-V).**

REFERENCE BOOKS:

1. Vasan Subramanian, Pro MERN Stack, Full Stack Web App Development with Mongo, Express, React, and Node, 2nd Edition, Apress, 2019.
2. Chris Northwood, The Full Stack Developer: Your Essential Guide to the Everyday Skills Expected of a Modern Full Stack Web Developer', 1st edition, Apress, 2018.
3. Kirupa Chinnathambi, Learning React: A Hands-On Guide to Building Web Applications Using React and Redux, 2nd edition, Addison-Wesley Professional, 2018.

COURSEOUTCOMES:

1. Understand Full stack components for developing web application.
2. Students are able to develop a dynamic webpage by the use of java script and jQuery.
3. Design faster and effective single page applications using Angular and Create interactive user interfaces with react components
4. Apply packages of NodeJS to work with Data, Files, Http Requests and Responses.
5. Use MongoDB data base for storing and processing huge data and connects with NodeJS application.

UNIT – I

Web Development Basics: Understanding the Basic Web Development Framework- User, Browser, Webserver, Backend Services, **HTML Basics:** Headings, Paragraphs, Links, Images, Lists, Tables, Div Element, Forms, **Cascading Style Sheets:** Syntax, Types, Selectors, Background, Border, Font, Text, Table, box model, **Version Control:** Getting Started with Git, Git Basics, Git Branching and Merging, working with remote repositories.

Web Development Basics:

Understanding the Basic Web Development Framework:

- This topic focuses on the fundamental components of the web development framework and then describes the components of the Node.js-to-Angular stack and discusses various aspects of the general website/web application development framework from users to backend services.
- The main components of any given web framework are the user, browser, webserver, and backend services.
- In the following figure, components are described in a top-down manner from user down to backend services.

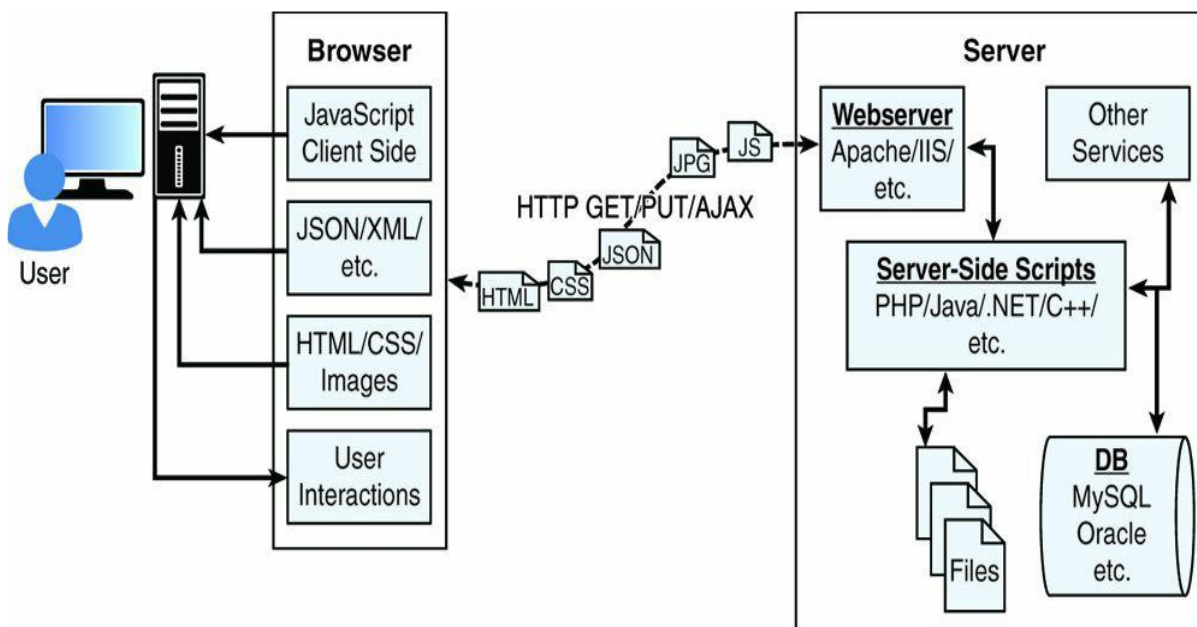


Figure 1.1 Diagram showing the components of a basic website/web application.

User:

Users are a fundamental part of all websites; User expectations define the requirements for developing a good website, and these expectations have changed a lot over the years.

- The **user role** in a web framework is to sit on the visual output and interaction input of webpages. That is, users view the results of the web framework processing and then provide interactions using mouse clicks, keyboard input, and swipes and taps on mobile devices.

Browser

The browser plays three roles in the web framework.

- First, it provides communication to and from the webserver.
- Second, it interprets the data from the server and renders it into the view that the user actually sees.
- Finally, the browser handles user interaction through the keyboard, mouse, touchscreen, or other input device and takes the appropriate action.

✓ **Browser to Webserver Communication**

- Browser-to-webserver communication consists of a series of requests using the HTTP and HTTPS protocols.
- Hypertext Transfer Protocol (HTTP) defines communication between the browser and the webserver.
- HTTP defines what types of requests can be made as well as the format of those requests and the HTTP response.
- HTTPS adds an additional security layer, SSL/TLS, to ensure secure connections by requiring the webserver to provide a certificate to the browser. The user then can determine whether to accept the certificate before allowing the connection.

The browser makes three main types of requests to the server:

GET: The GET request is typically used to retrieve data from the server, such as .html files, images, or JSON data.

POST: POST requests are used when sending data to the server, such as adding an item to a shopping cart or submitting a web form.

AJAX: Asynchronous JavaScript and XML (AJAX) is actually just a GET or POST request done directly by JavaScript running in the browser. Despite the name, an AJAX request can receive XML, JSON, or raw data in the response.

✓ **Rendering the Browser View**

The screen that the user actually views and interacts with is often made up of several different pieces of data retrieved from the webserver.

The browser reads data from the initial URL and then renders the HTML document to build a Document Object Model (DOM).

The DOM is a tree structure object with the HTML document as the root. The structure of the tree basically matches the structure of the HTML document.

For example, the document will have html as a child, and html will have head and body as children, and body may have div, p, or other elements as children, like this:

```
document
  + html
    + head
    + body
      + div
      + p.
```

The browser interprets each DOM element and renders it to the user's screen to build the webpage view. The browser often ends up getting various types of data from multiple webserver requests to build the webpage.

The final view of the web page/web page behaviour is contains following content.

- **HTML files:** These provide the fundamental structure of the DOM.
- **CSS files:** These define how each of the elements on the page is to be styled;
for example, font, colour, borders, and spacing.
- **Client-side scripts:** These are typically JavaScript files. They can provide added functionality to the webpage, manipulate the DOM to change the look of the webpage, and provide any necessary logic required to display the page and provide functionality.
- **Media files:** Image, video, and sound files are rendered as part of the webpage.
- **Data:** Any data, such as XML, JSON, or raw text, can be provided by the webserver as a response to an AJAX request. Rather than sending a request back to the server to rebuild the webpage, new data can be retrieved via AJAX and inserted into the webpage via JavaScript.
- **HTTP headers:** The HTTP protocol defines a set of headers that can be used by the browser and client-side scripts to define the behaviour of the webpage.
For example, cookies are contained in the HTTP headers. The HTTP headers also define the type of data in the request as well as the type of data expected to be returned back to the browser.

✓ User Interaction

- The user interacts with the browser via input devices such as mice, keyboards, and touchscreens.
- The browser has an elaborate event system that captures these users input events and then takes the appropriate action. Actions vary from displaying a popup menu to loading a new document from the server to executing client-side JavaScript.

Webserver

- The webserver's main focus is handling requests from browsers.
- the browser may request a document, post data, or perform an AJAX request to get a data.
- The webserver uses the HTTP headers as well as the URL to determine what action to take.
- Different responses things will be generated depending on the server, its configurations and technologies.
- **Example: Apache and IIS**, are made to serve static files such as .html, .css, and media files.
- To handle POST requests that modify server data and AJAX requests to interact with backend services, webserver need to be extended with server-side scripts.
- A *server-side program* is really anything that can be executed by the webserver to perform the task the browser is requesting. These can be written in PHP, Python, C, C++, C#, Java, ... the list goes on and on. Webserver such as Apache and IIS provide mechanisms to include server-side scripts and then wire them up to specific URL locations requested by the browser.
- The server-side scripts either generate the response directly by executing their code or connect with other **backend servers** such as databases to obtain the necessary information and then use that information to build and send the appropriate response.

✓ Backend Services:

- Backend services are services that run behind the webserver and provide data used to build responses to the browser.
- The most common type of backend service is a database that stores information.

- When a request comes in from the browser that requires information from the database or other backend service, the server-side script connects to the database, retrieves the information, formats it, and then sends it back to the browser.
- Conversely, when data comes in from a web request that needs to be stored in the database, the server-side script connects to the database and updates the data.

Understanding the Node.js-to Angular Stack Components:

- This web development stack is the Node.js-to-Angular stack comprised of **MongoDB, Express, Angular, and Node.js**.
- **Node.js** provides the fundamental platform for development. The backend services and server-side scripts are all written in Node.js.
- **MongoDB** provides the data store for the website but is accessed via a **MongoDB driver** Node.js module.
- The webserver is defined by **Express**, which is also a Node.js module.
- The view in the browser is defined and controlled using the **Angular framework**.
- **Angular** is an MVC framework where the model is made up of JSON or JavaScript objects, the view is HTML/CSS, and the controller is made up of Angular JavaScript.

Figure 1.2 provides a basic diagram of how the Node.js-to-Angular stack fits into the basic **website/web application model**.

The following sections describe each of these technologies and why they were chosen as part of the Node.js-to-Angular stack.

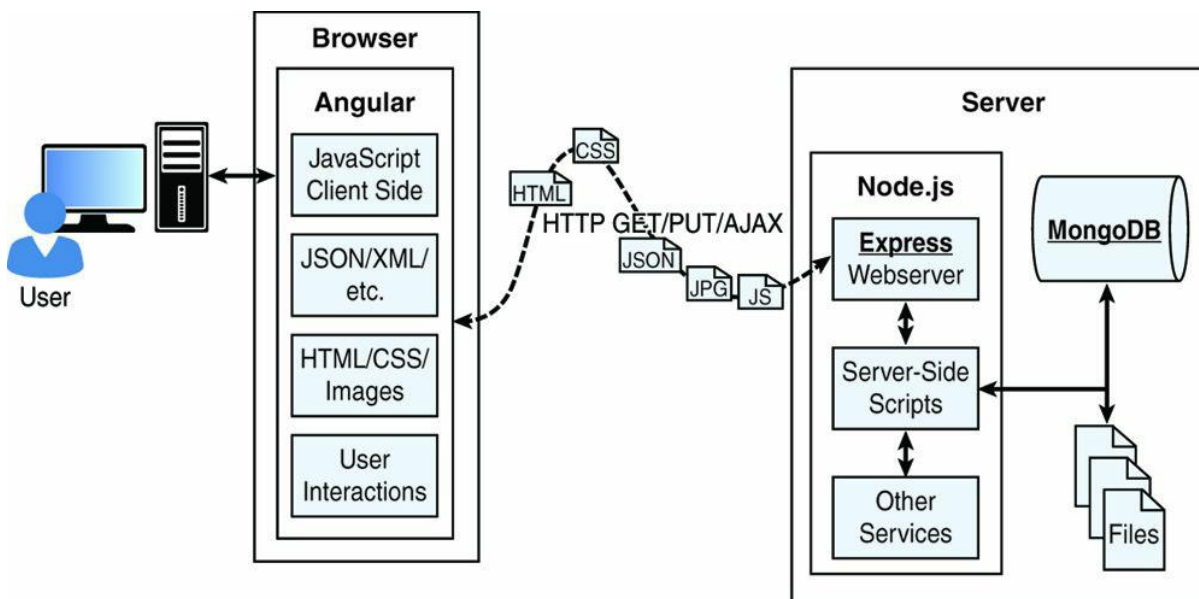


Figure 1.2 Basic diagram showing where Node.js, Express, MongoDB, and Angular fit in the web paradigm

Node.js:

- Node.js is a development framework based on **Google's V8 JavaScript engine**. Therefore, Node.js code is written in JavaScript and then compiled into machine code by V8 to be executed.
- Many of your **backend services** can be **written in Node.js**, as can the server-side scripts and any supporting web application functionality.

- Node.js is that it is all just JavaScript, so you can easily take functionality from a client-side script and place it in a server-side script. Also, the webserver can run directly within the Node.js platform as a Node.js module.

The following are just a few reasons why Node.js is a great framework to start from:

Features of Node.js:

JavaScript end-to-end: One of the biggest advantages to Node.js is that it allows you to write both server- and client-side scripts in JavaScript. There have always been difficulties in deciding where to put scripting logic. Too much on the client side makes the client cumbersome and unwieldy, but too much on the server side slows down web applications and puts a heavy burden on the webserver. With Node.js you can take JavaScript written on the client and easily adapt it for the server and vice versa. Also, your client developers and server developers will be speaking the same language.

- ✓ **Event-driven scalability:** Node.js applies a different logic to handling web requests. Rather **than having multiple threads waiting to process web requests, they are processed on the same thread** using a basic event model. This allows Node.js webserver to scale in ways that traditional webserver never can.
- ✓ **Extensibility:** Node.js has a great following and an active development community. New modules to extend Node.js functionality are being developed all the time. Also it is simple to install and include new modules in Node.js, making it easy to extend a Node.js project to include new functionality in minutes.
- ✓ **Time:** Let's face it, time is valuable. Node.js is super easy to set up and develop in. In only a few minutes, you can install Node.js and have a working webserver.

MongoDB:

- MongoDB is an agile and scalable NoSQL database. The name Mongo comes from “**humongous**.” It is based on the **NoSQL document store model**, meaning that data is stored in the database as a form of **JSON objects** rather than the traditional columns and rows of a relational database.
- MongoDB provides great website backend storage for high traffic websites that need to store data such as user comments, blogs, or other items because it is fast, scalable, and easy to implement.
- Node.js supports a variety of DB access drivers, so the data store could just as easily be MySQL or some other database

The following are some of the reasons that MongoDB really fits in the Node.js stack well:

- ✓ **Document orientation:** Because MongoDB is document-oriented, the data is stored in the database in a format close to what you will be dealing with in both server-side and client-side scripts. This eliminates the need to transfer data from rows to objects and back.
- ✓ **High performance:** MongoDB is one of the highest performing databases available. Especially today when more and more people interact with websites, it is important to have a backend that can support heavy traffic.
- ✓ **High availability:** MongoDB's replication model makes it easy to maintain scalability while keeping high performance.
- ✓ **High scalability:** MongoDB's structure makes it easy to scale horizontally by sharing the data across multiple servers.

- ✓ **No SQL injection:** MongoDB is not susceptible to SQL injection (putting SQL statements in web forms or other input from the browser that compromises the DB security) because objects are stored as objects, not using SQL strings.

Express:

- The Express module acts as the webserver in the Node.js-to-Angular stack. The fact that it is running in Node.js makes it easy to configure, implement, and control.
- The Express module extends Node.js to provide several key components for handling web requests. This allows you to implement a running webserver in Node.js with only a few lines of code.
- **For example**, the Express module provides the ability to easily set up destination route (URLs) for users to connect to. It also provides great functionality on working with the HTTP request and response objects, including things like cookies and HTTP headers.

The following is a partial list of the valuable features of Express:

- ✓ **Route management:** Express makes it easy to define routes (URL endpoints) that tie directly to Node.js script functionality on the server.
- ✓ **Error handling:** Express provides built-in error handling for documents not found and other errors.
- ✓ **Easy integration:** An Express server can easily be implemented behind an existing reverse proxy system such as Nginx or Varnish. This allows it to be easily integrated into your existing secured system.
- ✓ **Cookies:** Express provides easy cookie management.
- ✓ **Session and cache management:** Express also enables session management and cache management.

Angular:

- Angular is a client-side framework developed by Google.
- Angular provides all the functionality needed to handle user input in the browser, manipulate data on the client side, and control how elements are displayed in the browser view.
- It is written using TypeScript. The entire theory behind Angular is to provide a framework that makes it easy to implement web applications using the MVC framework.
- ✓ Other JavaScript frameworks could be used with the Node.js platform, such as Backbone, Ember, and Meteor. However, Angular has the best design, feature set, and trajectory at this writing.

Here are some of the benefits of Angular: **(features of Angular)**

- ✓ **Data binding:** Angular has a clean method to bind data to HTML elements using its powerful scope mechanism.
- ✓ **Extensibility:** The Angular architecture allows you to easily extend almost every aspect of the language to provide your own custom implementations.
- ✓ **Clean:** Angular forces you to write clean, logical code.
- ✓ **Reusable code:** The combination of extensibility and clean code makes it easy to write reusable code in Angular. In fact, the language often forces you to do so when creating custom services.
- ✓ **Support:** Google is investing a lot into this project, which gives it an advantage over other similar initiatives.

- ✓ **Compatibility:** Angular is based on **TypeScript**, which makes it easier to begin integrating Angular into your environment and to reuse pieces of your existing code within the structure of the Angular framework.

HTML Basics:

Headings, Paragraphs, Links, Images, Lists, Tables, Div Element, Forms.

HTML:

- HTML stands for Hypertext Markup Language. which is used for creating web pages and web applications.
- *Understanding meaning of Hypertext, Markup Language, and Web page.*
- **Hypertext** refers to the way in which Web pages (HTML documents) are linked together. A text has a link within it, is a hypertext. Whenever you click on a link, it will navigates you to another web page(document). Thus, the link available on a webpage is called **Hypertext**.
- HTML is a **Markup Language** which means you use HTML to simply "mark-up" a text document with tags that tell a Web browser how to structure it to display.
- A **web page** is a simple document displayable by a [browser](#). Such documents are written in the [HTML](#) language). A web page can embed a variety of different types of resources such as:

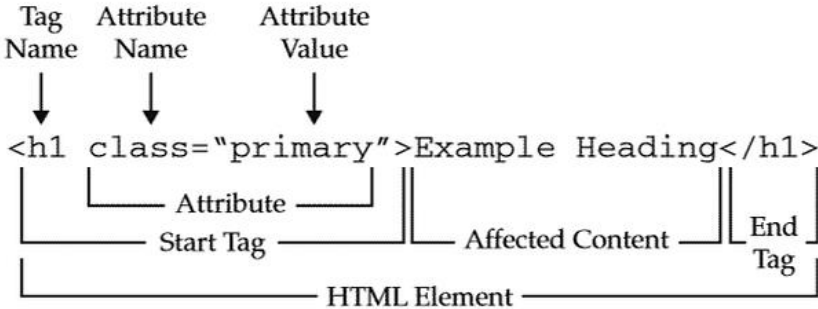
style information — controlling a page's look-and-feel

scripts — which add interactivity to the page

media — images, sounds, and videos.

- A web page can be identified by entering an URL.
- A Web page can be of the static or dynamic type. With the help of HTML only, we can create static web pages.
- A **website** is a collection of linked web pages (plus their associated resources) that share a unique domain name. Each web page of a given website provides explicit links—most of the time in the form of clickable portions of text—that allow the user to move from one page of the website to another.
- To access a website, type its domain name in your browser address bar, and the browser will display the website's main web page, or homepage (casually referred as "the home").
- An **HTML document** is simply a text file that contains the information you want to publish and the appropriate markup instructions indicating how the browser should structure or present the document.
- Markup elements are made up of a start tag, such as ****, and typically, though not always, an end tag, which is indicated by a slash within the tag, such as ****. The tag pair should fully enclose any content to be affected by the element, including text and other HTML markup.
- HTML FILE = INFORMATION + MARKUP instructions(tags).
- **Syntax of HTML element:**
- **HTML Element**
- An HTML Tag with attributes and content is called HTML Element. Element include start tag, end tag, attributes and content inside.
-

1



- **empty elements:** An element without content and has no ending tag is called “empty element”.
- **For example**, to insert a line break, use a single `
` tag, which represents the empty br element, because it doesn’t enclose any content and thus has no corresponding close tag:

`
`
- Empty element is added with “/” after the element name.

`
`
- The start tag of an element might contain **attributes** that modify the meaning of the tag.
- **For example**, in HTML, the simple inclusion of the **noshade** attribute in an `<hr>` tag, as shown here:
`<hr noshade>`

indicates that there should be no shading applied to this horizontal rule.

- Under XHTML, such style attributes are not allowed, because all attributes must have a value, so instead you have to use syntax like this:

Values are always enclosed in either single or double quotes.

- **Multiple attributes:**

```

```

- **Example of html document:** **structure of html document.**

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"

"http://www.w3.org/TR/html4/strict.dtd">

<html>

<head>

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

<title>Hello HTML 4 World**</title>**

```
<!-- Simple hello world in HTML 4.01 strict example -->
```

</head>

<body>

Welcome to the World of HTML

</body>

-

</html>

- **<!DOCTYPE>** : which indicates the particular version of HTML or XHTML being used in the document.
- **<html>** : This tag informs the browser that it is an HTML document. Text between html tag describes the web document. It is a container for all other elements of HTML except <!DOCTYPE>.
- **<head>**: It should be the first element inside the <html> element, which contains the metadata (information about the document). the information contained within the head element is information about the page that is useful for visual styling, defining interactivity, setting the page title, and providing other useful information that describes or controls the document.
- **<title>**: As its name suggested, it is used to add title of that HTML page which appears at the top of the browser window. It must be placed inside the head tag and should close immediately.
- The **<meta>** : Specifying Content Type, Character Set, and More.

A **<meta>** tag has a number of uses.

For example, it can be used to specify values that are equivalent to HTTP response headers.

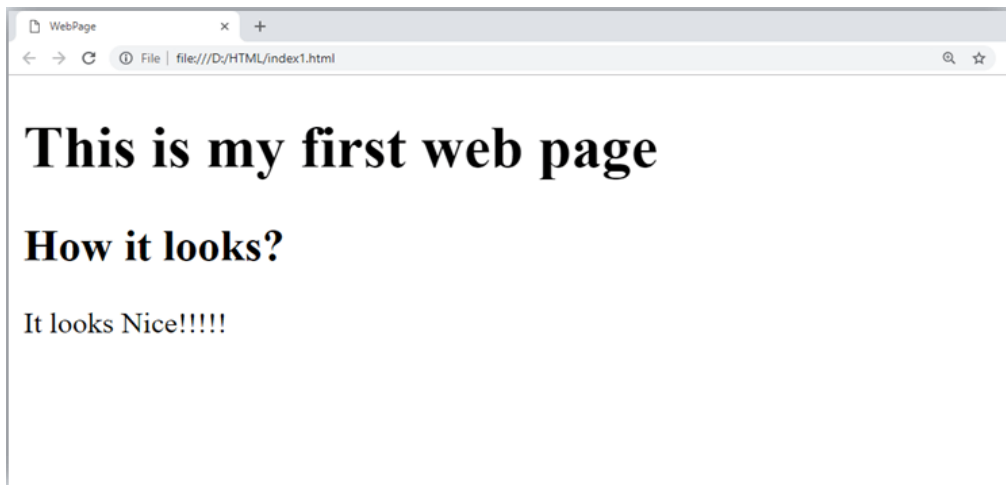
For example, if you want to make sure that your MIME type and character set for an English-based HTML document is set, you could use Most people would agree that using the UTF-8 character set is probably a good idea for Western-language page authors because it gives them access to international character glyphs when needed without causing them any trouble:

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"

- **<body>** : Text between body tag describes the body content of the page that is visible to the end user. This tag contains the main content of the HTML document.

Example:

```
<!DOCTYPE html>
<html>
<head>
<title> Webpage </title>
</head>
<body>
<h1>This is my first web page </h1>
<h2> How it looks? </h2>
<p>It looks Nice!!!! </p>
</body>
</html>
```



HTML Heading:

- A HTML heading or HTML h tag can be defined as a title or a subtitle which you want to display on the webpage. When you place the text within the heading tags `<h1>.....</h1>`, it is displayed on the browser in the bold format and size of the text depends on the number of heading.
- There are six different HTML headings which are defined with the `<h1>` to `<h6>` tags, from highest level h1 (main heading) to the least level h6 (least important heading).
- h1 is the largest heading tag and h6 is the smallest one. So h1 is used for most important heading and h6 is used for least important.
- **Example:**

```
<!DOCTYPE html>
<html>
<head>
<title> Webpage </title>
</head>
<h1> Welcome to Computer Application </h1>
<h2> Welcome to Computer Application </h2>
<h3> Welcome to Computer Application </h3>
<h4> Welcome to Computer Application </h4>
<h5> Welcome to Computer Application </h5>
<h6> Welcome to Computer Application </h6>
</body>
</html>
```

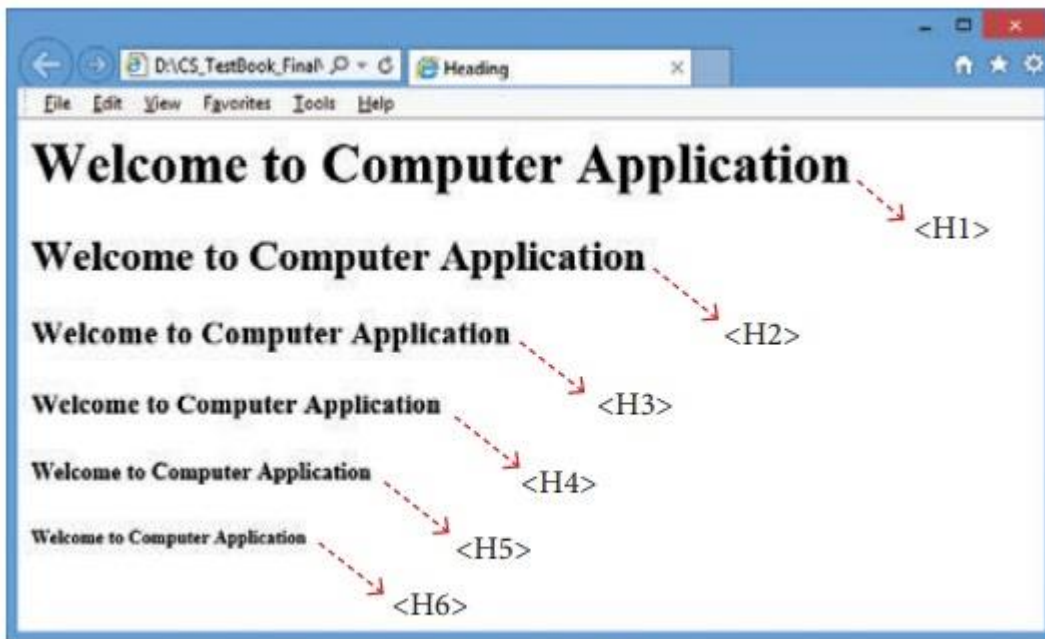


Figure10.12 – Different levels of Headings

Paragraphs:

- paragraphs tags or <p> tags in HTML help us create paragraphs on a web page. On web browsers, paragraphs display as blocks of text separated from adjacent blocks by blank lines, white spaces, or first-line indentation.
- You can use a <p> tag followed by the content you want to display in your paragraph and a </p>. Whenever the web browser comes across a <p> tag, it starts its contents on a new line.
- **HTML Paragraph Syntax:**

<p>Paragraph Content</p>

- **Explanation:**
 - <p>: Start tag for the paragraph.
 - Paragraph Content: The text will appear as a paragraph on a visitor's screen.
 - </p>: It is the closing tag for the paragraph.
- HTML paragraphs help us in multiple ways, such as:
 - They make a web page more readable by giving it a structural view.
 - Paragraphs can consist of different types of related content, such as text, images, forms, and more.
- **Here is a simple example in HTML to display different paragraphs on a web page:**

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

```

<title>Paragraph in HTML</title>

</head>

<body>

<p>
This is paragraph 1. This is paragraph 1. This is paragraph 1. This is paragraph 1. This is
paragraph 1. This is paragraph 1. This is paragraph 1. This is paragraph 1. This is
paragraph 1. This is paragraph 1.

</p>

<p>
This is paragraph 2.This is paragraph 2. This is paragraph 2.This is paragraph 2.This is
paragraph 2. This is paragraph 2.This is paragraph 2. This is paragraph 2. This is
paragraph 2.

</p>

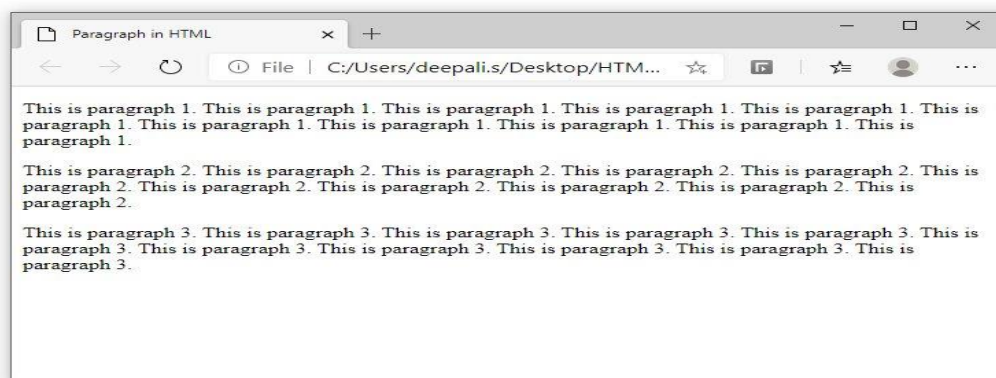
<p>
This is paragraph 3. This is paragraph 3. This is paragraph 3. This is paragraph 3. This is
paragraph 3. This is paragraph 3. This is paragraph 3. This is paragraph 3. This is
paragraph 3. This is paragraph 3. This is paragraph 3.

</p>

</body>

</html>

```



HTML Paragraph Tag Attributes:

Attribute	Value	Description
align	left right center justify	Aligns the text within a paragraph

Links:

- The <a> [HTML](#) element (or *anchor* element), with its [href](#) attribute, creates a hyperlink to web pages, files, email addresses, locations in the same page, or anything else a URL can address.(The <a> tag defines a hyperlink, which is used to link from one page to another).
- The most important attribute of the <a> element is the href attribute, which indicates the link's destination.
- By default, links will appear as follows in all browsers:
 - An unvisited link is underlined and blue
 - A visited link is underlined and purple
 - An active link is underlined and red

Example: External Link.

Standard Syntax:

```

<a
  Accesskey = "key"
  Charset = "character code for language of linked resource"
  Class = "class name(s)"
  coords = "comma-separated list of numbers"
  dir = "ltr | rtl"
  href = "URL"
  hreflang = "language code"
  id = "unique alphanumeric identifier"
  lang = "language code"
  name = "name of target location"
  rel = "comma-separated list of relationship values"
  rev = "comma-separated list of relationship values"
  shape="default | circle | poly | rect"
  style="style information"
  tabindex = "number"
  target = "frame or window name | _blank | _parent | _self | _top"
  title = "advisory text"
  type="content type of linked data" >
</a>

```

HTML 4 Event Attributes:

onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup,

onmousedown, onmousemove, onmouseout, onmouseover, onmouseup.

ATTRIBUTES:

charset: This attribute defines the character encoding of the linked resource.

href : This is the single required attribute for anchors defining a hypertext source link. It indicates the link target—either a URL or a URL fragment, which is a name preceded by a hash mark (#) specifying an internal target location within the current document. URLs are not restricted to Web-based (http) documents. URLs might use any protocol supported by the browser.

For example, file, ftp, and mailto work in most user agents.

hreflang : This attribute is used to indicate the language of the linked resource and should be set to whichever language is specified in the core lang attribute.

Download: Specifies that the target will be downloaded when a user clicks on the hyperlink.

Target: Specifies where to open the linked document.

_blank, which indicates a new window.

_parent, which indicates the parent frame set containing the source link;

_self, which indicates the frame containing the source link.

_top, which indicates the full browser window.

Attribute	Value	Description
download	<i>filename</i>	Specifies that the target will be downloaded when a user clicks on the hyperlink
href	<i>URL</i>	Specifies the URL of the page the link goes to
hreflang	<i>language_code</i>	Specifies the language of the linked document
ping	<i>list_of_URLs</i>	Specifies a space-separated list of URLs to which, when the link is followed, post requests with the body ping will be sent by the browser (in the background). Typically used for tracking.
referrerpolicy	no-referrer no-referrer-when-downgrade origin origin-when-cross-origin same-origin strict-origin-	Specifies which referrer information to send with the link

	when-cross-origin unsafe-url	
rel	alternate author bookmark external help license next nofollow noreferrer noopener prev search tag	Specifies the relationship between the current document and the linked document
target	_blank _parent _self _top	Specifies where to open the linked document
type	<i>media_type</i>	Specifies the media type of the linked document

shape : This attribute is used to define a selectable region for hypertext source links associated with a figure in order to create an image map. The values for the attribute are **circle**, **default**, **polygon**, and **rect**.

The format of the **coords** attribute depends on the value of **shape**. For **circle**, the value is x,y,r , where x and y are the pixel coordinates for the center of the circle and r is the radius value in pixels. For **rect**, the **coords** attribute should be x,y,w,h . The x,y values define the upper-left corner of the rectangle, while w and h define the width and height, respectively.

A value of **polygon** for **shape** requires $x1,y1,x2,y2,\dots$ values for **coords**. Each of the x,y pairs defines a point in the polygon, with successive points being joined by straight lines and the last point joined to the first. The value of **default** for **shape** requires that the entire enclosed area, typically an image, be used.

coords : For use with object shapes, this attribute uses a comma-separated list of numbers to define the coordinates of the object on the page.

Examples:

```
<!-- anchor linking to external file -->
```

```
<a href="http://www.democompany.com/">External Link</a>
```

```
<!-- anchor linking to file on local file system -->
```

```
<a href="file://c:\html\index.html">local file link</a>
```

```
<!-- anchor invoking anonymous FTP -->
```

```
<a href="ftp://ftp.democompany.com/freestuff">Anonymous FTP link</a>
```

```

-
<!-- anchor invoking FTP with password -->
<a href="ftp://joeuser:secretpassword@democompany.com/path/file"> FTP with password</a>

<!-- anchor invoking mail -->
<a href="mailto:fakeid@democompany.com">Send mail</a>

<!-- anchor used to define target destination within document -->
<a name="jump">Jump target</a>

<!-- anchor linking internally to previous target anchor -->
<a href="#jump">Local jump within document</a>

<!-- anchor linking externally to previous target anchor -->
<a href="http://www.democompany.com/document#jump"> Remote jump to a position within a
document</a>

```

Images

This element indicates a media object to be included in an (X)HTML document. Usually, the object is a bitmap graphic image, but some implementations support movies, vector formats, and animations.

Or

The HTML **** tag is used to embed an image in a web page.

Images are not technically inserted into a web page; images are linked to web pages. The **** tag creates a holding space for the referenced image.

The **** tag is empty element.

Attributes:

Standard Syntax:

```



```

align: This attribute controls the horizontal alignment of the image with respect to the page. The default value is **left**.

src : This attribute indicates the URL of an image file to be displayed. Most browsers will display .gif, .jpeg, and .png files directly.

Alt: This attribute contains a string to be displayed instead of the image for browsers that cannot display images.

Border: This attribute indicates the width, in pixels, of the border surrounding the image.

Dynsrc: In the Microsoft implementation, this attribute indicates the URL of a movie file and is used instead of the **src** attribute. Common formats used here are .avi (Audio-Visual Interleaved), .mov (QuickTime), and .mpg and .mpeg (Motion Picture Experts Group).

Ismap: This attribute indicates that the image is a server-side image map. User mouse actions over the image are sent to the server for processing.

longdesc This attribute specifies the URL of a document that contains a long description of the image. This attribute is used as a complement to the alt attribute.

loop In the Microsoft implementation, this attribute is used with the dynsrc attribute to cause a movie to loop. Its value is either a numeric loop count or the keyword infinite.

lowsrc This nonstandard attribute, supported in most browsers, contains the URL of an image to be initially loaded. Typically, the lowsrc image is a low-resolution or black-and-white image that provides a quick preview of the image to follow. Once the primary image is loaded, it replaces the lowsrc image.

Name: This common attribute is used to bind a name to the image.

start : In the Microsoft implementation, this attribute is used with the dynsrc attribute to indicate when a movie should play. The default value, if no value is specified, is to play the video as soon as it has finished loading. Alternatively, a value of **mouseover** can be set to play the movie once the user has moved their mouse over the video.

usemap This attribute makes the image support client-side image mapping. Its argument is a URL specifying the map file, which associates image regions with hyperlinks. The URL is generally a fragment identifier that references a location in the current document rather than a remote resource.

HTML Image Maps:

With HTML image maps, you can create clickable areas on an image.

The HTML `<map>` tag defines an image map. An image map is an image with clickable areas. The areas are defined with one or more `<area>` tags.

The Image

The image is inserted using the `` tag. The only difference from other images is that you must add a **usemap** attribute:

```

```

The **usemap** value starts with a hash tag `#` followed by the name of the image map, and is used to create a relationship between the image and the image map.

Create Image Map

-
Then, add a `<map>` element.

The `<map>` element is used to create an image map, and is linked to the image by using the required `name` attribute:

Shape="rect"

The coordinates for `shape="rect"` come in pairs, one for the x-axis and one for the y-axis.

So, the coordinates `34,44` is located 34 pixels from the left margin and 44 pixels from the top:

The coordinates `270,350` is located 270 pixels from the left margin and 350 pixels from the top:

```
<area shape="rect" coords="34, 44, 270, 350" href="computer.htm">
```

Categories of Attributes:

Core Attributes:

The four core attributes that can be used on the majority of HTML elements (although not all) are:

- Id
- Title
- Class
- Style

➤ The Id Attribute

The **id** attribute of an HTML tag can be used to uniquely identify any element within an HTML page. There are two primary reasons that you might want to use an id attribute on an element:

- If an element carries an id attribute as a unique identifier, it is possible to identify just that element and its content.
- If you have two elements of the same name within a Web page (or style sheet), you can use the id attribute to distinguish between elements that have the same name.

Example

```
<p id="html">This para explains what is HTML</p>  
<p id="css">This para explains what is Cascading Style Sheet</p>
```

➤ The title Attribute

The **title** attribute gives a suggested title for the element. The syntax for the **title** attribute is similar as explained for **id** attribute:

The behavior of this attribute will depend upon the element that carries it, although it is often displayed as a tooltip when cursor comes over the element or while the element is loading.

Example:

```
<html>
<head>
<title>The title Attribute Example</title>
</head>
<body>
<h3 title="Hello HTML!">Titled Heading Tag Example</h3>
</body>
</html>
```

This will produce the following result:
Titled Heading Tag Example

Now try to bring your cursor over "Titled Heading Tag Example" and you will see that whatever title you used in your code is coming out as a tooltip of the cursor.

➤ **The class Attribute**

The class attribute is used to associate an element with a style sheet, and specifies the class of element. You will learn more about the use of the class attribute when you will learn Cascading Style Sheet (CSS). So for now you can avoid it.

The value of the attribute may also be a space-separated list of class names.

For example:

```
class="className1 className2 className3"
```

➤ **The style Attribute**

The style attribute allows you to specify Cascading Style Sheet (CSS) rules within the element.

```
<!DOCTYPE html>
<html>
<head>
<title>The style Attribute</title>
</head>
<body>
<p style="font-family:arial; color:#FF0000;">Some text...</p>
```

-
</body>
</html>

This will produce the following result:

Some text...

Internationalization Attributes:

There are three internationalization attributes, which are available for most (although not all) XHTML elements.

- dir
- lang
- xml:lang

The dir Attribute

The dir attribute allows you to indicate to the browser about the direction in which the text should flow. The dir attribute can take one of two values, as you can see in the table that follows:

Value Meaning

ltr Left to right (the default value)

rtl Right to left (for languages such as Hebrew or Arabic that are read right to left).

Example:

```
<!DOCTYPE html>
<html dir="rtl">
<head>
<title>Display Directions</title>
</head>
<body>
This is how IE 5 renders right-to-left directed text.
</body>
</html>
```

This will produce the following result:

This is how IE 5 renders right-to-left directed text.

The lang Attribute:

The lang attribute allows you to indicate the main language used in a document, but this attribute was kept in HTML only for backwards compatibility with earlier versions of HTML. This attribute has been replaced by the xml:lang attribute in new XHTML documents.

The values of the lang attribute are ISO-639 standard two-character language codes. Check HTML Language Codes: ISO 639 for a complete list of language codes.

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>English Language Page</title>
</head>
<body>
This page is using English Language
</body>
</html>
```

Generic Attributes:

Here's a table of some other attributes that are readily usable with many of the HTML tags.

Attribute	Options	Function
align	right, left, center	Horizontally aligns tags
valign	top, middle, bottom	Vertically aligns tags within an HTML element.
bgcolor	numeric, hexadecimal, RGB values	Places a background color behind an element
background	URL	Places a background image behind an element
id	User Defined	Names an element for use with Cascading Style Sheets.
class	User Defined	Classifies an element for use with Cascading Style Sheets.
width	Numeric Value	Specifies the width of tables, images, or table cells.
height	Numeric Value	Specifies the height of tables, images, or table cells.
title	User Defined	"Pop-up" title of the elements.

HTML LIST

HTML List is a collection of related information.

The lists can be ordered or unordered depending on the requirement. In html we can create both order and unordered lists by using [](#) and [](#) tags. There is one more list which is description list -

-
HTML [`<dl>`](#), [`<dt>`](#) & [`<dd>`](#) tag are used to create description lists.

three ways for specifying lists of information namely ordered, unordered, and definition lists. All lists must contain one or more list items. Items are marked by ``.

HTML Unorder Lists

Unorder lists are marked with bullet points, by using html [``](#) & [``](#) tag we can create a unordered list. This is also know as unordered list.

```
<!DOCTYPE html>
<html>
<head>
  <title>HTML List</title>
</head>
<body>
  <h2>Example of HTML List</h2>
  <ul>
    <li>HTML</li>
    <li>CSS</li>
    <li>JavaScript</li>
    <li>Java</li>
    <li>JavaFX</li>
  </ul>
</body>
</html>
```

Output:

Example of HTML List

- HTML
- CSS
- JavaScript
- Java
- JavaFX

By default, list items are preceded with bullet symbol. Using type attribute we can have other symbols also such as disc, square.

HTML Order Lists:

-
Order list are marked with numbers by default, we can change the number into alphabet, roman numbers, etc. By using html `` & `` tag we can create a order list and using type attribute we can change the default numeric marking.

```
<!DOCTYPE html>
<html>
<head>
  <title>HTML List</title>
</head>
<body>
  <h2>Example of HTML List</h2>
  <ol>
    <li>HTML</li>
    <li>CSS</li>
    <li>JavaScript</li>
    <li>Java</li>
    <li>JavaFX</li>
  </ol>
</body>
</html>
```

Output:

Example of HTML List

1. HTML
2. CSS
3. JavaScript
4. Java
5. JavaFX

By default, list items are preceded with decimal number symbol. Using type attribute, we can have other symbols also such as roman, alphabets.

HTML Description Lists

Description list is list of items with description, to create a description list we can use `<dl>`, `<dt>` & `<dd>` tag. In which `<dl>` refers to the description list.

`<dt>` refers to the data term. `<dd>` refers to the data definition.

```

<!DOCTYPE html>
<html>
<head>
  <title>HTML List</title>
</head>
<body>
  <h2>Example of HTML List</h2>
  <dl>
    <dt>HTML</dt>
    <dd>HyperText markup language</dd>
    <dt>CSS</dt>
    <dd>Cascading Style Sheet</dd>
    <dt>JS</dt>
    <dd>JavaScript</dd>
  </dl>
</body>
</html>

```

Example of HTML List

HTML

HyperText markup language

CSS

Cascading Style Sheet

JS

JavaScript

HTML TABLES

An HTML Table is an arrangement (structured set) of data in rows and columns in tabular format. Tables are useful for various tasks, such as presenting text information and numerical data.

Tags used in HTML Tables:

HTML Tags	Descriptions
<u><table></u>	Defines the structure for organizing data in rows and columns within a web page.
<u><tr></u>	Represents a row within an HTML table, containing individual cells.

HTML Tags	Descriptions
<u><th></u>	Shows a table header cell that typically holds titles or headings.
<u><td></u>	Represents a standard data cell, holding content or data.
<u><caption></u>	Provides a title or description for the entire table.
<u><thead></u>	Defines the header section of a table, often containing column labels.
<u><tbody></u>	Represents the main content area of a table, separating it from the header or footer.
<u><tfoot></u>	Specifies the footer section of a table, typically holding summaries or totals.
<u><col></u>	Defines attributes for table columns that can be applied to multiple columns at once.
<u><colgroup></u>	Groups together a set of columns in a table to which you can apply formatting or properties collectively.

Defining Tables in HTML:

An HTML table is defined with the “table” tag. Each table row is defined with the “tr” tag. A table header is defined with the “th” tag. By default, table headings are bold and centered. A table data/cell is defined with the “td” tag.

Table Cells:

Table Cell are the building blocks for defining the Table. It is denoted with `<td>` as a start tag & `</td>` as a end tag.

Syntax:

```
</td> Content...</td>
```

Table Rows:

The rows can be formed with the help of combination of Table Cells. It is denoted by `<tr>` and `</tr>` tag as a start & end tags.

Syntax:

```
</tr> Content...</tr>
```

Table Headers

The Headers are generally use to provide the Heading. The Table Headers can also be used to add the heading to the Table. This contains the <th> & </th> tags.

Syntax:

</th> Content...</th>

Example 1: Creating a simple table in HTML using a table tag.

```
<!-- index.html -->
<!DOCTYPE html>
<html>

<body>
  <table>
    <tr>
      <th>Book Name</th>
      <th>Author Name</th>
      <th>Genre</th>
    </tr>
    <tr>
      <td>The Book Thief</td>
      <td>Markus Zusak</td>
      <td>Historical Fiction</td>
    </tr>
    <tr>
      <td>The Cruel Prince</td>
      <td>Holly Black</td>
      <td>Fantasy</td>
    </tr>
    <tr>
      <td>The Silent Patient</td>
      <td> Alex Michaelides</td>
      <td>Psychological Fiction</td>
    </tr>
  </table>
</body>
</html>
```

Attribute	Description
accesskey	Specifies a shortcut key to activate/focus an element
class	Specifies one or more classnames for an element (refers to a class in a style sheet)

dir	Specifies the text direction for the content in an element
draggable	Specifies whether an element is draggable or not
id	Specifies a unique id for an element
lang	Specifies the language of the element's content
spellcheck	Specifies whether the element is to have its spelling and grammar checked or not
style	Specifies an inline CSS style for an element
title	Specifies extra information about an element
border	Border for the table cells.
rowspan	Used to combine row cells.
colspan	Used to combine column cells.

example:

```
<colgroup>
  <col span="2" style="background-color: #D6EEEE">
</colgroup>
```

<colgroup> will apply background color will be applied to two columns.

HTML Div Element

The <div> element is used as a container for other HTML elements. Defines a section in a document (block-level)

The <div> Element:

The <div> element is by default a block element, meaning that it takes all available width, and comes with line breaks before and after.

Example:

```
<div>
  <h2>Motivation</h2>
  <p>Your present circumstances don't determine where you can go; they merely determine where you start.</p>
  <p>Start where you are. Use what you have. Do what you can</p>
</div>
```

<div> as a container:

The <div> element is often used to group sections of a web page together.

Center align a <div> element:

If you have a <div> element that is not 100% wide, and you want to center-align it, set the CSS **margin** property to **auto**.

Example:

```
<style>
div {
  width:300px;
  margin:auto;
}
</style>
```

Multiple <div> elements

You can have many <div> containers on the same page.

Aligning <div> elements side by side

When building web pages, you often want to have two or more <div> elements side by side, like this:

There are different methods for aligning elements side by side, all include some CSS styling. We will look at the most common methods:

Float	Inline-block	Flex
<p>The CSS float property was not originally meant to align <div> elements side-by-side, but has been used for this purpose for many years.</p> <p>The CSS float property is used for positioning and formatting content and allow elements float next</p>	<p>If you change the <div> element's display property from block to inline-block, the <div> elements will no longer add a line break before and after, and will be displayed side by side instead of on top of each other.</p> <pre><style> div { width: 30%;</pre>	<p>The CSS Flexbox Layout Module was introduced to make it easier to design flexible responsive layout structure without using float or positioning.</p> <p>To make the CSS flex method work, surround the <div> elements with another <div> element</p>

<p>to each other instead of on top of each other.</p> <p>Example</p> <p>How to use float to align div elements side by side:</p> <pre> <style> .mycontainer { width:100%; overflow:auto; } .mycontainer div { width:33%; float:left; } </style> </pre>	<pre> display: inline-block; } </style> </pre>	<p>and give it the status as a flex container.</p> <p>Example</p> <p>How to use flex to align div elements side by</p> <pre> <style> .mycontainer { display: flex; } .mycontainer > div { width:33%; } </style> </pre>
---	--	--

Grid

The CSS Grid Layout Module offers a grid-based layout system, with rows and columns, making it easier to design web pages without having to use floats and positioning.

Sounds almost the same as flex, but has the ability to define more than one row and position each row individually.

The CSS grid method requires that you surround the `<div>` elements with another `<div>` element and give the status as a grid container, and you must specify the width of each column.

Example

How to use grid to align `<div>` elements side by side:

```

<style>
.grid-container {
  display: grid;
  grid-template-columns: 33% 33% 33%;
}
</style>

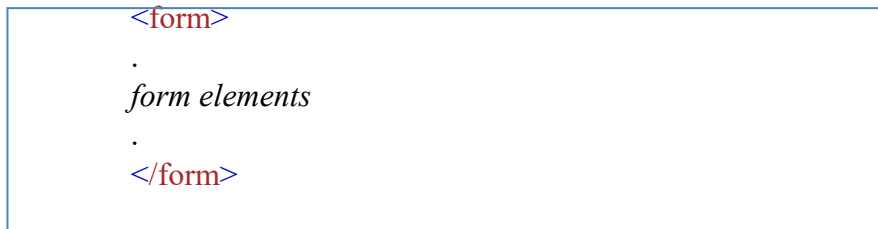
```

HTML Forms

An HTML form is used to collect user input. The user input is most often sent to a server for processing.

The <form> Element

The HTML <form> element is used to create an HTML form for user input:



The <form> element is a container for different types of input elements, such as: text fields, checkboxes, radio buttons, submit buttons, etc.

The HTML <form> element can contain one or more of the following form elements:

- <input>
- <label>
- <select>
- <textarea>
- <button>
- <fieldset>
- <legend>
- <datalist>
- <output>
- <option>
- <optgroup>

HTML Form Attributes:

The Action Attribute:

The **action** attribute defines the action to be performed when the form is submitted.

Usually, the form data is sent to a file on the server when the user clicks on the submit button.

Example:

```
<form action="/login.js">
<label for="fname">First name:</label><br>
  <input type="text" id="fname" name="fname" value="John"><br>
  <input type="submit" value="Submit">
</form>
```

The Target Attribute:

The **target** attribute specifies where to display the response that is received after submitting the form.

The **target** attribute can have one of the following values:

Value	Description
_blank	The response is displayed in a new window or tab
_self	The response is displayed in the current window
_parent	The response is displayed in the parent frame
_top	The response is displayed in the full body of the window
framename	The response is displayed in a named iframe

The Method Attribute:

The **method** attribute specifies the HTTP method to be used when submitting the

form data.

The form-data can be sent as URL variables (with **method="get"**) or as HTTP post transaction (with **method="post"**).

The default HTTP method when submitting form data is GET.

Example

This example uses the GET method when submitting the form data:

```
<form action="/action_page.js" method="get">
```

Notes on GET:

- Appends the form data to the URL, in name/value pairs
- NEVER use GET to send sensitive data! (the submitted form data is visible in the URL!)
- The length of a URL is limited (2048 characters)
- Useful for form submissions where a user wants to bookmark the result
- GET is good for non-secure data, like query strings in Google

Notes on POST:

- Appends the form data inside the body of the HTTP request (the submitted form data is not shown in the URL)
- POST has no size limitations, and can be used to send large amounts of data.
- Form submissions with POST cannot be bookmarked

The Autocomplete Attribute

The **autocomplete** attribute specifies whether a form should have autocomplete on or off.

-
When autocomplete is on, the browser automatically complete values based on values that the user has entered before.

The Novalidate Attribute

The **novalidate** attribute is a boolean attribute.

When present, it specifies that the form-data (input) should not be validated when submitted.

HTML FORM ELEMENTS

1.HTML Input Types:

different input types you can use in HTML

SYNTAX	DESCRIPTION
<code><input type="text"></code>	defines a single-line text input field
<code><input type="password"></code>	defines a password field
<code><input type="submit"></code>	defines a button for submitting form data to a form-handler. The form-handler is typically a server page with a script for processing input data. The form-handler is specified in the form's action attribute.
<code><input type="reset"></code>	defines a reset button that will reset all form values to their default values
<code><input type="radio"></code>	defines a radio button. Radio buttons let a user select ONLY ONE of a limited number of choices
<code><input type="checkbox"></code>	defines a checkbox. Checkboxes let a user select ZERO or MORE options of a limited number of choices
<code><input type="button"></code>	defines a button.
<code><input type="color"></code>	is used for input fields that should contain a colour. Depending on browser support, a color picker can show up in the input field
<code><input type="date"></code>	is used for input fields that should contain a date. Depending on browser support, a date picker can show up in the input field. You can also use the min and max attributes to add restrictions to dates. EX: max="1979-12-31"
<code><input type="datetime-local"></code>	specifies a date and time input field, with no time zone
<code><input type="email"></code>	is used for input fields that should contain an e-mail address
<code><input type="image"></code>	defines an image as a submit button. The path to the image is specified in the src attribute. EX: <code><input type="image" src="img_submit.gif" alt="Submit" width="48" height="48"></code>
<code><input type="file"></code>	defines a file-select field and a "Browse" button for file uploads.
<code><input type="hidden"></code>	defines a hidden input field (not visible to a user)
<code><input type="month"></code>	allows the user to select a month and year.

<input type="number">	defines a numeric input field. EX: <input type="number" id="quantity" name="quantity" min="1" max="5">
<input type="range">	defines a control for entering a number whose exact value is not important (like a slider control). Default range is 0 to 100
<input type="search">	is used for search fields (a search field behaves like a regular text field)
<input type="tel">	is used for input fields that should contain a telephone number. EX: <input type="tel" id="phone" name="phone" pattern="[0-9]{3}-[0-9]{2}-[0-9]{3}">
<input type="time">	allows the user to select a time (no time zone).
<input type="url">	is used for input fields that should contain a URL address.
<input type="week">	allows the user to select a week and year.

HTML Input Attributes:

different attributes for the HTML **<input>** element.

ATTRIBUTE	DESCRIPTION
value	attribute specifies an initial value(default) for an input field. Ex: <input type="text" id="fname" name="fname" value="John">
readonly	attribute specifies that an input field is read-only. A read-only input field cannot be modified. Ex: <input type="text" id="fname" name="fname" value="John" readonly>
disabled	attribute specifies that an input field should be disabled. Ex: <input type="text" id="fname" name="fname" value="John" disabled>
size	attribute specifies the visible width, in characters, of an input field. The default value for size is 20. EX: <input type="text" id="fname" name="fname" size="50">
maxlength	attribute specifies the maximum number of characters allowed in an input field. EX: <input type="text" id="pin" name="pin" maxlength="4" size="4">
min and max	attributes specify the minimum and maximum values for an input field. The min and max attributes work with the following input types: number, range, date, datetime-local, month, time and week.
multiple	attribute specifies that the user is allowed to enter more than one value in an input field. for input types: email, and file
pattern	attribute specifies a regular expression that the input field's value is checked against, when the form is submitted. pattern attribute works with the following input types: text, date, search, url, tel, email, and password. Ex: <input type="text" id="country_code" name="country_code" pattern="[A-Za-z]{3}" title="Three letter country code">

placeholder	<p>attribute specifies a short hint that describes the expected value of an input field (a sample value or a short description of the expected format).</p> <p>The short hint is displayed in the input field before the user enters a value. The placeholder attribute works with the following input types: text, search, url, number, tel, email, and password.</p> <p>Ex: <code><input type="tel" id="phone" name="phone" placeholder="123-45-678"></code></p>
required	<p>attribute specifies that an input field must be filled out before submitting the form. The required attribute works with the following input types: text, search, url, tel, email, password, date pickers, number, checkbox, radio, and file</p> <p>Ex: <code><input type="text" id="username" name="username" required></code></p>
step	<p>attribute specifies the legal number intervals for an input field.</p> <p>Example: if <code>step="3"</code>, legal numbers could be -3, 0, 3, 6, etc.</p> <p>Ex: <code><input type="number" id="points" name="points" step="3"></code></p>
autofocus	<p>attribute specifies that an input field should automatically get focus when the page loads.</p> <p>Ex: <code><input type="text" id="fname" name="fname" autofocus></code></p>
height and width	<p>attributes specify the height and width of an <code><input type="image"></code> element.</p> <p>Ex: <code><input type="image" src="img_submit.gif" alt="Submit" width="48" height="48"></code></p>
list and <datalist>	<p>element that contains pre-defined options for an <code><input></code> element.</p>
autocomplete	<p>attribute specifies whether a form or an input field should have autocomplete on or off.</p>

2. <label> Element:

The `<label>` element defines a label for several form elements.

The `for` attribute of the `<label>` tag should be equal to the `id` attribute of the `<input>` element to bind them together.

3.<select> Element:

The `<select>` element defines a drop-down list. The `<option>` element defines an option that can be selected. By default, the first item in the drop-down list is selected. To define a pre-selected option, add the `selected` attribute to the option. Use the `size` attribute to specify the number of visible values:

```
<label for="B.TECH">Choose Programme</label>
<select id="course" name="course" size=3 multiple>
  <option value="CSE">CSE</option>
  <option value="IT">IT</option>
  <option value="AIML" selected>AIML</option>
  <option value="DS">DATASCIENCE</option>
</select>
```

Use the `multiple` attribute to allow the user to select more than one value.

4. <textarea> Element:

The <textarea> element defines a multi-line input field (a text area). The **rows** attribute specifies the visible number of lines in a text area. The **cols** attribute specifies the visible width of a text area.

```
<textarea name="message" rows="10" cols="30">  
write content here  
</textarea>
```

5. <button> Element:

The <button> element defines a clickable button.

```
<button type="button" onclick="alert('Hello World!')">Click Me!</button>
```

6.<fieldset> and <legend> Elements:

The <fieldset> element is used to group related data in a form.

The <legend> element defines a caption for the <fieldset> element.

CSS

CSS (Cascading Style Sheets) is a language designed to simplify the process of making web pages presentable. It allows you to apply styles to HTML documents, describing how a webpage should look by prescribing colors, fonts, spacing, and positioning. CSS provides developers and designers with powerful control over the presentation of HTML elements.

HTML uses tags and CSS uses rulesets. CSS styles are applied to the HTML element using selectors. CSS is easy to learn and understand, but it provides powerful control over the presentation of an HTML document.

Syntax

CSS consists of style rules that are interpreted by the browser and applied to the corresponding elements. A style rule set includes a selector and a declaration block.

- **Selector:** Targets specific HTML elements to apply styles.
- **Declaration:** Combination of a property and its corresponding value.

```
<h1>GeeksforGeeks</h1>
```

```
// CSS Style
```

```
h1 { color: blue; font-size: 12px; }
```

Where -

Selector - h1

Declaration - { color: blue; font-size: 12px; }

- The selector points to the HTML element that you want to style.
- The declaration block contains one or more declarations separated by semicolons.
- Each declaration includes a CSS property name and a value, separated by a colon.
-

Types of CSS

CSS (Cascading Style Sheets) is used to style and layout of web pages, and controlling the appearance of HTML elements. CSS targets HTML elements and applies style rules to dictate their appearance.

Below are the types of CSS:

- Inline CSS
- Internal or Embedded CSS
- External CSS

1. Inline CSS

Inline CSS involves applying styles directly to individual HTML elements using the style attribute. This method allows for specific styling of elements within the HTML document, overriding any external or internal styles.

```
<p style="color:#009900;  
font-size:50px;  
font-style:italic;  
text-align:center;">
```

Inline CSS

```
</p>
```

Output:

Inline CSS

2. Internal or Embedded CSS

[Internal or Embedded CSS](#) is defined within the HTML document's <style> element. It applies styles to specified HTML elements. The CSS rule set should be within the HTML file in the head section i.e. the CSS is embedded within the <style> tag inside the head section of the HTML file.

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
<style>
```

```
.main {  
    text-align: center;  
}
```

```
.GFG {  
    color: #009900;  
    font-size: 50px;  
    font-weight: bold;  
}
```

```
.geeks {  
    font-style: bold;  
    font-size: 20px;
```

```

-
}

</style>

</head>

<body>

  <div class="main">

    <div class="GFG">Internal CSS</div>

    <div class="geeks">

      Implementation of Internal CSS

    </div>

  </div>

</body>

</html>

```

3. External CSS

[External CSS](#) contains separate CSS files that contain only style properties with the help of tag attributes (For example class, id, heading, ... etc). CSS property is written in a separate file with a .css extension and should be linked to the HTML document using a **link** tag. It means that, for each element, style can be set only once and will be applied across web pages.

Html File

```

<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <div class="main">
    <div class="GFG">External CSS </div>
    <div id="geeks">
      This shows implementation of External CSS
    </div>
  </div>
</body>
</html>

```

CSS file

```

body {

```

```

-
background-color: powderblue;
}
.main {
  text-align: center;
}
.GFG {
  color: #009900;
  font-size: 50px;
  font-weight: bold;
}
#geeks {
  font-style: bold;
  font-size: 20px;
}

```

CSS Selectors

CSS selectors target the HTML elements on your pages, allowing you to add styles based on their ID, class, type, attribute, and more. This guide will help you to understand the intricacies of CSS selectors and their important role in enhancing the user experience of your web pages. Understanding these selectors—such as the universal selector, attribute selector, pseudo-class selector, and combinatory selectors—enables more efficient and dynamic web design.

Types of CSS Selectors

CSS selectors come in various types, each with its unique way of selecting HTML elements. Let's explore them:

CSS Selectors	Description
Simple Selectors	It is used to select the HTML elements based on their element name, id, attributes, etc
<u>Universal Selector</u>	Selects all elements on the page.
<u>Attribute Selector</u>	Targets elements based on their attribute values.
<u>Pseudo-Class Selector</u>	Selects elements based on their state or position, such as :hover for hover effects.
<u>Combinator Selectors</u>	Combine selectors to specify relationships between elements, such as descendants () or child (>).
<u>Pseudo-Element Selector</u>	Selects specific parts of an element, such as ::before or ::after.

CSS Background

The CSS background is the area behind an element's content, which can be a color, image, or both. The background property lets you control these aspects, including color, image, position, and repetition.

CSS Background Property

The CSS Background is a shorthand property for the following:

Background Property	Description
<u>background-color</u>	Defines the background color of an element using color names, HEX, or RGB values.
<u>background-image</u>	Adds one or more images as the background of an element.
<u>background-repeat</u>	Specifies how the background image should be repeated—horizontally, vertically, or not at all.
<u>background-attachment</u>	Controls the scrolling behavior of the background image, making it fixed or scrollable with the page.
<u>background-position</u>	Determines the initial position of the background image within the element.
<u>background-origin</u>	Adjusts the placement of the background image relative to the padding, border, or content box.
<u>background-clip</u>	Sets how far the background (color or image) should extend within an element (e.g., to the padding or border).

CSS Borders

Borders in CSS are used to create a visible outline around an element. They can be customized in terms of

- **Width:** The thickness of the border.
- **Style:** The appearance of the border (solid, dashed, dotted, etc.).
- **Color:** The color of the border.

Syntax:

- ```
element {
 border: 1px solid black;
}
```

### CSS Border Properties

CSS provides several properties to control and customize borders:

| Property | Description |
|----------|-------------|
|----------|-------------|



| Property                      | Description                                                       |
|-------------------------------|-------------------------------------------------------------------|
| <a href="#">border-style</a>  | Determines the type of border (e.g., solid, dashed, dotted).      |
| <a href="#">border-width</a>  | Sets the width of the border (in pixels, points, or other units). |
| <a href="#">border-color</a>  | Specifies the border color.                                       |
| <a href="#">border-radius</a> | Creates rounded corners for elements.                             |

### CSS Fonts

CSS fonts control how text appears on a webpage. With CSS, you can specify various properties like font family, size, weight, style, and line height to create visually appealing and readable typography

#### Key Properties of CSS Fonts

To customize fonts effectively in web design, it's crucial to understand the main CSS font properties:

- **font-family:** Specifies the font type.
- **font-size:** Determines the size of the text.
- **font-weight:** Adjusts the thickness of the text.
- **font-style:** Controls the slant of the text (e.g., italic).
- **line-height:** Sets the space between lines of text.
- **letter-spacing:** Modifies the space between characters.
- **text-transform:** Controls the capitalization of text.

### CSS Text Formatting

CSS Text Formatting allows you to control the visual presentation of text on a webpage. From changing fonts to adjusting spacing and alignment, CSS provides powerful properties to enhance the readability and aesthetic of text.

- CSS lets you adjust font properties, text alignment, spacing, and decorations.
- It helps in creating visually appealing text and improving the user experience.
- Various text-related properties can be combined to achieve unique text styles and layouts.

#### CSS Text Formatting Properties

These are the following text formatting properties.

| Property                        | Description                                                              |
|---------------------------------|--------------------------------------------------------------------------|
| <a href="#">text-color</a>      | Sets the color of the text using color name, hex value, or RGB value.    |
| <a href="#">text-align</a>      | Specifies horizontal alignment of text in a block or table-cell element. |
| <a href="#">text-align-last</a> | Sets alignment of last lines occurring in an element.                    |

| Property                                     | Description                                                                   |
|----------------------------------------------|-------------------------------------------------------------------------------|
| <a href="#"><u>text-decoration</u></a>       | Decorates text content.                                                       |
| <a href="#"><u>text-decoration-color</u></a> | Sets color of text decorations like overlines, underlines, and line-throughs. |
| <a href="#"><u>text-decoration-line</u></a>  | Sets various text decorations like underline, overline, line-through.         |
| <a href="#"><u>text-decoration-style</u></a> | Combines text-decoration-line and text-decoration-color properties.           |
| <a href="#"><u>text-indent</u></a>           | Indents first line of paragraph.                                              |
| <a href="#"><u>text-justify</u></a>          | Justifies text by spreading words into complete lines.                        |
| <a href="#"><u>text-overflow</u></a>         | Specifies hidden overflow text.                                               |
| <a href="#"><u>text-transform</u></a>        | Controls capitalization of text.                                              |
| <a href="#"><u>text-shadow</u></a>           | Adds shadow to text.                                                          |
| <a href="#"><u>letter-spacing</u></a>        | Specifies space between characters of text.                                   |
| <a href="#"><u>line-height</u></a>           | Sets space between lines.                                                     |
| <a href="#"><u>direction</u></a>             | Sets text direction.                                                          |
| <a href="#"><u>word-spacing</u></a>          | Specifies space between words of line.                                        |

## CSS Tables

Tables in CSS are used to style HTML table elements, allowing data to be presented in a structured, organized format with rows and columns. CSS provides a variety of properties that can be applied to tables to enhance their appearance and functionality.

## CSS Table Properties

### 1. Border

The border property defines the appearance of borders around table elements (e.g., table, tr, td, th). It specifies the border's width, style, and color.

-  
**Syntax:**

```
border: table_width table_color;
```

## 2. Border Collapse

The [border-collapse](#) property controls whether the borders of adjacent cells are merged into a single border or kept separate.

**Syntax:**

```
border-collapse: collapse/separate;
```

## 3. Border Spacing

[Border Spacing](#) property specifies the distance between the borders of adjacent cells when border-collapse is set to separate.

**Syntax:**

```
border-spacing: value;
```

## 4. Caption Side

[Caption Side](#) property specifies the placement of the table caption relative to the table.

**Syntax:**

```
caption-side: top/bottom;
```

## 5. Empty cells

[Empty cells](#) property specifies whether or not to display borders and background on empty cells in a table.

**Syntax:**

```
empty-cells: show/hide;
```

## CSS Box Model

The **CSS Box Model** is very important to understand how elements are structured, styled, and spaced on a webpage. By understanding the Box Model, developers can create attractive layouts that adapt seamlessly to various screen sizes and devices.

What is the CSS Box Model?

The CSS Box Model is a layout model that describes how different components of a web element (**content, padding, border, and margin**) are structured and positioned. Each web element generates a rectangular box that encompasses these components, and the Box Model allows developers to control the element's size and spacing effectively.

#### Key Components of the Box Model

The CSS Box Model consists of four primary components:

##### 1. Content Area

- The content area is where the actual content, such as text, images, or other media, is displayed.
- It is sized using the width and height properties.
- The boundary of the content area is known as the content edge.

##### 2. Padding Area

- The padding surrounds the content area and creates space inside the border.
- It can be adjusted using the padding property (or padding-top, padding-right, padding-bottom, and padding-left for individual sides).
- The padding area increases the overall size of the element without changing the content area.

##### 3. Border Area

- The border wraps around the padding and content, defining the edge of the element.
- It can be styled using properties such as border width border **color, border style, and border-color**.
- The width of the border affects the overall size of the element.

##### 4. Margin Area

- The margin is the outermost space that separates the element from adjacent elements.
- It can be set using the margin property (or margin-top, margin-right, margin-bottom, and margin-left for individual sides).
- Unlike padding and border, margin does not increase the element's total size but affects its placement on the page.

The following figure illustrates the **Box model** in CSS.

### CSS Box-Model Property



## Version Control

### What Is Git Version Control?

In software development, keeping track of changes, managing multiple versions of code, and collaborating seamlessly across teams is very important. This is where version control systems (VCS) come into play, and Git is one of the most popular version control systems used today.

Whether you're working on a personal project or part of a large-scale development team, Git helps you manage your codebase effectively, ensuring that changes are tracked, versions are maintained, and collaboration is smooth.

### What is Version Control?

Before diving into Git, it's important to understand the concept of **version control**. In simple terms, version control is a system that tracks changes made to files over time. It allows developers to:

- **Save and track changes:** Every modification made to the codebase is recorded.
- **Revert to previous versions:** If something breaks or a feature doesn't work as expected, you can revert to a stable version.
- **Collaborate:** Multiple developers can work on the same project without overwriting each other's work.
- **Branching and Merging:** Developers can create branches for different features, work on them independently, and merge them back to the main codebase when ready.

**Git** is a powerful version control system widely used for tracking changes in source code during software development. Created by Linus Torvalds in 2005, Git has become an essential tool for developers worldwide. Understanding Git can significantly enhance your coding efficiency and collaboration.

### What is Git?

**Git** is a distributed version control system (DVCS) that allows developers to track changes in their codebase, collaborate with others, and manage different versions of their projects efficiently.

- *Git was developed by Linus Torvalds in 2005 for Linux kernel development.*
- *Git is 2.45.1 is the Lastest Versions of GIT, released on May 2024.*

### Why Use Git?

1. **Version Control:** Git helps in tracking changes, allowing you to revert to previous states if something goes wrong.
2. **Collaboration:** It enables multiple developers to work on a project simultaneously without interfering with each other's work.
3. **Backup:** Your entire project history is saved in a Git repository, providing a backup of all versions.
4. **Branching and Merging:** Git's branching model allows you to experiment with new features or bug fixes independently from the main project.
5. **Open Source Projects:** Most open source projects use Git for version control. Learning Git allows you to contribute to these projects.
6. **Industry Standard:** Git is widely used in the software industry, making it an essential skill for developers.

### Working with Git

1. **Initializing a Repository:** When you initialize a folder with Git, it becomes a repository. Git logs all changes made to a hidden folder within that repository.

2. **Staging Changes:** Git marks modified files as “staged.” Staging prepares changes for a snapshot you want to keep.
3. **Committing Changes:** Once staged changes are satisfactory, commit them. Git maintains a complete record of each commit.

### What is Github?

GitHub, a hosting service for Git repositories, allows you to access and download projects from any computer. Here’s what you can do with GitHub:

1. **Store Repositories:** GitHub hosts your repositories.
2. **Collaborate:** Work with other developers from any location.
3. **Version Control:** Manage collaborative workflows using Git and GitHub.

### Various Approaches To Use Git For Version Control

#### Approach 1: Git via Command Line

This is the most common method where developers use terminal commands to interact with Git. By utilizing git through command prompt, one has exhaustive authority over git functions.

Step 1: Install Git.

1. Download and install Git from the official website : Git Downloads.
2. After Installation, verify Git by running the following command in your terminal.

```
git --version
```

Step 2: Initialize a Git Repository

1. Navigate to your project folder in the terminal.
2. Initialize Git in the project folder by running:

```
git init
```

This creates a hidden .git folder that tracks your project.

Step 3: Staging Changes

To start tracking files, you need to stage them. This moves the files into a "staging area" before committing them:

```
git add <file-name>
```

or to add all files:

```
git add .
```

Step 4: Committing Changes

After staging, commit your changes with a message describing what you have done:

```
git commit -m "Initial commit"
```

Step 5: Viewing Commit History

You can view the history of commits using:

```
git log
```

Step 6: Creating and working with Branches

Create a new branch for a feature or experiment:

```
git checkout -b <branch-name>
```

Switch back to the main branch:

```
git checkout main
```

Step 7: Pushing to a Remote Repository

To collaborate with others, push your changes to a remote repository like GitHub:

```
git remote add origin <repository-URL>
```

```
git push -u origin main
```

### Approach 2: Git with GUI Clients

-

Many Git GUI clients provide a visual interface to work with Git repositories. Examples include [GitHub Desktop](#), [Sourcetree](#), and GitKraken. Using a GUI client is much more user-friendly because it has a graphical interface for executing Git operations.

Step 1: Install a Git GUI/Client.

Get and Install a Git GUI Client (example, GitHub Desktop, Sourcetree, GitKraken Desktop).

Step 2: Clone or Create a Repository

Launch the client program and create an image repository or clone from one on GitHub.

Step 3: Stage and Commit Changes

Files can be added to the staging area by dragging them over or using buttons in the GUI to commit changes.

Step 4: Push to Remote

Once your changes are committed, select push and this will upload the modified file(s) back to GitHub or any other remote location selected.

### Approach 3: Git in Integrated Development Environments (IDEs)

Popular IDEs like Visual Studio Code, IntelliJ IDEA, and PyCharm have built-in Git support, allowing you to perform Git operations from within the editor.

Step 1: Configure Git in the IDE

Start your IDE and switch on the built-in Git functionality (if not enabled by default) within its configuration parameters.

Step 2: Cloning or initializing repository

Use the graphical user interface options to either clone an existing repository or to initialize a new one using the option available on GUI itself.

Step 3: Use stage/commit/push

With this approach one can stage, commit and push changes to remotes via their visual interface without need for command line interaction at all.

### Branching strategies In Git

Branches are independent lines of work, stemming from the original codebase. Developers create separate branches for independently working on features so that changes from other developers don't interfere with an individual's line of work. Developers can easily pull changes from different branches and also merge their code with the main branch. This allows easier collaboration for developers working on one codebase.

Git branching strategies are essential for efficient code management and collaboration within development teams. In this comprehensive guide, we will delve into the various Git branching strategies, their benefits, implementation steps, and best practices.

#### Key Terminologies

- **Git Branch:** A parallel version of the code within a [Git repository](#), allowing for separate development and experimentation.
- **Main Branch (formerly Master Branch):** The primary branch of a Git repository where the production-ready code resides.
- **Feature Branch:** A branch created to work on a specific feature or task isolated from the main branch.
- **Merge:** The process of combining changes from one branch into another.
- **Pull Request (PR):** A request made by a developer to merge their changes into another branch, often used for code review.

- **CI/CD Pipeline:** Continuous Integration and [Continuous Deployment](#) pipeline, automating the process of building, testing, and deploying code changes.

### What Is A Branching Strategy?

A branching strategy is a strategy that software development teams adopt for writing, merging and deploying code with the help of a version control system like Git. It lays down a set of rules that aid the developers on how to go about the development process and interact with a shared codebase. Strategies like these are essential as they help in keeping project repositories organized, error free and avoid the dreaded [merge conflicts](#) when multiple developers simultaneously push and pull code from the same repository.

Encountering merge conflicts can impede the swift delivery of code, thereby obstructing the establishment and upkeep of an efficient [DevOps](#) workflow. DevOps aims to facilitate a rapid process for releasing incremental code changes. Therefore, implementing a structured branching strategy can alleviate this challenge, enabling developers to collaborate seamlessly and minimize conflicts. This approach fosters parallel workstreams within teams, promoting quicker releases and reduced likelihood of conflicts through a well-defined process for source control modifications.

The Branching strategies provides following features:

- Parallel development
- Enhanced productivity due to efficient collaboration
- Organized and structured feature releases
- Clear path for software development process
- Bug-free environment without disrupting development workflow

### Step By Step Implementation Of Creating A Branch

The following are the steps for creating a branch:

#### Step 1: Create Branch

- Create a branch with the name you want to specify, here we are naming the branch name as "new-feature".

```
git branch new-feature
```

#### Step 2: Navigate to Branch

- Now navigate to the new feature branch from the current branch with the following command:

```
git checkout new-feature
```

( or )

#### Step 3: Creating And Navigating Branch At A Time

- The following one command only helps in creating the branch and navigating to the branch.

```
git checkout -b new-feature
```

#### Step 4: Check Current Branch

- Execute the following command to check the current branch that you're on.

```
git branch
```

#### Step 5: Delete a Branch

Ensure you are present on the branch you want to delete.

```
git branch -d <branch-to-delete>
```

### Common Git Branching Strategies

The following are the common git branching strategies:

#### Gitflow Workflow

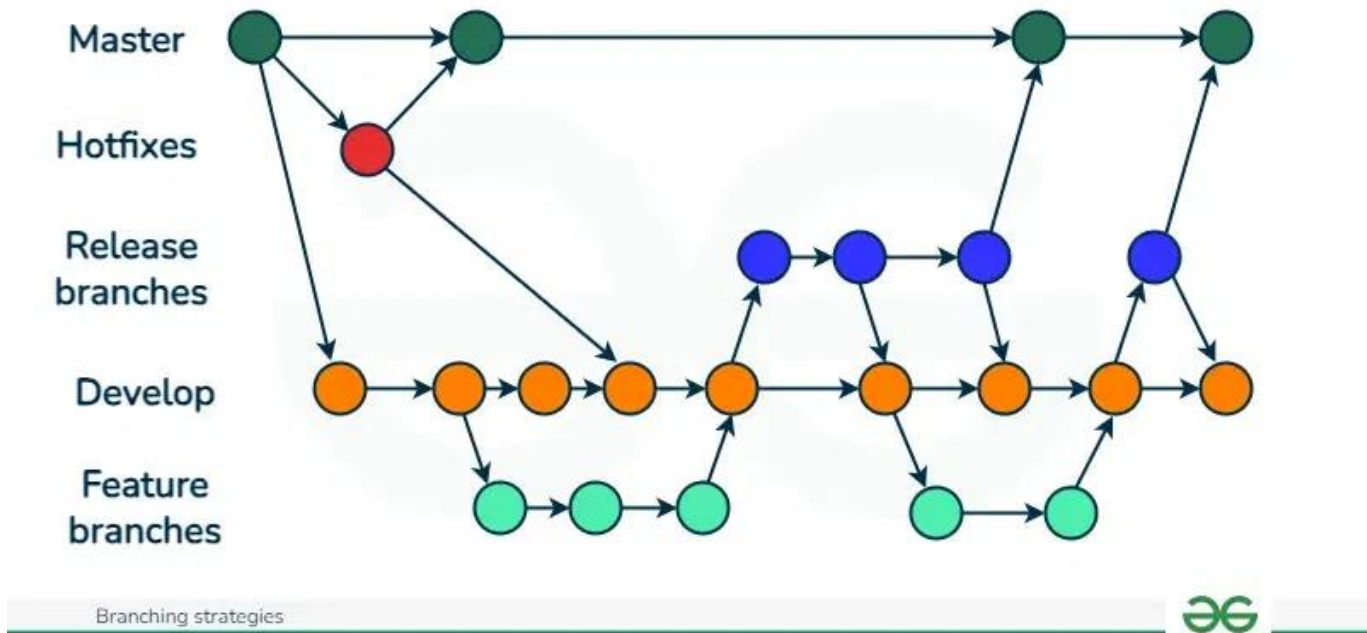
[GitFlow](#) enables parallel development, where developers can work separately on feature branches, where a feature branch is created from a [master branch](#). After completion of changes, the feature branch is merged with the master branch.

The types of branches that can be present in GitFlow are:

- **Master** - Used for product release
- **Develop** - Used for ongoing development



- **Feature Branching** - branches off the develop branch to develop new features.
  - **Release** - Assist in preparing a new production release and bug fixing, typically branched from the develop branch, and necessitating merges back into both develop and master branches.
  - **Hotfix** - Hotfix branches aid in addressing discovered bugs swiftly, allowing developers to continue their work on the develop branch while the issue is resolved. Unlike release branches, hotfix branches are created from master branch specifically for critical bug resolution in the production release.
- The Master and Develop branches are the main branches, and persist throughout the journey of the software. The other branches are essentially supporting branches and are short-lived.



### Pros Of Gitflow

- Facilitates parallel development, ensuring production code stability while developers work on separate branches.
- Organizes work effectively with separate branches for specific purposes.
- Ideal for managing multiple versions of production code.
- GitFlow streamlines the release management process, expediting the rollout of new features and bug fixes.
- By advocating for feature-based development through individual branches, GitFlow fosters independent feature implementation. This approach allows seamless merging of features into the main codebase, minimizing conflicts.
- GitFlow offers a well-defined procedure for addressing bugs and deploying hotfixes, facilitating their rapid integration into production environments.

### Cons Of Gitflow

- Complexity increases as more branches are added, potentially leading to difficulties in management.
- Merging changes from development branches to the main branch requires multiple steps, increasing the chance of errors and merge conflicts.
- Debugging issues becomes challenging due to the extensive [commit history](#).
- GitFlow's complexity may slow down the development process and release cycle, making it less suitable for continuous integration and continuous delivery.

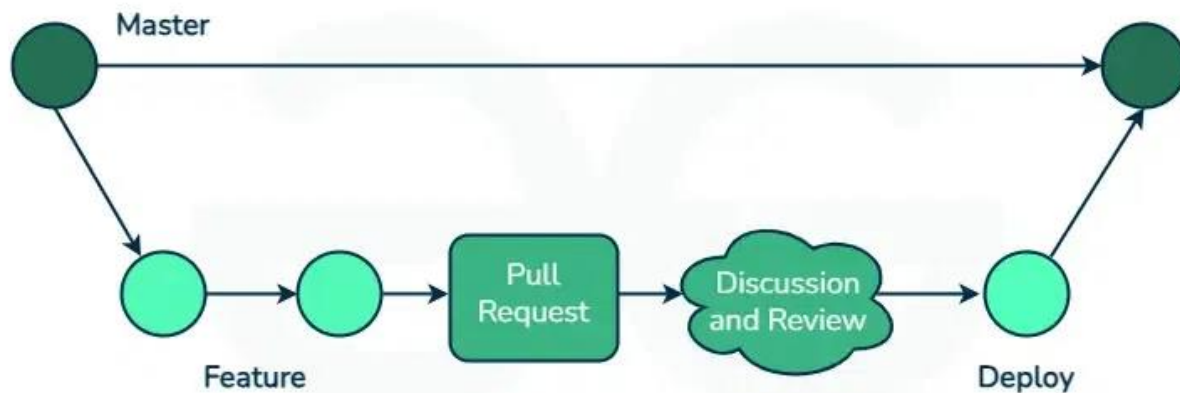
## GitHub Flow

GitHub flow is a simpler alternative to GitFlow, idea for smaller teams. [GitHub](#) flow only has feature branches that stem directly from the master branch and are merged back to master after completing changes. They don't have release branches. The fundamental concept of this model revolves around maintaining the master code in a consistently deployable condition, thereby enabling the seamless implementation of faster release cycles, continuous integration and continuous delivery workflows.

The types of branches that can be present in GitFlow are:

- **Master** - The GitHub Flow workflow initiates with the master branch, housing the most recent stable code prepared for release.

**Feature** - Developers initiate feature branches from the main branch to implement new features or address bugs. Upon completion, the feature branch is merged back into the main branch. If a [merge conflict](#) arises, developers are required to resolve it prior to finalizing the merge.



Branching strategies



## Pros Of Github Flow

- GitHub Flow emphasizes fast and streamlined branching, short production cycles, and frequent releases, aligning well with Agile methodologies.
- Teams can quickly identify and resolve issues due to the strategy's focus on fast feedback loops.
- Testing and automating changes to a single branch enable quick and continuous deployment.
- GitHub Flow is particularly well-suited for small teams and web applications, where maintaining a single production version is sufficient.

## Cons Of Github Flow

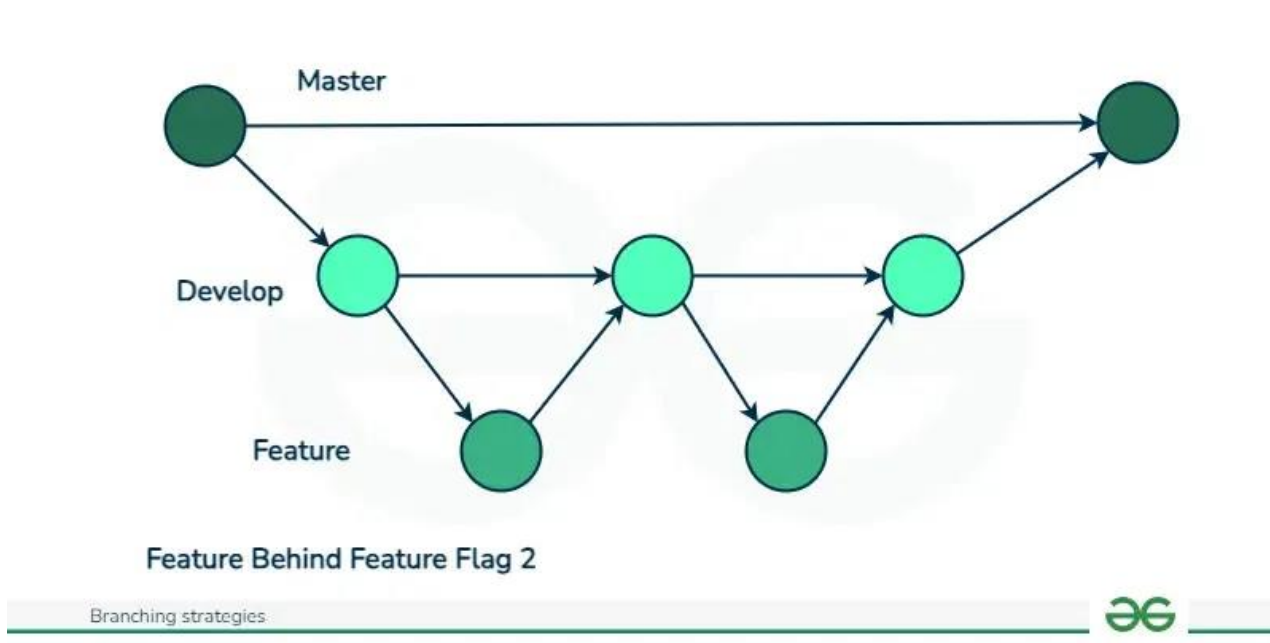
- GitHub Flow is not ideal for managing multiple versions of the codebase.
- The lack of development branches can lead to unstable production code if changes are not properly tested before merging.
- Without separate development branches, the master branch can become cluttered, serving both production and development purposes.
- As teams grow, merge conflicts may occur more frequently due to everyone merging changes to the same branch. Lack of transparency can exacerbate this issue, as developers may not see what others are working on.

## GitLab Flow

[GitLab](#) flow is also an alternative to GitFlow, designed to be more robust and scalable than [GitHub](#) Flow. Designed for teams using GitLab, a web-based Git repository manager, this approach streamlines development by concentrating on a solitary, protected branch, usually the master branch. Continuous integration and automated testing form the core elements of GitLab Flow, guaranteeing the stability of the master branch.

The types of branches that can be present in GitFlow are:

- **Master:** Main production branch housing stable release ready code.
- **Develop:** Contains new features and bug fixes.
- **Feature:** Developers initiate feature branches from the develop branch to implement new features or address bugs. Upon completion, they integrate the changes from the feature branch into the develop branch.
- **Release:** Prior to a new release, a release branch is branched off from the develop branch. This release branch serves as a staging area for integrating new features and bug fixes intended for the upcoming release. Upon completion, developers merge the changes from the release branch into both the develop and main branches.



## Pros Of Gitlab Flow

- GitLab Flow offers a robust and scalable Git branching strategy, particularly suitable for larger teams and projects.
- This approach ensures a distinct separation between code under development and production-ready code, minimizing the risk of inadvertent changes to the production code.
- With GitLab Flow, each feature is developed in its own branch, promoting independent development and reducing conflicts during integration into the main codebase.
- The use of separate branches enables developers to work concurrently on different features, leading to quicker feature development.

## Cons Of Github Flow

- GitLab Flow may pose challenges due to its complexity, particularly for teams new to Git.
- Merging feature branches into the develop branch can result in conflicts, as these branches may diverge from the develop branch over time.

- The GitLab Flow strategy may slow down development, as it necessitates merging changes into the develop branch before release. This could be problematic for teams requiring rapid release of new features and bug fixes.

### Trunk Based Development

It is a branching strategy where developers work on a single "trunk" branch, mostly the master branch and use feature flags to isolate features until they are ready for release. This main branch should be ready for release any time. No additional branches are created. The main idea behind this strategy is to make smaller changes more frequently to avoid merge conflicts and the goal is to limit long-lasting branches. This strategy enables continuous integration and delivery, making it an attractive choice for teams aiming to release updates swiftly and frequently. It is particularly well-suited for smaller projects or teams seeking a streamlined workflow.

### Pros Of Trunk Based Development

- Trunk-based development keeps the trunk consistently updated, enabling continuous integration of code changes.
- Developers have better visibility into each other's changes as commits are made directly to the trunk, promoting collaboration and transparency.
- Without the need for branches, there is less likelihood of [encountering merge conflicts](#) or "merge hell," as developers push small changes more frequently, simplifying conflict resolution.
- The shared trunk remains in a constant releasable state, allowing for faster and more stable releases due to the continuous integration of work.

### Cons Of Trunk Based Development

- Trunk-based development requires a significant amount of autonomy and may be daunting for less experienced developers who interact directly with the shared trunk, hence it is suitable for senior developers.
- Trunk-based development demands a considerable level of discipline and effective communication among developers to prevent conflicts and ensure proper isolation of new features.
- Difficult to manage for large teams.
- Maintaining backward compatibility with older releases can also pose challenges.

### Picking The Right Branching Strategy

Git offers a wide range of branching strategies, each suited to different project requirements and team dynamics. For beginners, starting with simpler approaches like GitHub Flow or [Trunk-based development](#) is recommended, gradually advancing to more complex strategies as needed. Feature flagging can also help reduce the necessity for excessive branching. GitFlow is beneficial for projects requiring strict access control, particularly in open-source environments. However, it may not align well with DevOps practices. Therefore, teams seeking an [Agile](#) DevOps workflow with strong support for [continuous integration](#) and delivery may find GitHub Flow or Trunk-based development more suitable. Ultimately, the choice of branching strategy depends on the specific needs and goals of the project and team.

| Product Type                                   | Team   | Applicable Strategy  |
|------------------------------------------------|--------|----------------------|
| Continuous Deployment and Release              | Small  | GitHub Flow and TB   |
| Scheduled and Periodic Version Release         | Medium | GitFlow and GitLab I |
| Continuous deployment for quality-focused prod | Medium | GitLab Flow          |

| Product Type                          | Team  | Applicable Strategy |
|---------------------------------------|-------|---------------------|
| Products with long maintenance cycles | Large | GitFlow             |

## Git Merging

Git is an important tool that helps developers manage changes to their codebase. One of the most critical and frequently used commands in Git is merge. Merging allows you to integrate changes from different branches into a single branch, ensuring that all updates are consolidated. In this article, we will see more about git merge command, its syntax, uses, and provide examples to help you understand how to effectively use it in your projects.

### What is Git Merge?

**git merge** is a command used to combine the changes from one or more branches into the current branch. It integrates the history of these branches, ensuring that all changes are included and conflicts are resolved.

### Syntax:

```
git merge <branch-name>
```

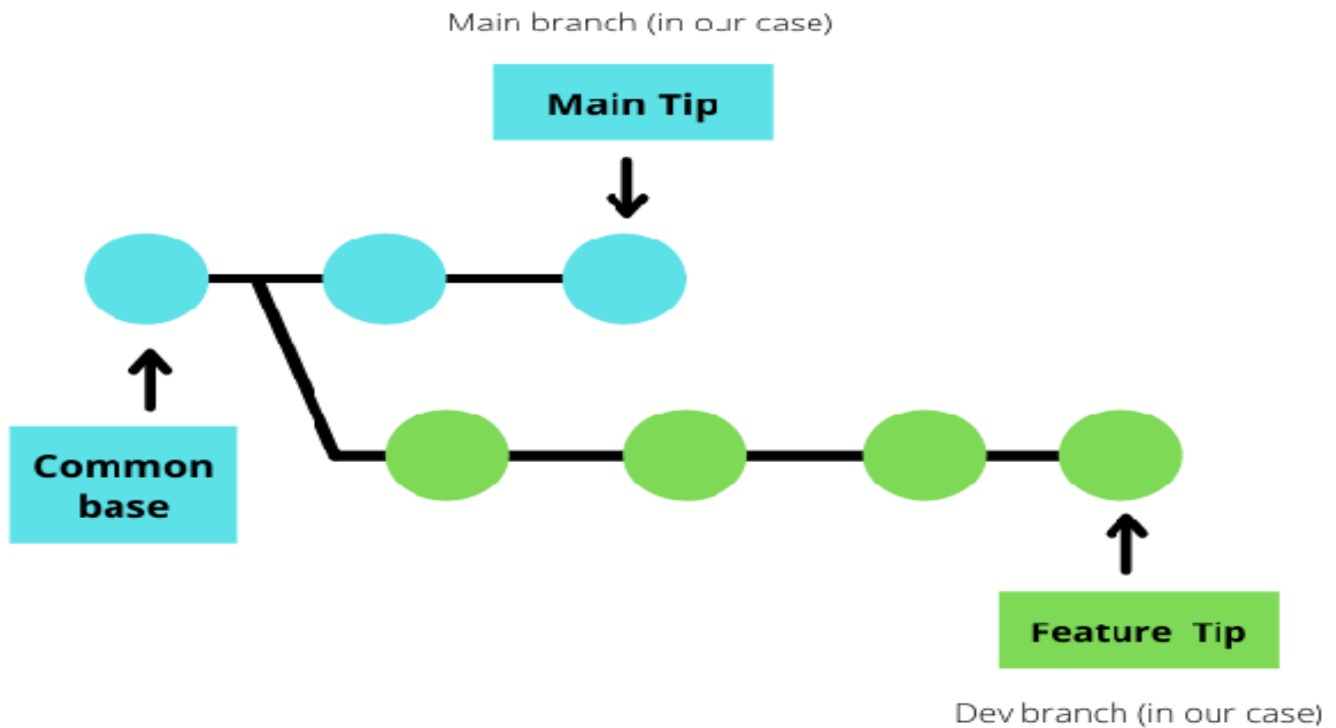
#### Uses of Git Merge

- To incorporate changes from another branch into the current branch.
- To consolidate development work done in different branches.
- To bring feature branches into the main branch (e.g., main or master).

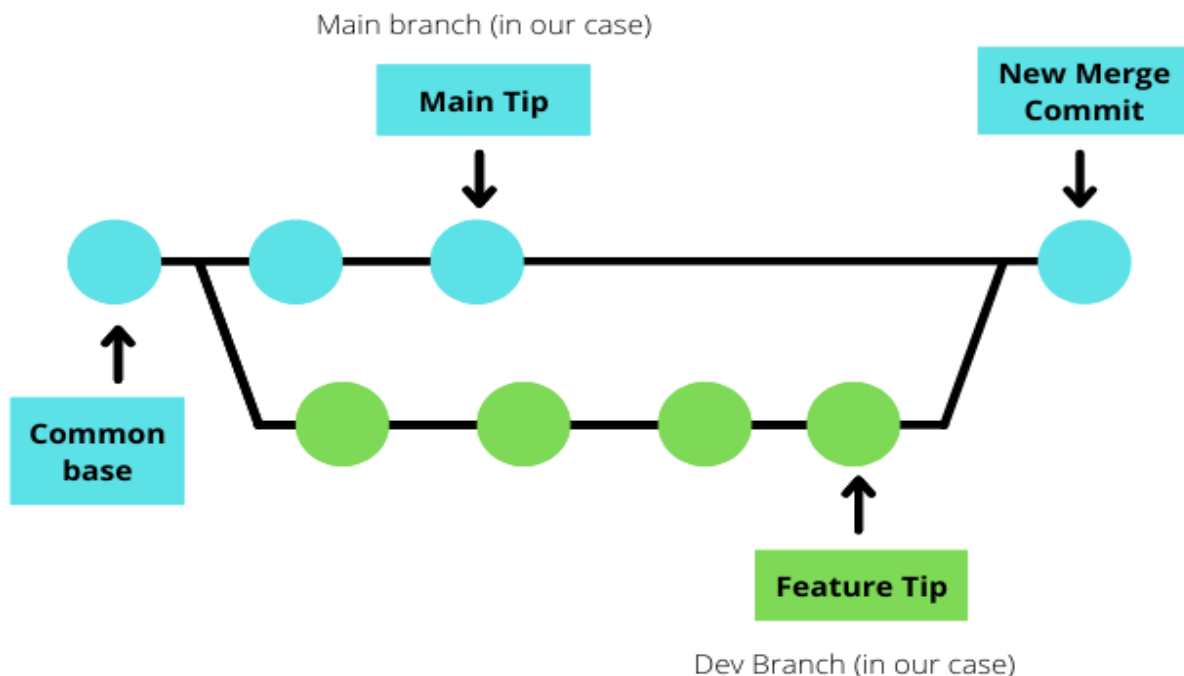
### How Does Git Merge Work?

The concept of git merging is basically to merge multiple sequences of commits, stored in multiple branches in a unified history, or to be simple you can say in a [single branch](#).

What happens is when we try to merge two branches, git takes two commit pointers and starts searching for a common base commit in those two specified bit branches. When git finds the common base commit it simply creates a “merge commit” automatically and merges each queued merge commit sequence. There is a proper merging algorithm in git, with the help of which git performs all of these operations and presents conflicts if there are any.



In our case, we have two branches one is the default branch called “main” and the other branch named “dev” and this is how our git repo looks before merging. Here git finds the common base, creates a new merge commit, and merged them.



A git merge operation is performed by running the below command. When we perform merging, git always merges with the current branch from where we are performing the operation(in our case it is “main”). By this, the branch being merged is not affected.

"git merge <name of the branch to be merged (in our case it is "dev")>".

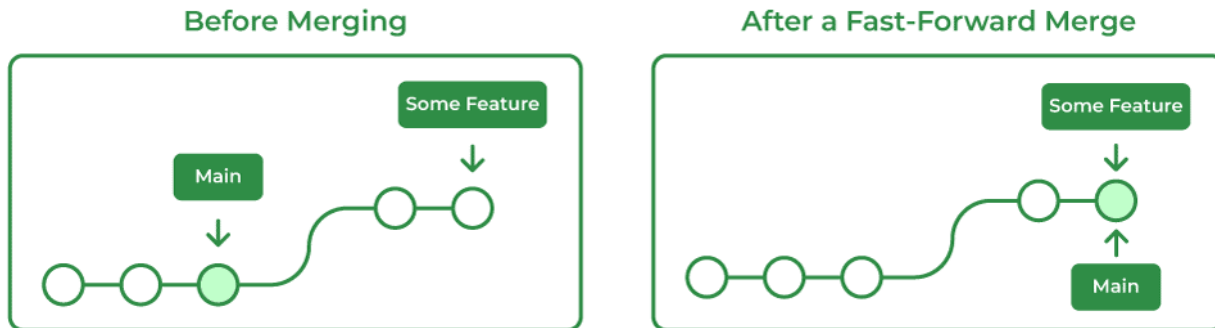
## Merging Types

In Git, there are two primary types of merging. There are.

1. Fast-forward merging.
2. Three-way merging.

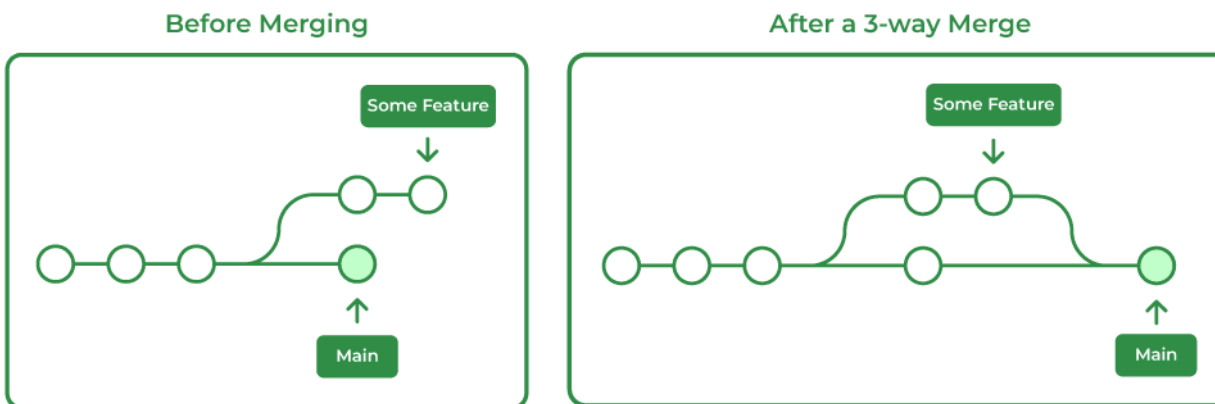
### 1. Fast-Forward Merging

Fast forward merge happens when the tip of the current branch (“dev” in our case) is a direct ancestor of the target branch (“main” in our case). Here instead of actually merging the two branches git simply moves the current branch tip up to the target branch tip. Fast-forward merge is not possible if the branches have diverged. Then we need a **3-way merge** which uses a dedicated commit to merge two histories or you can say branches.



### 2. Three-Way Merging

When the base branch has changed since the branch was first created, this kind of merging takes place. Git in this situation generates a fresh merging commit that incorporates the modifications from both branches. Git compares the modifications made to both branches with those made to the base branch using a three-way merge process. Following that, it integrates both sets of changes into a single new commit.



Git also supports some other types of merging like recursive and octopus margin. With the help of a single merge commit “**octopus merging**” can merge multiple branches at once. “**Recursive merging**” is similar to three-way merging but it can handle some more complex merge operations than the three-way merging.

### Steps To Merge a Branch

To ensure that the merging process goes smoothly we need to follow a series of steps for merging which involves resolving any conflicts that we may face. Below are the basic steps involved.



### **Step 1: Creating a new branch.**

Create a new branch from the remote repository branch which you want to merge. If errors are faced while merging we can go back to the previous version immediately.

### **Step 2: Make sure always latest changes are pulled.**

Always make sure before merging the latest changes that the latest changes are pulled from both branches like the master and the branch you want to merge.

### **Step 3: Resolving the merge conflicts.**

While merging the branches it is possible that some merge conflicts will be raised then git will prompt you to resolve the merge conflicts. If any merge conflicts are not raised then git will automatically merge the branches.

### **Step 4: Merged code needs to be tested.**

It is essential to test the merged code and we have to make sure that the code doesn't have any bugs and it is working properly. To test the code we do it automatically or manually.

### **Step 5: Commit the merged code.**

Once completing of merging the code if you are satisfied with the work, Know it's time to commit the new changes of code into the new branch.

### **Step 6: Push the merged branch.**

Lastly, make the new branch accessible to other team members by pushing it to the repository.

In conclusion, pulling the most recent changes, resolving conflicts, testing the merged code, committing the changes, and pushing the new branch are the essential phases in the Git merge preparation process. The merging procedure in Git can be streamlined and effective with careful planning and attention to detail.

### **How To Resolve Merge Conflicts?**

While merging the two branches if changes are made to two different branches then git will not merge automatically it prompts the user to resolve the [merge conflicts](#) manually. Below are the steps to resolve the merge conflicts in git:

#### **Step 1: Identify the conflict files.**

Git will automatically display a message by indicating the file to be resolved from merge conflicts. You have to resolve the conflicts manually.

#### **Step 2: Open the conflict files.**

Open the merge conflict files by using editors whatever you are convenient with like ([IDE](#)). After opening we can conflict markers as shown below it will indicate where the conflicts are located.

#### **Conflict markers**

```
(<<<<<<<, =====, and >>>>>>>)
```

#### **Step 3: Resolve the conflicts.**

Remove the unnecessary changes after examining them carefully and keep the changes that are more important.

#### **Step 4: Moving to the staging.**

Use the git add command to add the updated files to the staging area after the conflicts have been resolved.

#### **Step 5: Commit and Push the changes**

After resolving conflicts [commit](#) the changes by using the below command. Including the message which gives information about changes made while resolving the conflicts.

```
git commit -m "message"
```

[Push](#) the changes made to the [remote repository](#) by using. Below command.

```
git push
```

Where other developers can access the code. And perform any changes that are required.



-  
After resolving the conflicts, it is crucial to carefully analyze and test the merged code to make sure that the modifications are functioning as intended and that no new problems have been introduced. These procedures can help developers resolve merge disputes in Git and maintain a dependable and stable codebase.

## UNIT - II

### JavaScript and jQuery: JavaScript basics, Functions, form validation, OOPS Aspects of JavaScript, JQuery Framework, jQuery events, AJAX for data exchange with server, JSON data format.

JavaScript is a general-purpose, prototype-based, object-oriented scripting language. It is designed to be embedded in diverse applications and systems, without consuming much memory. JavaScript borrows most of its syntax from Java, but also inherits from Awk and Perl, with some indirect influence from Self in its object prototype system.

JavaScript is dynamically typed, that is, programs do not declare variable types, and the type of a variable is unrestricted and can change at runtime. Source code can be generated at runtime and evaluated against an arbitrary scope. Typical implementations compile by translating source into an unspecified bytecode format, to check syntax and source consistency. Note that the ability to generate and interpret programs at runtime implies the presence of a compiler at runtime.

JavaScript is a high-level scripting language that does not depend on or expose particular machine representations or operating system services.

It provides automatic storage management, typically using a garbage collector.

The language has the standard objects and functions .

| Edition | Date published                             | Name               | Changes from prior edition                                                                                                                                                                                                                                                                     | Editor                            |
|---------|--------------------------------------------|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------|
| 1       | June 1997                                  |                    | First edition based on JavaScript 1.1 as implemented in Netscape Navigator 3.0. <sup>[1]</sup>                                                                                                                                                                                                 | <a href="#">Guy L. Steele Jr.</a> |
| 2       | June 1998                                  |                    | Editorial changes to keep the specification fully aligned with ISO/IEC 16262:1998.                                                                                                                                                                                                             | <a href="#">Mike Cowlshaw</a>     |
| 3       | December 1999                              |                    | Based on JavaScript 1.2 as implemented in Netscape Navigator 4.0. <sup>[2]</sup> Added <a href="#">regular expressions</a> , better string handling, new control statements, try/catch exception handling, tighter definition of errors, formatting for numeric output, and other enhancements | <a href="#">Mike Cowlshaw</a>     |
| 4       | <i>Abandoned</i> (last draft 30 June 2003) | ECMAScript 4 (ES4) | Fourth Edition was abandoned, due to political differences concerning language complexity. Many features proposed for the Fourth Edition have been completely dropped; some were incorporated into the sixth edition.                                                                          |                                   |

|            |                           |                          |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |                                                            |
|------------|---------------------------|--------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------|
| <b>5</b>   | December 2009             |                          | Adds "strict mode", a subset intended to provide more thorough error checking and avoid error-prone constructs. Clarifies many ambiguities in the 3rd edition specification, and accommodates behavior of real-world implementations that differed consistently from that specification. Adds some new features, such as getters and <a href="#">setters</a> , library support for <a href="#">JSON</a> , and more complete <a href="#">reflection</a> on object properties. <sup>[3]</sup> | Pratap Lakshman,<br>Allen Wirfs-Brock                      |
| <b>5.1</b> | June 2011                 |                          | Changes to keep the specification fully aligned with ISO/IEC 16262:2011.                                                                                                                                                                                                                                                                                                                                                                                                                    | Pratap Lakshman,<br>Allen Wirfs-Brock                      |
| <b>6</b>   | June 2015 <sup>[4]</sup>  | ECMAScript 2015 (ES2015) | See <a href="#">#6th Edition – ECMAScript 2015</a>                                                                                                                                                                                                                                                                                                                                                                                                                                          | Allen Wirfs-Brock                                          |
| <b>7</b>   | June 2016 <sup>[5]</sup>  | ECMAScript 2016 (ES2016) | See <a href="#">#7th Edition – ECMAScript 2016</a>                                                                                                                                                                                                                                                                                                                                                                                                                                          | Brian Terlson                                              |
| <b>8</b>   | June 2017 <sup>[6]</sup>  | ECMAScript 2017 (ES2017) | See <a href="#">#8th Edition – ECMAScript 2017</a>                                                                                                                                                                                                                                                                                                                                                                                                                                          | Brian Terlson                                              |
| <b>9</b>   | June 2018 <sup>[7]</sup>  | ECMAScript 2018 (ES2018) | See <a href="#">#9th Edition – ECMAScript 2018</a>                                                                                                                                                                                                                                                                                                                                                                                                                                          | Brian Terlson                                              |
| <b>10</b>  | June 2019 <sup>[8]</sup>  | ECMAScript 2019 (ES2019) | See <a href="#">#10th Edition – ECMAScript 2019</a>                                                                                                                                                                                                                                                                                                                                                                                                                                         | Brian Terlson, Bradley Farias, Jordan Harband              |
| <b>11</b>  | June 2020 <sup>[9]</sup>  | ECMAScript 2020 (ES2020) | See <a href="#">#11th Edition – ECMAScript 2020</a>                                                                                                                                                                                                                                                                                                                                                                                                                                         | Jordan Harband, Kevin Smith                                |
| <b>12</b>  | June 2021 <sup>[10]</sup> | ECMAScript 2021 (ES2021) | See <a href="#">#12th Edition – ECMAScript 2021</a>                                                                                                                                                                                                                                                                                                                                                                                                                                         | Jordan Harband, Shu-yu Guo, Michael Ficarra, Kevin Gibbons |
| <b>13</b>  | June 2022 <sup>[11]</sup> | ECMAScript 2022 (ES2022) | See <a href="#">#13th Edition – ECMAScript 2022</a>                                                                                                                                                                                                                                                                                                                                                                                                                                         | Shu-yu Guo, Michael Ficarra, Kevin Gibbons                 |
| <b>14</b>  | June 2023 <sup>[12]</sup> | ECMAScript 2023 (ES2023) | See <a href="#">#14th Edition – ECMAScript 2023</a>                                                                                                                                                                                                                                                                                                                                                                                                                                         | Shu-yu Guo, Michael Ficarra, Kevin Gibbons                 |

|    |                           |                          |                                                                  |                                            |
|----|---------------------------|--------------------------|------------------------------------------------------------------|--------------------------------------------|
| 15 | June 2024 <sup>[13]</sup> | ECMAScript 2024 (ES2024) | See <a href="#">#15th Edition – ECMAScript 2024</a>              | Shu-yu Guo, Michael Ficarra, Kevin Gibbons |
| 16 | (pending)                 | ECMAScript 2025 (ES2025) | Pending, see features being considered: <a href="#">#ES.next</a> | (pending)                                  |

## JAVASCRIPT

JavaScript is the scripting language of the Web.

JavaScript is used in millions of Web pages to add functionality, validate forms, detect browsers, and much more.

### Introduction to JavaScript

JavaScript is used in millions of Web pages to improve the design, validate forms, detect browsers, create cookies, and much more.

JavaScript is the most popular scripting language on the Internet, and works in all major browsers, such as Internet Explorer, Mozilla Firefox, and Opera.

### What is JavaScript?

- JavaScript was designed to add interactivity to HTML pages
- JavaScript is a scripting language
- A scripting language is a lightweight programming language
- JavaScript is usually embedded directly into HTML pages
- JavaScript is an interpreted language (means that scripts execute without preliminary compilation)
- Everyone can use JavaScript without purchasing a license

Java and JavaScript are two completely different languages in both concept and design!

Java (developed by Sun Microsystems) is a powerful and much more complex programming language - in the same category as C and C++.

### What can a JavaScript Do ?

- **JavaScript gives HTML designers a programming tool** - HTML authors are normally not programmers, but JavaScript is a scripting language with a very simple syntax! Almost anyone can put small "snippets" of code into their HTML pages
- **JavaScript can put dynamic text into an HTML page** - A JavaScript statement like this: `document.write("<h1>" + name + "</h1>")` can write a variable text into an HTML page
- **JavaScript can react to events** - A JavaScript can be set to execute when something happens,

- like when a page has finished loading or when a user clicks on an HTML element
- **JavaScript can read and write HTML elements** - A JavaScript can read and change the content of an HTML element
- **JavaScript can be used to validate data** - A JavaScript can be used to validate form data before it is submitted to a server. This saves the server from extra processing
- **JavaScript can be used to detect the visitor's browser** - A JavaScript can be used to detect the visitor's browser, and - depending on the browser - load another page specifically designed for that browser
- **JavaScript can be used to create cookies** - A JavaScript can be used to store and retrieve information on the visitor's computer.

### JavaScript Variables

Variables are "containers" for storing information. JavaScript variables are used to hold values or expressions.

A variable can have a short name, like x, or a more descriptive name, Student\_name

#### Rules for JavaScript variable names:

- Variable names are case sensitive (y and Y are two different variables)
- Variable names must begin with a letter or the underscore character

**Note:** Because JavaScript is case-sensitive, variable names are case-sensitive.

#### Example

A variable's value can change during the execution of a script. You can refer to a variable by its name to display or change its value.

```
<html>
<body>
<script type="text/javascript">var firstname;
firstname="Welcome";
document.write(firstname);
document.write("
");
firstname="XYZ";
document.write(firstname);
</script>
<p>The script above declares a variable, assigns a value to it, displays the value, change the value,
and displays the value again.</p>
</body>
</html>
```

Output :

Welcome  
XYZ

-

The script above declares a variable, assigns a value to it, displays the value, change the value, and displays the value again.

### Declaring (Creating) JavaScript Variables

Creating variables in JavaScript is most often referred to as "declaring" variables.

You can declare JavaScript variables with the **var statement**:

```
var x;
var carname;
```

After the declaration shown above, the variables are empty (they have no values yet). However, you can also assign values to the variables when you declare them:

```
var x=5;
var carname="Scorpio";
```

After the execution of the statements above, the variable **x** will hold the value **5**, and **carname** will hold the value **Scorpio**.

**Note:** When you assign a text value to a variable, use quotes around the value.

### Assigning Values to Undeclared JavaScript Variables

If you assign values to variables that have not yet been declared, the variables will automatically be declared.

These statements:

```
x=5;
carname="Scorpio";
```

have the same effect as:

```
var x=5;
var carname="Scorpio";
```

### Redeclaring JavaScript Variables

If you redeclare a JavaScript variable, it will not lose its original value.

```
var x=5;var
x;
```

-  
After the execution of the statements above, the variable x will still have the value of 5. The value of x is not reset (or cleared) when you redeclare it.

## DataTypes

- **Numbers** - are values that can be processed and calculated. You don't enclose them in quotation marks. The numbers can be either positive or negative.
- **Strings** - are a series of letters and numbers enclosed in quotation marks. JavaScript uses the string literally; it doesn't process it. You'll use strings for text you want displayed or values you want passed along.
- **Boolean (true/false)** - lets you evaluate whether a condition meets or does not meet specified criteria.
- **Null** - is an empty value. **null** is not the same as 0 -- 0 is a real, calculable number, whereas **null** is the **absence of any value**.

### Data Types

TYPE	EXAMPLE
Numbers	Any number, such as 17, 21, or 54e7
Strings	"Greetings!" or "Fun"
Boolean	Either true or false
Null	A special keyword for exactly that – the null value (that is, nothing)

### JavaScript Arithmetic

As with algebra, you can do arithmetic operations with JavaScript variables:

```
y=x-5;
z=y+5
```

### JavaScript Operators

The operator = is used to assign values. The operator + is used to add values.

The assignment operator = is used to assign values to JavaScript variables.

The arithmetic operator + is used to add values together.

```
y=5;
z=2;
x=y+z;
```

-  
The value of x, after the execution of the statements above is 7.

## **JavaScript Arithmetic Operators**

Arithmetic operators are used to perform arithmetic between variables and/or values.

Given that **y=5**, the table below explains the arithmetic operators:

Operator	Description	Example	Result
+	Addition	$x=y+2$	$x=7$
-	Subtraction	$x=y-2$	$x=3$
*	Multiplication	$x=y*2$	$x=10$
/	Division	$x=y/2$	$x=2.5$
%	Modulus (division remainder)	$x=y\%2$	$x=1$
++	Increment	$x=++y$	$x=6$
--	Decrement	$x=--y$	$x=4$

## **JavaScript Assignment Operators**

Assignment operators are used to assign values to JavaScript variables.

Given that **x=10** and **y=5**, the table below explains the assignment operators:

Operator	Example	Same As	Result
=	$x=y$		$x=5$
+=	$x+=y$	$x=x+y$	$x=15$
-=	$x-=y$	$x=x-y$	$x=5$
*=	$x*=y$	$x=x*y$	$x=50$
/=	$x/=y$	$x=x/y$	$x=2$
%=	$x\%=y$	$x=x\%y$	$x=0$

### **The + Operator Used on Strings**

The + operator can also be used to add string variables or text values together.

To add two or more string variables together, use the + operator.



```
txt1="What a very";
txt2="nice day";
txt3=txt1+txt2;
```

After the execution of the statements above, the variable txt3 contains "What a very nice day". To add a space between the two strings, insert a space into one of the strings:

```
txt1="What a very ";
txt2="nice day";
txt3=txt1+txt2;
```

or insert a space into the expression:

```
txt1="What a very";
txt2="nice day";
txt3=txt1+" "+txt2;
```

After the execution of the statements above, the variable txt3 contains: "What a very nice day"

### Adding Strings and Numbers

Look at these examples:

```
x=5+5;
document.write(x);

x="5"+"5";
document.write(x);

x=5+"5";
document.write(x);

x="5"+5;
document.write(x);
```

The rule is:

**If you add a number and a string, the result will be a string.**

**JavaScript Comparison and Logical Operators** Comparison and Logical operators are used to test for true or false.

### Comparison Operators

Comparison operators are used in logical statements to determine equality or difference between variables or

-  
values.

Given that **x=5**, the table below explains the comparison operators:

Operator	Description	Example
==	is equal to	x==8 is false
===	is exactly equal to (value and type)	x===5 is true x==="5" is false
!=	is not equal	x!=8 is true
>	is greater than	x>8 is false
<	is less than	x<8 is true
>=	is greater than or equal to	x>=8 is false
<=	is less than or equal to	x<=8 is true

### How Can it be Used

Comparison operators can be used in conditional statements to compare values and take action depending on the result:

```
if (age<18) document.write("Too young");
```

You will learn more about the use of conditional statements in the next chapter of this tutorial.

### Logical Operators

Logical operators are used to determine the logic between variables or values. Given that **x=6 and y=3**, the table below explains the logical operators:

Operator	Description	Example
&&	and	(x < 10 && y > 1) is true
	or	(x==5    y==5) is false
!	not	!(x==y) is true

### Conditional Operator

JavaScript also contains a conditional operator that assigns a value to a variable based on some condition.

Syntax

```
variablename=(condition)?value1:value2
```

### Example

```
greeting=(visitor=="PRES")?"Dear President ":"Dear ";
```

-

If the variable **visitor** has the value of "PRES", then the variable **greeting** will be assigned the value "Dear President " else it will be assigned "Dear".

## Conditional Statements

Very often when you write code, you want to perform different actions for different decisions. You can use conditional statements in your code to do this.

In JavaScript we have the following conditional statements:

- **if statement** - use this statement if you want to execute some code only if a specified condition is true
- **if...else statement** - use this statement if you want to execute some code if the condition is true and another code if the condition is false
- **if...else if... else statement** - use this statement if you want to select one of many blocks of code to be executed
- **switch statement** - use this statement if you want to select one of many blocks of code to be executed

### If Statement

You should use the if statement if you want to execute some code only if a specified condition is true.

Syntax

```
if (condition)
{
 code to be executed if condition is true
}
```

Note that if is written in lowercase letters. Using uppercase letters (IF) will generate a JavaScript error!

Example 1

```
<script type="text/javascript">
//Write a "Good morning" greeting if
//the time is less than
10var d=new Date();
var time=d.getHours();
```

```
if (time<10)
{
document.write("Good morning");
}
</script>
```

### Example 2

```
<script type="text/javascript">
//Write "Lunch-time!" if the time is 11var
d=new Date();
var time=d.getHours();

if (time==11)
{
document.write("Lunch-time!");
}
</script>
```

**Note:** When **comparing** variables you must always use two equals signs next to each other (==)!

Notice that there is no **..else..** in this syntax. You just tell the code to execute some code **only if the specified condition is true**.

### If...else Statement

If you want to execute some code if a condition is true and another code if the condition is not true, use the if ...else statement.

Syntax

```
if (condition)
{
code to be executed if condition is true
}
else
{
code to be executed if condition is not true
}
```

## Example

```
<script type="text/javascript">
//If the time is less than 10,
//you will get a "Good morning" greeting.
//Otherwise you will get a "Good day" greeting.var d = new Date();
```

```
var time = d.getHours();

if (time < 10)
{
document.write("Good morning!");
}
else
{
document.write("Good day!");
}
</script>
```

## If...else if...else Statement

You should use the if....else if...else statement if you want to select one of many sets of lines to execute.

### Syntax

```
if (condition 1)
{
code to be executed if condition 1 is true
}
else if (condition 2)
{
code to be executed if condition 2 is true
}
else
{
code to be executed if condition 1 and
condition 2 are not true
}
```

## Example

```
<script type="text/javascript">
var d = new Date()
var time =
d.getHours()if
(time<10)
{
document.write("Good morning");
}
else if (time>10 && time<16)
{
document.write("Good day");
}
{
document.write("Hello World!");
}
</script>
```

### The JavaScript Switch Statement

```
<script type="text/javascript">
//You will receive a different greeting based
//on what day it is. Note that Sunday=0,
//Monday=1, Tuesday=2, etc.
var d=new Date();
theDay=d.getDay();
switch (theDay)
{
case 5:
 document.write("Finally Friday");
 break;
case 6:
 document.write("Super Saturday");
 break;
case 0:
 document.write("Sleepy Sunday");
```

You should use the switch statement if you want to select one of many blocks of code to be executed.

-

## Syntax

```
switch(n)
{
case 1:
 execute code block 1
 break;
case 2:
 execute code block 2
 break;
default:
 code to be executed if n is
 different from case 1 and 2
}
```

This is how it works: First we have a single expression  $n$  (most often a variable), that is evaluated once. The value of the expression is then compared with the values for each case in the structure. If there is a match, the block of code associated with that case is executed. case **break** to prevent the code from running into the next case automatically.

### Example

```
break;
default:
 document.write("I'm looking forward to this weekend!");
}
</script>
```

## JavaScript Controlling(Looping) Statements

Loops in JavaScript are used to execute the same block of code a specified number of times or while a specified condition is true.

### JavaScript Loops

Very often when you write code, you want the same block of code to run over and over again in a row. Instead of adding several almost equal lines in a script we can use loops to perform a task like this.

In JavaScript there are two different kind of loops:

- **for** - loops through a block of code a specified number of times
- **while** - loops through a block of code while a specified condition is true



## The for Loop

The for loop is used when you know in advance how many times the script should run.

Syntax

```
for (var=startvalue;var<=endvalue;var=var+increment)
{
 code to be executed
}
```

### Example

Explanation: The example below defines a loop that starts with `i=0`. The loop will continue to run as long as `i` is less than, or equal to 10. `i` will increase by 1 each time the loop runs.

**Note:** The increment parameter could also be negative, and the `<=` could be any comparing statement.

```
<html>
<body>
<script type="text/javascript">
var i=0;
for (i=0;i<=10;i++)
{
```

```
document.write("The number is " +
i);document.write("
");
}
</script>
</body>
</html>
```

Result

```
The number is 0
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
The number is 6
The number is 7
The number is 8
The number is 9
The number is
10
```

## **JavaScript While Loop**

Loops in JavaScript are used to execute the same block of code a specified number of times or while a specified condition is true.

The while loop

The while loop is used when you want the loop to execute and continue executing while the specified condition is true.

```
while (var<=endvalue)
{
 code to be executed
}
```

**Note:** The <= could be any comparing statement.

Example

Explanation: The example below defines a loop that starts with i=0. The loop will continue to run as long as i is less than, or equal to 10. i will increase by 1 each time the loop runs.

```
<html>
```

```

<body>
<script type="text/javascript">
var i=0;
while (i<=10)
{
document.write("The number is " + i);
document.write("
");
i=i+1;
}
</script>
</body>
</html>

```

Result

```

The number is 0
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
The number is 6
The number is 7
The number is 8
The number is 9
The number is 10

```

### **The do...while Loop**

The do...while loop is a variant of the while loop. This loop will always execute a block of code ONCE, and then it will repeat the loop as long as the specified condition is true. This loop will always be executed at least once, even if the condition is false, because the code is executed before the condition is tested.

```

do
{
 code to be executed
}
while (var<=endvalue);

```

Example

```

<html>
<body>
<script type="text/javascript">

```

```
var
i=0;do
{
document.write("The number is " + i);
document.write("
");
i=i+1;
}
while (i<0);
</script>
</body>
</html>
```

## Result

The number is 0

### JavaScript Break and Continue

There are two special statements that can be used inside loops: break and continue.

JavaScript break and continue Statements

There are two special statements that can be used inside loops: break and continue.

### **Break**

The break command will break the loop and continue executing the code that follows after the loop (if any).

Example

```
<html>
<body>
<script type="text/javascript">
var i=0;
for (i=0;i<=10;i++)
{
if (i==3)
{
break;
}
document.write("The number is " + i);
document.write("
");
}
</script>
</body>
```

```
</html>
```

## Result

```
The number is 0
The number is 1
The number is 2
```

## Continue

The continue command will break the current loop and continue with the next value.

```
<html>
<body>
<script type="text/javascript">var
i=0
for (i=0;i<=10;i++)
{
if (i==3)
{
continue;
}
document.write("The number is " + i);
document.write("
");
}
</script>
</body>
</html>
```

## Result

```
The number is 0
The number is 1
The number is 2
The number is 4
The number is 5
The number is 6
The number is 7
The number is 8
The number is 9
The number is
10
```

## JavaScript Functions

A function (also known as a *method*) is a self-contained piece of code that performs a particular "function". You can recognise a function by its format - it's a piece of descriptive text, followed by open and close brackets. A function is a reusable code-block that will be executed by an event, or when the function is called. To keep the browser from executing a script when the page loads, you can put your script into a function. A function contains code that will be executed by an event or by a call to that function.

You may call a function from anywhere within the page (or even from other pages if the function is embedded in an external .js file).

Functions can be defined both in the <head> and in the <body> section of a document. However, to

assure that the function is read/loaded by the browser before it is called, it could be wise to put it in the <head> section.

Example

```
<html>
<head>
<script type="text/javascript">
function displaymessage()
{
alert("Hello World!");
}
</script>
</head>
<body>
<form>
<input type="button" value="Click me!"
onclick="displaymessage()" >
</form>
</body>
</html>
```

If the line: `alert("Hello world!!")` in the example above had not been put within a function, it would have been executed as soon as the line was loaded. Now, the script is not executed before the user hits the button. We have added an `onClick` event to the button that will execute the function `displaymessage()` when the button is clicked.

You will learn more about JavaScript events in the JS Events chapter.

## How to Define a Function

The syntax for creating a function is:

```
function functionname(var1,var2,...,varX)
{
some code
}
```

`var1, var2, etc` are variables or values passed into the function. The `{` and the `}` defines the start and end of the function.

**Note:** A function with no parameters must include the parentheses () after the function name:

```
function functionname()
{
 some code
}
```

**Note:** Do not forget about the importance of capitals in JavaScript! The word function must be written in lowercase letters, otherwise a JavaScript error occurs! Also note that you must call a function with the exact same capitals as in the function name.

### The return Statement

The return statement is used to specify the value that is returned from the function. So, functions that are going to return a value must use the return statement.

#### Example

The function below should return the product of two numbers (a and b):

```
function prod(a,b)
{
 x=a*b;
 return x;
}
```

When you call the function above, you must pass along two parameters:

```
product=prod(2,3);
```

The returned value from the prod() function is 6, and it will be stored in the variable called product.

## The Lifetime of JavaScript Variables

When you declare a variable within a function, the variable can only be accessed within that function. When you exit the function, the variable is destroyed. These variables are called local variables. You can have local variables with the same name in different functions, because each is recognized only by the function in which it is declared.

If you declare a variable outside a function, all the functions on your page can access it. The lifetime of these variables starts when they are declared, and ends when the page is closed.

## What is an Event?

### Event Handlers

**Event Handlers** are JavaScript methods, i.e. functions of objects, that allow us as JavaScript programmers to control what happens when events occur.



Directly or indirectly, an **Event** is always the result of something a user does. For example, we've already seen Event Handlers like **onClick** and **onMouseOver** that respond to mouse actions. Another type of Event, an internal change-of-state to the page (**completion** of loading or **leaving** the page). An **onLoad** Event can be considered an indirect result of a user action.

Although we often refer to Events and Event Handlers interchangeably, it's important to keep in mind the distinction between them. An **Event** is merely something that happens - something that is initiated by an **Event Handler** (**onClick**, **onMouseOver**, etc...).

The elements on a page which can trigger events are known as "**targets**" or "**target elements**," and we can easily understand how a button which triggers a *Click* event is a target element for this event. Typically, events are defined through the use of Event Handlers, which are bits of script that tell the browser what to do when a particular event occurs at a particular target. These Event Handlers are commonly written as attributes of the target element's HTML tag.

The Event Handler for a *Click* event at a form field button element is quite simple to understand:

```
<INPUT TYPE="button" NAME="click1" VALUE="Click me for fun!"
onClick="event_handler_code">
```

The **event\_handler\_code** portion of this example is any valid JavaScript and it will be executed when the specified event is triggered at this target element. This particular topic will be continued in [Incorporating JavaScripts into your HTML pages](#).

There are "three different ways" that Event Handlers can be used to trigger Events or Functions.

#### Method 1 (Link Events):

Places an Event Handler as an attribute within an **<A HREF= >** tag, like this:

```
 ...
```

You can use an Event Handler located within an **<A HREF= >** tag to make either an image or a text link respond to a mouseover Event. Just enclose the image or text string between the **<A HREF= >** and the **</A>** tags.

Whenever a user clicks on a link, or moves her cursor over one, JavaScript is sent a **Link Event**. One Link Event is called **onClick**, and it gets sent whenever someone clicks on a link. Another link event is called **onMouseOver**. This one gets sent when someone moves the cursor over the link.

You can use these events to affect what the user sees on a page. Here's an example of [how to use link events](#). Try it out, View Source, and we'll go over it.

```
<A HREF="javascript:void(""
onClick="open('index.htm', 'links', 'height=200,width=200');">How to Use Link Events

```

The first interesting thing is that there are no **<SCRIPT>** tags. That's because anything that appears in the quotes of an **onClick** or an **onMouseOver** is automatically interpreted as JavaScript. In fact, because semicolons mark the end of statements allowing you to write entire JavaScripts in one line, you can fit an entire JavaScript program between the quotes of an **onClick**. It'd be ugly, but you could do it.

Here are the three lines of interest:

1. `<A HREF="#" onClick="alert('Ooo, do it again!');">Click on me!</A>`
2. `<A HREF="javascript:void()" onClick="alert('Ooo, do it again!');">Click on me!</A>`
3. `<A HREF="javascript:alert('Ooo, do it again!')">Click on me!</A>`

In the first example we have a normal `<A>` tag, but it has the magic `onClick=""` element, which says, "When someone clicks on this link, run the little bit of JavaScript between my quotes." Notice, there's even a terminating semicolon at the end of the alert. **Question:** is this required? NO.

Let's go over each line:

1. `HREF="#"` tells the browser to look for the anchor `#`, but there is no anchor `#`, so the browser reloads the page and goes to top of the page since it couldn't find the anchor.
2. `<A HREF="javascript:void()"` tells the browser not to go anywhere - it "deadens" the link when you click on it. `HREF="javascript:"` is the way to call a function when a link (hyperlink or an HREFed image) is clicked.
3. `HREF="javascript:alert('Ooo, do it again!')"` here we kill two birds with one stone. The default behavior of a hyperlink is to click on it. By clicking on the link we call the window Method `alert()` and also at the same time "deaden" the link.

The next line is

```
<A HREF="javascript:void()" onMouseOver="alert('Hee
hee!');">Mouse over me!

```

This is just like the first line, but it uses an `onMouseOver` instead of an `onClick`.

## Method 2 (Actions within FORMs):

The second technique we've seen for triggering a Function in response to a mouse action is to place an `onClick` Event Handler inside a button type form element, like this:

```
<FORM>
 <INPUT TYPE="button" onClick="doSomething()">
</FORM>
```

While any JavaScript statement, methods, or functions can appear inside the quotation marks of an Event Handler, typically, the JavaScript script that makes up the Event Handler is actually a call to a function defined in the header of the document or a single JavaScript command. Essentially, though, anything that appears inside a command block (inside curly braces `{}`) can appear between the quotation marks.

For instance, if you have a form with a text field and want to call the function `checkField()` whenever the value of the text field changes, you can define your text field as follows:

```
<INPUT TYPE="text" onChange="checkField(this)">
```

Nonetheless, the entire code for the function could appear in quotation marks rather than a function call:

```
<INPUT TYPE="text" onChange="if(this.value <= 5) {
 alert('Please enter a number greater than 5');
}">
```

To separate multiple commands in an Event Handler, use semicolons

```
<INPUT TYPE="text" onChange="alert('Thanks for the entry.');"
confirm('Do you want to continue?');">
```

The advantage of using functions as Event Handlers, however, is that you can use the same Event Handler code for multiple items in your document and, functions make your code easier to read and understand.

Method 3 (BODY **onLoad** & **onUnload**):

The third technique is to use an Event Handler to ensure that all required objects are defined involve the **onLoad** and **onUnload**. These Event Handlers are defined in the **<BODY>** or **<FRAMESET>** tag of an HTML file and are invoked when the document or frameset are fully loaded or unloaded. If you set a flag within the **onLoad** Event Handler, other Event Handlers can test this flag to see if they can safely run, with the knowledge that the document is fully loaded and all objects are defined.

For example:

```
<SCRIPT>

var loaded = false;function doit() {
 // alert("Everything is \"loaded\" and loaded = " + loaded);
 alert("Everything is \"loaded\" and loaded = ' + loaded);
}

</SCRIPT>

<BODY onLoad="loaded = true;">
-- OR --
<BODY onLoad="window.loaded = true;">

<FORM>
 <INPUT TYPE="button" VALUE="Press Me"
 onClick="if (loaded == true) doit();">
-- OR --
 <INPUT TYPE="button" VALUE="Press Me"
 onClick="if (window.loaded == true) doit();">
-- OR --
 <INPUT TYPE="button" VALUE="Press Me"
 onClick="if (loaded) doit();">
</FORM>

</BODY>
```

The **onLoad** Event Handler is executed when the document or frameset is fully loaded, which means that all images have been downloaded and displayed, all subframes have loaded, any Java Applets and Plugins (Navigator) have started running, and so on. The **onUnload** Event Handler is executed just before the page is unloaded, which occurs when the browser is about to move on to a new page. Be aware that when you are working with multiple frames, there is no guarantee of the order in which the **onLoad** Event Handler is invoked for the various frames, except that the Event Handlers for the parent frame is invoked after the Event Handlers of all its children frames -- This will be discussed in detail in **Week 8**.

Setting the bgColor Property

The first example allows the user to change the color by clicking buttons, while the second example

allows you to change colors by using drop down boxes.

### Event Handlers

EVENT	DESCRIPTION
<b>onAbort</b>	the user cancels loading of an image
<b>onBlur</b>	input focus is removed from a form element (when the user clicks outside the field) or focus is removed from a window
<b>onClick</b>	the user clicks on a link or form element
<b>onChange</b>	the value of a form field is changed by the user
<b>onError</b>	an error happens during loading of a document or image
<b>onFocus</b>	input focus is given to a form element or a window
<b>onLoad</b>	once a page is loaded, NOT while loading
<b>onMouseOut</b>	the user moves the pointer off of a link or clickable area of an image map
<b>onMouseOver</b>	the user moves the pointer over a hypertext link
<b>onReset</b>	the user clears a form using the Reset button
<b>onSelect</b>	the user selects a form element's field
<b>onSubmit</b>	a form is submitted (ie, when the users clicks on a submit button)
<b>onUnload</b>	the user leaves a page

**Note:** *Input focus* refers to the act of clicking on or in a form element or field. This can be done by clicking in a text field or by tabbing between text fields.

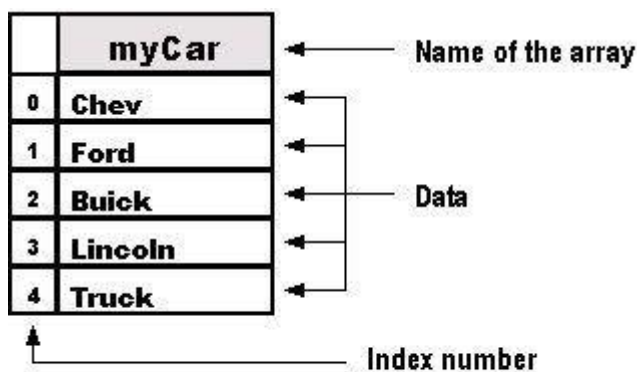
### Which Event Handlers Can Be Used

OBJECT	EVENT HANDLERS AVAILABLE
Button element	<b>onClick, onMouseOver</b>
Checkbox	<b>onClick</b>
Clickable ImageMap area	<b>onClick, onMouseOver, onMouseOut</b>
Document	<b>onLoad, onUnload, onError</b>
Form	<b>onSubmit, onReset</b>
Framesets	<b>onBlur, onFocus</b>
Hypertext link	<b>onClick, onMouseOver, onMouseOut</b>
Image	<b>onLoad, onError, onAbort</b>

Radio button	<b>onClick</b>
Reset button	<b>onClick</b>
Selection list	<b>onBlur, onChange, onFocus</b>
Submit button	<b>onClick</b>
TextArea element	<b>onBlur, onChange, onFocus, onSelect</b>
Text element	<b>onBlur, onChange, onFocus, onSelect</b>
Window	<b>onLoad, onUnload, onBlur, onFocus</b>

## JavaScript Arrays

An array object is used to create a database-like structure within a script. Grouping data points (*array elements*) together makes it easier to access and use the data in a script. There are methods of accessing actual databases (which are beyond the scope of this series) but here we're talking about small amounts of data.



An array can be viewed like a column of data in a spreadsheet. The name of the array would be the same as the name of the column. Each piece of data (*element*) in the array is referred to by a number (*index*), just like a row number in a column.

### Comparison of an array to a column of data

An array is an *object*. Earlier, I said that an object is a thing, a collection of properties (*array elements*, in this case) grouped together. You can name an array using the same format as a variable, a function or an object. Remember our basic rules: The first character cannot be a number, you cannot use a reserved word, and you cannot use spaces. Also, be sure to remember that the name of the array object is capitalized, e.g. Array.

The JavaScript interpreter uses numbers to access the collection of elements (i.e. the data) in an array. Each index number (as it is the number of the data in the array's index) refers to a specific piece of data in the array, similar to an ID number. It's important to remember that the index numbering of the data starts at "0." So, if you have 8 elements, the first element will be numbered "0" and the last one will be "7."

Elements can be of any type: character string, integer, Boolean, or even another array. An array can even have different types of elements within the same array. Each element in the array is accessed by placing its index number in brackets, i.e. `myCar[4]`. This would mean that we are looking for data located in the array `myCar` which has an index of "4." Since the numbering of an index starts at "0," this would actually be the fifth index. For instance,

in the following array,

```
var myCar = new array("Chev","Ford","Buick","Lincoln","Truck");alert(myCar[4])
```

the data point with an index of "4" would be Truck. In this example, the indexes are numbered as follows: 0=Chev, 1=Ford, 2=Buick, 3=Lincoln, and 4=Truck. When creating loops, it's much easier to refer to a number than to the actual data itself.

### The Size of the Array

The size of an array is determined by either the actual number of elements it contains or by actually specifying a given size. You don't need to specify the size of the array. Sometimes, though, you may want to pre-set the size, e.g.:

```
var myCar = new Array(20);
```

That would pre-size the array with 20 elements. You might pre-size the array in order to set aside the space in memory.

### Multidimensional Arrays

This type of an array is similar to [parallel arrays](#). In a multidimensional array, instead of creating two or more arrays in tandem as we did with the parallel array, we create an array with several levels or "dimensions." Remember [our example](#) of a spreadsheet with rows and columns? This time, however, we have a couple more columns.

	maker	model	color	
0	Gibson	Les Paul	Sunburst	← Name of the parameters
1	Fender	Stratocaster	Black	← Data
2	Martin	D-28	Mahogany	←
3	Takamine	EG330SC	Spruce	←

↑ Index number (objects created as arrays)

#### Comparison of a multidimensional array to a column of data

Multidimensional arrays can be created in different ways. Let's look at one of these methods. First, we create the main array, which is similar to what we did with previous arrays.

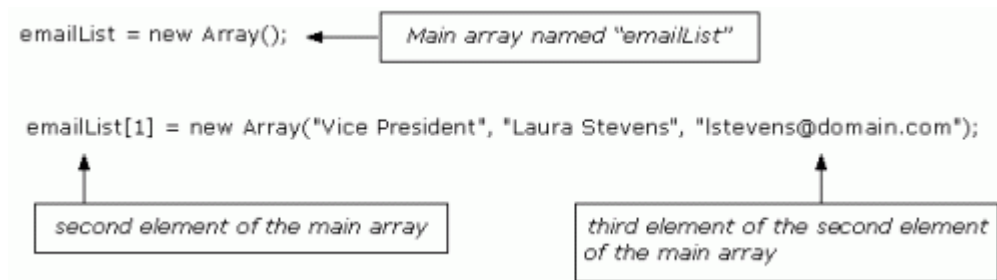
```
var emailList = new Array();
```

Next, we create arrays for elements of the main array:

```
emailList[0] = new Array("President", "Paul Smith", psmith@domain.com");
emailList[1] = new Array("Vice President", "Laura Stevens", lstevens@domain.com");
emailList[2] = new Array("General Manager", "Mary Larsen", mlarsen@domain.com");
emailList[3] = new Array("Sales Manager", "Bob Lark", blark@domain.com");
```

In this script we created "sub arrays" or arrays from another level or "dimension." We used the name of the main array and gave it an index number (e.g., emailList[0]). Then we created a new instance of an array and gave it a value with three elements.

- In order to access a single element, we need to use a double reference. For example, to get the e-mail address for the Vice President in our example above, access the third element "[2]" of the second element "[1]" of the array named emailList.



It would be written like this:

```
var vpEmail = emailList[1][2] alert("The address is: "+ vpEmail)
```

1. We declared a variable, named it emailList, and initialized it with a value of a new instance of an array.
2. Next, we created an array for each of the elements within the original array. Each of the new arrays contained three elements.
3. Then we declared a variable named vpEmail and initialized it with the value of the third element (lstevens@domain.com) of the second element "[1]" of the array named emailList.

You could also retrieve the information using something like:

```
var title = emailList[1][0]
var email = emailList[1][2]
alert("The e-mail address for the " + title + " is: " + email)
```

## Array Properties

### length

The length property returns the number of elements in an array. The format is arrayName.length. The length property is particularly useful when using a loop to cycle through an array. One example would be an array used to cycle banners:

```
var bannerImg = new Array();
bannerImg[0]="image-1.gif";
bannerImg[1]="image-2.gif";
bannerImg[2]="image-3.gif";
var newBanner = 0
var totalBan = bannerImg.length
```

```
function cycleBan() {
newBanner++
if(newBanner == totalBan) {
newBanner = 0
}
document.banner.src=bannerImg[newBanner]
setTimeout("cycleBan()", 3*1000)
}
window.onload=cycleBan;
```



This portion is then placed in the body where the banner is to be displayed:

```

```

Let's take a look and see what happened here:

1. On the first line, we created a new instance of the array `bannerImg`, and gave it three data elements. (Remember, we are only making a copy of the Array object here.)
2. Next, we created two variables: `newBanner`, which has a beginning value of zero; and `totalBan`, which returns the length of the array (the total number of elements contained in the array).
3. Then we created a function named `cycleBan`. This function will be used to create a loop to cycle the images.
  - a. We set the `newBanner` variable to be increased each time the function cycles. (Review: By placing the increment operator `[" ++ "]` after the variable [the "operand"], the variable is incremented only after it returns its current value to the script. For example, its beginning value is "0", so in the first cycle it will return a value of "0" to the script and then its value will be increased by "1".)
  - b. When the value of the `newBanner` variable is equal to the variable `totalBan` (which is the length of the array), it is then reset to "0". This allows the images to start the cycle again, from the beginning.
  - c. The next statement uses the Document Object Method (DOM - we'll be taking a look at that soon) to display the images on the Web page. Remember, we use the dot operator to access the properties of an object. We also read the statement backwards, i.e., "take the element from the array `bannerImg`, that is specified by the current value of the variable `newBanner`, and place it in the `src` attribute located in the element with the name attribute of `banner`, which is located in the document object."
  - d. We then used the `setTimeout` function to tell the script how long to display each image. This is always measured in milliseconds so, in this case, the function `cycleBan` is called every 3,000 milliseconds (i.e., every 3 seconds).
4. Finally, we used the `window.onload` statement to execute the function `cycleBan` as soon as the document is loaded.

There are a total of five properties for the Array object. In addition to the length property listed above, the others are:

1. *constructor*: Specifies the function that creates an object's prototype.
2. *index*: Only applies to JavaScript arrays created by a regular expression match.
3. *input*: Only applies to JavaScript arrays created by a regular expression match.
4. *prototype*: Used to add properties or methods.

The other properties listed here are either more advanced or seldom used. For now, we'll stick to the basics.

## Javascript Object Hierarchy

Hierarchy Objects			
Object	Properties	Methods	Event Handlers



Window	defaultStatus frames opener parent scroll self status top window	alert blur close confirm focus open prompt clearTimeout setTimeout	onLoad onUnload onBlur onFocus
History	length forward go	back	none
Navigator	appName appVersion mimeTypes plugins userAgent	javaEnabled	none
document	alinkColor anchors applets area bgColor cookie fgColor forms images lastModified linkColor links location referrer title	clear close open write writeln	none (the onLoad and onUnload event handlers belong to the Window object.)
	vlinkColor		
image	border complete height hspace lowsrc name src vspace width	none	none
form	action elements encoding FileUpload method name target	submit reset	onSubmit onReset

text	defaultValue name type value	focus blur select	onBlur onCharge onFocus onSelect
Built-in Objects			
Array	length	join reverse sort xx	none
Date	none	getDate getDay getHours getMinutes getMonth getSeconds getTime getTimeZoneoffset getYear parse prototype setDate setHours setMinutes setMonth setSeconds setTime	none
		setYear toGMTString toLocaleString UTC	
String	length prototype	anchorbig blink bold charAtfixed fontColor fontSize indexOf italics lastIndexOf link small split strike sub substring sup toLowerCase toUpperCase	Window

## JavaScript Array Object

The Array object is used to store multiple values in a single variable.

### Create an Array

The following code creates an Array object called myCars:

```
var myCars=new Array();
```

There are two ways of adding values to an array (you can add as many values as you need to define as many variables you require).

1: You could also pass an integer argument to control the array's size:

```
var myCars=new Array();
myCars[0]="Saab";
myCars[1]="Volvo";
myCars[2]="BMW";
```

2:**Note:** If you specify numbers or true/false values inside the array then the type of variables will be numeric or Boolean instead of string.

```
var myCars=new Array("Saab","Volvo","BMW");
```

```
var myCars=new Array(3);
myCars[0]="Saab";
myCars[1]="Volvo";
myCars[2]="BMW";
```

### Access an Array

You can refer to a particular element in an array by referring to the name of the array and the index number. The index number starts at 0.

The following code line:

```
document.write(myCars[0]);
```

will result in the following output:

```
Saab
```

### Modify Values in an Array

To modify a value in an existing array, just add a new value to the array with a specified index number:

```
myCars[0]="Opel";
```

-  
Now, the following code line:

```
document.write(myCars[0]);
```

will result in the following output:

```
Opel
```

## JavaScript Date Object

### Create a Date Object

The Date object is used to work with dates and times.

The following code create a Date object called

myDate:

```
var myDate=new Date()
```

**Note:** The Date object will automatically hold the current date and time as its initial value!

### Set Dates

We can easily manipulate the date by using the methods available for the Date object.

In the example below we set a Date object to a specific date (14th January 2010):

```
var myDate=new Date();
myDate.setFullYear(2010,0,14);
```

And in the following example we set a Date object to be 5 days into the future:

```
var myDate=new Date();
myDate.setDate(myDate.getDate()
```

**Note:** If adding five days to a date shifts the month or year, the changes are handled automatically by the Date object itself!

### Compare Two Dates

The Date object is also used to compare two dates.

The following example compares today's date with the 14th January 2010:

```
var myDate=new Date();
myDate.setFullYear(2010,0,14);var
today = new Date();
if (myDate>today)
{
alert("Today is before 14th January 2010");
}
else
{
alert("Today is after 14th January 2010");
}
```

## JavaScript Math Object

### Math Object

The Math object allows you to perform mathematical tasks.

The Math object includes several mathematical constants and methods.

#### Syntax for using properties/methods of Math:

```
var pi_value=Math.PI;
var sqrt_value=Math.sqrt(16);
```

**Note:** Math is not a constructor. All properties and methods of Math can be called by using Math as an object without creating it.

## Mathematical Constants

JavaScript provides eight mathematical constants that can be accessed from the Math object. These are: E,PI, square root of 2, square root of 1/2, natural log of 2, natural log of 10, base-2 log of E, and base-10 log of E.

You may reference these constants from your JavaScript like this:

```
Math.E
Math.PI
Math.SQRT2
Math.SQRT1_2
Math.LN2
Math.LN10
Math.LOG2E
Math.LOG10E
```

## Mathematical Methods

In addition to the mathematical constants that can be accessed from the Math object there are also several methods available.

The following example uses the round() method of the Math object to round a number to the nearest integer:

```
document.write(Math.round(4.7));
```

The code above will result in the following output:

```
5
```

The following example uses the random() method of the Math object to return a random number between 0 and 1:

```
document.write(Math.random());
```

The code above can result in the following output:

```
0.4218824567728053
```

The following example uses the floor() and random() methods of the Math object to return a random number between 0 and 10:

The code above can result in the following output:

```
4
```

JavaScript String Object

String object

The String object is used to manipulate a stored piece of text.

### Examples of use:

The following example uses the length property of the String object to find the length of a string:

```
var txt="Hello world!";
document.write(txt.length);
```

The code above will result in the following output:

The following example uses the toUpperCase() method of the String object to convert a string to uppercase

```
12
```

letters:

```
var txt="Hello world!";
document.write(txt.toUpperCase());
```

The code above will result in the following output:

```
HELLO WORLD!
```

### Window Object

The Window object is the top level object in the JavaScript

hierarchy. The Window object represents a browser window.

A Window object is created automatically with every instance of a <body> or <frameset> tag.

**IE:** Internet Explorer, **F:** Firefox, **O:** Opera.

#### Window Object Collections

Collection	Description	IE	F	O
frames[]	Returns all named frames in the window	4	1	9

#### Window Object Properties

Property	Description	IE	F	O
<b>closed</b>	Returns whether or not a window has been closed	4	1	9
<b>defaultStatus</b>	Sets or returns the default text in the statusbar of the window	4	No	9
<b>document</b>	See Document object	4	1	9
<b>history</b>	See History object	4	1	9
<b>length</b>	Sets or returns the number of frames in the window	4	1	9
<b>location</b>	See Location object	4	1	9
<b>name</b>	Sets or returns the name of the window	4	1	9
<b>opener</b>	Returns a reference to the window that created the window	4	1	9
<b>outerHeight</b>	Sets or returns the outer height of a window	No	1	No
<b>outerWidth</b>	Sets or returns the outer width of a window	No	1	No
<b>pageXOffset</b>	Sets or returns the X position of the current page in relation to the upper left corner of a window's display area	No	No	No
<b>pageYOffset</b>	Sets or returns the Y position of the current page in relation to the upper left corner of a window's display area	No	No	No
<b>parent</b>	Returns the parent window	4	1	9
<b>personalbar</b>	Sets whether or not the browser's personal bar (or directories bar) should be visible			
<b>scrollbars</b>	Sets whether or not the scrollbars should be visible			
<b>self</b>	Returns a reference to the current window	4	1	9
<b>status</b>	Sets the text in the statusbar of a window	4	No	9
<b>statusbar</b>	Sets whether or not the browser's statusbar should be visible			
<b>toolbar</b>	Sets whether or not the browser's tool bar is visible or not (can only be set before the window is opened and you must have UniversalBrowserWrite privilege)			
<b>top</b>	Returns the topmost ancestor window	4	1	9

#### Window Object Methods

Method	Description	IE	F	O
<b>alert()</b>	Displays an alert box with a message and an OK button	4	1	9
<b>blur()</b>	Removes focus from the current window	4	1	9
<b>clearInterval()</b>	Cancels a timeout set with setInterval()	4	1	9
<b>clearTimeout()</b>	Cancels a timeout set with setTimeout()	4	1	9
<b>close()</b>	Closes the current window	4	1	9

<u>confirm()</u>	Displays a dialog box with a message and an OK and a Cancel button	4	1	9
<u>createPopup()</u>	Creates a pop-up window	4	No	No
<u>focus()</u>	Sets focus to the current window	4	1	9
<u>moveBy()</u>	Moves a window relative to its current position	4	1	9
<u>moveTo()</u>	Moves a window to the specified position	4	1	9
<u>open()</u>	Opens a new browser window	4	1	9
<u>print()</u>	Prints the contents of the current window	5	1	9
<u>prompt()</u>	Displays a dialog box that prompts the user for input	4	1	9
<u>resizeBy()</u>	Resizes a window by the specified pixels	4	1	9
<u>resizeTo()</u>	Resizes a window to the specified width and height	4	1.5	9
<u>scrollBy()</u>	Scrolls the content by the specified number of pixels	4	1	9
<u>scrollTo()</u>	Scrolls the content to the specified coordinates	4	1	9
<u>setInterval()</u>	Evaluates an expression at specified intervals	4	1	9
<u>setTimeout()</u>	Evaluates an expression after a specified number of milliseconds	4	1	9

## Document Object

The Document object represents the entire HTML document and can be used to access all elements in a page.

The Document object is part of the Window object and is accessed through the window.document property.

**IE:** Internet Explorer, **F:** Firefox, **O:** Opera, **W3C:** World Wide Web Consortium (Internet Standard).

## Document Object Collections

Collection	Description	IE	F	O	W3C
<u>anchors[]</u>	Returns a reference to all Anchor objects in the document	4	1	9	Yes
<u>forms[]</u>	Returns a reference to all Form objects in the document	4	1	9	Yes
<u>images[]</u>	Returns a reference to all Image objects in the document	4	1	9	Yes
<u>links[]</u>	Returns a reference to all Area and Link objects in the document	4	1	9	Yes

## Document Object Properties

Property	Description	IE	F	O	W3C
body	Gives direct access to the <body> element				
<u>cookie</u>	Sets or returns all cookies associated with the current document	4	1	9	Yes
domain	Returns the domain name for the current document	4	1	9	Yes
<u>lastModified</u>	Returns the date and time a document was last modified	4	1	No	No



<u>referrer</u>	Returns the URL of the document that loaded the current document	4	1	9	Yes
<u>title</u>	Returns the title of the current document	4	1	9	Yes
<u>URL</u>	Returns the URL of the current document	4	1	9	Yes

## Document Object Methods

Method	Description	IE	F	O	W3C
<u>close()</u>	Closes an output stream opened with the document.open() method, and displays the collected data	4	1	9	Yes
<u>getElementById()</u>	Returns a reference to the first object with the specified id	5	1	9	Yes
<u>getElementsByName()</u>	Returns a collection of objects with the specified name	5	1	9	Yes
<u>getElementsByTagName()</u>	Returns a collection of objects with the specified tagname	5	1	9	Yes
<u>open()</u>	Opens a stream to collect the output from any document.write() or document.writeln() methods	4	1	9	Yes
<u>write()</u>	Writes HTML expressions or JavaScript code to a document	4	1	9	Yes
<u>writeln()</u>	Identical to the write() method, with the addition of writing a new line character after each expression	4	1	9	Yes

## History Object

The History object is actually a JavaScript object, not an HTML DOM object.

The History object is automatically created by the JavaScript runtime engine and consists of an array of URLs. These URLs are the URLs the user has visited within a browser window.

The History object is part of the Window object and is accessed through the window.history property.

**IE:** Internet Explorer, **F:** Firefox, **O:** Opera.

## History Object Properties

Property	Description	IE	F	O
<u>length</u>	Returns the number of elements in the history list	4	1	9

## History Object Methods

Method	Description	IE	F	O
<u>back()</u>	Loads the previous URL in the history list	4	1	9
<u>forward()</u>	Loads the next URL in the history list	4	1	9
<u>go()</u>	Loads a specific page in the history list	4	1	9

## Form Object

The Form object represents an HTML form.

For each instance of a <form> tag in an HTML document, a Form object is created.

**IE:** Internet Explorer, **F:** Firefox, **O:** Opera, **W3C:** World Wide Web Consortium (Internet Standard).

## Form Object Collections

Collection	Description	IE	F	O	W3C
<code>elements[]</code>	Returns an array containing each element in the form	5	1	9	Yes

## Form Object Properties

Property	Description	IE	F	O	W3C
<code>acceptCharset</code>	Sets or returns a list of possible character-sets for the form data	No	No	No	Yes
<code>action</code>	Sets or returns the action attribute of a form	5	1	9	Yes
<code>enctype</code>	Sets or returns the MIME type used to encode the content of a form	6	1	9	Yes
<code>id</code>	Sets or returns the id of a form	5	1	9	Yes
<code>length</code>	Returns the number of elements in a form	5	1	9	Yes
<code>method</code>	Sets or returns the HTTP method for sending data to the server	5	1	9	Yes
<code>name</code>	Sets or returns the name of a form	5	1	9	Yes
<code>target</code>	Sets or returns where to open the action-URL in a form	5	1	9	Yes

## Standard Properties

Property	Description	IE	F	O	W3C
<code>className</code>	Sets or returns the class attribute of an element	5	1	9	Yes
<code>dir</code>	Sets or returns the direction of text	5	1	9	Yes
<code>lang</code>	Sets or returns the language code for an element	5	1	9	Yes
<code>title</code>	Sets or returns an element's advisory title	5	1	9	Yes

## Form Object Methods

Method	Description	IE	F	O	W3C
<code>reset()</code>	Resets the values of all elements in a form	5	1	9	Yes
<code>submit()</code>	Submits a form	5	1	9	Yes

## Image Object

The Image object represents an embedded image.

For each instance of an <img> tag in an HTML document, an Image object is created.

**IE:** Internet Explorer, **F:** Firefox, **O:** Opera, **W3C:** World Wide Web Consortium (Internet Standard).

## Image Object Properties

Property	Description	IE	F	O	W3C
<code>align</code>	Sets or returns how to align an image according to the surrounding text	5	1	9	Yes
<code>alt</code>	Sets or returns an alternate text to be displayed, if a browser cannot show an image	5	1	9	Yes

<u>border</u> <u>complete</u>	Sets or returns the border around an image	4	1	9	Yes
	Returns whether or not the browser has finished loading the image	4	1	9	No
<u>height</u> <u>hspace</u>	Sets or returns the height of an image	4	1	9	Yes
	Sets or returns the white space on the left and right side of the image	4	1	9	Yes
<u>id</u> <u>isMap</u> <u>longDesc</u>	Sets or returns the id of the image	4	1	9	Yes
	Returns whether or not an image is a server-side image map	5	1	9	Yes
	Sets or returns a URL to a document containing a description of the image	6	1	9	Yes
<u>lowsrc</u> <u>name</u> <u>src</u> <u>useMap</u>	Sets or returns a URL to a low-resolution version of an image	4	1	9	No
	Sets or returns the name of an image	4	1	9	Yes
	Sets or returns the URL of an image	4	1	9	Yes
	Sets or returns the value of the usemap attribute of an client-side image map	5	1	9	Yes
<u>vspace</u>	Sets or returns the white space on the top and bottom of the image	4	1	9	Yes
<u>width</u>	Sets or returns the width of an image	4	1	9	Yes

## Standard Properties

Property	Description	IE	F	O	W3C
<u>className</u>	Sets or returns the class attribute of an element	5	1	9	Yes
<u>title</u>	Sets or returns an element's advisory title	5	1	9	Yes

## Area Object

The Area object represents an area of an image-map (An image-map is an image with clickable regions).For each instance of an <area> tag in an HTML document, an Area object is created.

**IE:** Internet Explorer, **F:** Firefox, **O:** Opera, **W3C:** World Wide Web Consortium (Internet Standard).

### Area Object Properties

Property	Description	IE	F	O	W3C
<u>accessKey</u> <u>alt</u>	Sets or returns the keyboard key to access an area	5	1	No	Yes
	Sets or returns an alternate text to be displayed, if a browser cannot show an area	5	1	9	Yes
<u>coords</u>	Sets or returns the coordinates of a clickable area in an image-map	5	1	9	Yes
<u>hash</u>	Sets or returns the anchor part of the URL in an area	4	1	No	No
<u>host</u>	Sets or returns the hostname and port of the URL in an area	4	1	No	No
<u>href</u>	Sets or returns the URL of a link in an image-map	4	1	9	Yes
<u>id</u>	Sets or returns the id of an area	4	1	9	Yes
<u>noHref</u>	Sets or returns whether an area should be active or inactive	5	1	9	Yes
<u>pathname</u>	Sets or returns the pathname of the URL in an area	4	1	9	No
<u>protocol</u>	Sets or returns the protocol of the URL in an area	4	1	9	No
<u>search</u>	Sets or returns the query string part of the URL in an area	4	1	9	No
<u>shape</u>	Sets or returns the shape of an area in an image-map	5	1	9	Yes
<u>tabIndex</u>	Sets or returns the tab order for an area	5	1	9	Yes
<u>target</u>	Sets or returns where to open the link-URL in an area	4	1	9	Yes

## Standard Properties

Property	Description	IE	F	O	W3C
<a href="#">className</a>	Sets or returns the class attribute of an element	5	1	9	Yes
<a href="#">dir</a>	Sets or returns the direction of text	5	1	9	Yes
<a href="#">lang</a>	Sets or returns the language code for an element	5	1	9	Yes
<a href="#">title</a>	Sets or returns an element's advisory title	5	1	9	Yes

## Navigator Object

The Navigator object is actually a JavaScript object, not an HTML DOM object.

The Navigator object is automatically created by the JavaScript runtime engine and contains information about the client browser.

**IE:** Internet Explorer, **F:** Firefox, **O:** Opera.

### Navigator Object Collections

Collection	Description	IE	F	O
<a href="#">plugins[]</a>	Returns a reference to all embedded objects in the document	4	1	9

### Navigator Object Properties

Property	Description	IE	F	O
<a href="#">appCodeName</a>	Returns the code name of the browser	4	1	9
<a href="#">appMinorVersion</a>	Returns the minor version of the browser	4	No	No
<a href="#">appName</a>	Returns the name of the browser	4	1	9
<a href="#">appVersion</a>	Returns the platform and version of the browser	4	1	9
<a href="#">browserLanguage</a>	Returns the current browser language	4	No	9
<a href="#">cookieEnabled</a>	Returns a Boolean value that specifies whether cookies are enabled in the browser	4	1	9
<a href="#">cpuClass</a>	Returns the CPU class of the browser's system	4	No	No
<a href="#">onLine</a>	Returns a Boolean value that specifies whether the system is in offline mode	4	No	No
<a href="#">platform</a>	Returns the operating system platform	4	1	9
<a href="#">systemLanguage</a>	Returns the default language used by the OS	4	No	No
<a href="#">userAgent</a>	Returns the value of the user-agent header sent by the client to the server	4	1	9
<a href="#">userLanguage</a>	Returns the OS' natural language setting	4	No	9

### Navigator Object Methods

Method	Description	IE	F	O
<a href="#">javaEnabled()</a>	Specifies whether or not the browser has Java enabled	4	1	9
<a href="#">taintEnabled()</a>	Specifies whether or not the browser has data tainting enabled	4	1	9

## ZIP CODE VALIDATION

<!-- TWO STEPS TO INSTALL ZIP CODE VALIDATION:

1. Copy the coding into the HEAD of your HTML document

2. Add the last code into the BODY of your HTML document -->

<!-- STEP ONE: Paste this code into the HEAD of your HTML document -->

<HEAD>

<SCRIPT LANGUAGE="JavaScript">

<!-- Original: Brian Swalwell -->

<!-- This script and many more are available free online at -->

<!-- The JavaScript Source!! <http://javascript.internet.com> -->

<!--Begin

function validateZIP(field)

{

var valid = "0123456789-";

var hyphencount = 0;

if (field.length!=5 && field.length!=10) {

alert("Please enter your 5 digit or 5 digit+4 zip code.");

return false;

}

for (var i=0; i < field.length; i++) {temp = "" + field.substring(i, i+1);

if (temp == "-")

hyphencount++;

if (valid.indexOf(temp) == "-1")

{

alert("Invalid characters in your zip code. Please try again.");

return false;

}

if ((hyphencount > 1) || ((field.length==10) && ""+field.charAt(5)!="-"))

{

alert("The hyphen character should be used with a properly formatted 5 digit+four zip code, like '12345-6789'. Please try again.");

return false;

}

```

}

return true;
}

// End -->

</script>

</HEAD>

<!-- STEP TWO: Copy this code into the BODY of your HTML document -->

<BODY>

<center>

<form name=zip onSubmit="return validateZIP(this.zip.value)">

Zip: <input type=text size=30 name=zip>

<input type=submit value="Submit">

</form>

</center>

<p><center>

Free JavaScripts provided

by The JavaScript Source

</center><p>

```

## JavaScript Form Validation

JavaScript Form Validation is a way to ensure that the data users enter into a form is correct before it gets submitted. This helps ensure that things like emails, passwords, and other important details are entered properly, making the user experience smoother and the data more accurate.

### Steps for Form Validation in JavaScript

When we validate a form in JavaScript, we typically follow these steps:

- **Data Retrieval:**
  - The first step is to get the user's values entered into the form fields (like name, email, password, etc.). This is done using `document.forms.RegForm`, which refers to the form with the name "RegForm".
- **Data Validation:**
  - Name Validation: We check to make sure the name field isn't empty and doesn't contain any numbers.
  - Address Validation: We check that the address field isn't empty.

- - o Email Validation: We make sure that the email field isn't empty and that it includes the "@" symbol.
    - o Password Validation: We ensure that the password field isn't empty and that the password is at least 6 characters long.
    - o Course Selection Validation: We check that a course has been selected from a dropdown list.
  - **Error Handling:**
    - o If any of the checks fail, an alert message is shown to the user using window.alert, telling them what's wrong.
    - o The form focuses on the field that needs attention, helping the user easily fix the error.
  - **Submission Control:**
    - o If all the validation checks pass, the function returns true, meaning the form can be submitted. If not, it returns false, stopping the form from being submitted.
  - **Focus Adjustment:**
    - o The form automatically focuses on the first field that has an error, guiding the user to fix it.
- Types of Form Validation

- **Client-side Validation:**
  - o This is done in the user's browser before the form is submitted. It provides quick feedback to the user, helping them fix errors without sending data to the server first.
- **Server-side Validation:**
  - o Even though client-side validation is useful, it's important to check the data again on the server. This ensures that the data is correct, even if someone tries to bypass the validation in the browser.

#### Various Use case

Below are the some use cases of Form Validation in JavaScript.

##### 1. [Form validation using jQuery](#)

Here, we have validated a simple form that consists of a username, password, and a confirmed password using **jQuery**,

##### 2. [Number validation in JavaScript](#)

Sometimes the data entered into a text field needs to be in the right format and must be of a particular type in order to effectively use the form. For instance, Phone number, Roll number, etc are some details that must be in digits not in the alphabet.

##### 3. [Password Validation Form Using JavaScript](#)

The password Validation form is used to check the password requirements such as the password must have at least one Uppercase, or lowercase, number, and the length of the password.

##### 4. [How to validate confirm password using JavaScript ?](#)

You can validate a confirm password field using JavaScript by comparing it with the original password field.

##### 5. [JavaScript Program to Validate Password using Regular Expressions](#)

A [Regular Expression](#) is a sequence of characters that forms a search pattern. The search pattern can be used for text search and text to replace operations. A regular expression can be a single character or a more complicated pattern.

## **JQuery**

jQuery is a fast, small, cross-platform and feature-rich JavaScript library. It is designed to simplify the client-side scripting of HTML. It makes things like HTML document traversal and manipulation, animation, event handling, and AJAX very simple with an easy-to-use API that works on a lot of different type of browsers.

The main purpose of jQuery is to provide an easy way to use JavaScript on your website to make it more interactive and attractive. It is also used to add animation.

### **What is jQuery**

jQuery is a small, light-weight and fast JavaScript library. It is cross-platform and supports different types of browsers. It is also referred as "write less do more" because it takes a lot of common tasks that requires many lines of JavaScript code to accomplish, and binds them into methods that can be called with a single line of code whenever needed. It is also very useful to simplify a lot of the complicated things from JavaScript, like AJAX calls and DOM manipulation.

- jQuery is a small, fast and lightweight JavaScript library.
  - jQuery is platform-independent.
  - jQuery means "write less do more".
  - jQuery simplifies AJAX call and DOM manipulation.
- 

### **jQuery Features**

Following are the important features of jQuery.

- HTML manipulation
  - DOM manipulation
  - DOM element selection
  - CSS manipulation
  - Effects and Animations
  - Utilities
  - AJAX
  - HTML event methods
  - JSON Parsing
  - Extensibility through plug-ins
- 

### **Why jQuery is required**

Sometimes, a question can arise that what is the need of jQuery or what difference it makes on bringing jQuery instead of AJAX/ JavaScript? If jQuery is the replacement of AJAX and JavaScript? For all these questions, you can state the following answers.

- It is very fast and extensible.
  - It facilitates the users to write UI related function codes in minimum possible lines.
  - It improves the performance of an application.
  - Browser's compatible web applications can be developed.
  - It uses mostly new features of new browsers.
- 

So, you can say that out of the lot of JavaScript frameworks, jQuery is the most popular and the most extendable. Many of the biggest companies on the web use jQuery.

Some of these companies are:



- Microsoft
- Google
- IBM
- Netflix

## How to add jQuery to HTML Page?

There are two methods to use jQuery in your HTML page.

### 1. Using jQuery from CDN Link

The easiest method to include jQuery in your HTML page is by using a CDN link. CDN links hosted the jQuery files to the servers that can be easily used without downloading the files.

Include jQuery CDN link to the HTML page using `<script>` tag inside the head section of the page.

### 2. Download the jQuery Files Locally and use them

Visit the jQuery Official Website and download the latest version of jQuery. Then include the downloaded jQuery file into your project. Place the *jquery.min.js* file in a directory within your project, such as `js/`. Next, use `<script>` tag inside the `<head>` section to add jQuery file into your web page.

### Basic Syntax for jQuery Function

In jQuery, the syntax for selecting and manipulating HTML elements is written as:

**`$(selector).action()`**

Where –

- **`$`** – This symbol is used to access jQuery. It's a shorthand for the jQuery function.
- **`(selector)`** – It specifies which HTML elements you want to target. The selector can be any valid CSS selector, such as a class, ID, or tag name.
- **`.action()`** – It represents the action or method that you want to perform on the selected elements. Common actions include manipulating CSS properties, handling events, or performing animations.

### jQuery Selectors

jQuery Selectors are used to select and manipulate HTML elements. They are very important part of jQuery library.

With jQuery selectors, you can find or select HTML elements based on their id, classes, attributes, types and much more from a DOM.

In simple words, you can say that selectors are used to select one or more HTML elements using jQuery and once the element is selected then you can perform various operation on that.

All jQuery selectors start with a dollar sign and parenthesis e.g. `$()`. It is known as the factory function.

#### The `$()` factory function

Every jQuery selector start with this sign `$()`. This sign is known as the factory function. It uses the three basic building blocks while selecting an element in a given document.

|    |            |                                                                                                                                                                     |
|----|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1) | Tag Name:  | It represents a tag name available in the DOM. For example: \$('p') selects all paragraphs'p'in the document.                                                       |
| 2) | Tag ID:    | It represents a tag available with a specific ID in the DOM. For example: \$('#real-id') selects a specific element in the document that has an ID of real-id.      |
| 3) | Tag Class: | It represents a tag available with a specific class in the DOM. For example: \$('real-class') selects all elements in the document that have a class of real-class. |

### How to use Selectors

The jQuery selectors can be used single or with the combination of other selectors. They are required at every steps while using jQuery. They are used to select the exact element that you want from your HTML document.

|    |                         |                                                                           |
|----|-------------------------|---------------------------------------------------------------------------|
|    |                         |                                                                           |
| 1) | Name:                   | It selects all elements that match with the given element name.           |
| 2) | #ID:                    | It selects a single element that matches with the given id.               |
| 3) | .Class:                 | It selects all elements that matches with the given class.                |
| 4) | Universal(*)            | It selects all elements available in a DOM.                               |
| 5) | Multiple Elements A,B,C | It selects the combined results of all the specified selectors A,B and C. |

|                |                           |                                                                            |
|----------------|---------------------------|----------------------------------------------------------------------------|
|                |                           |                                                                            |
| *              | \$("*")                   | It is used to select all elements.                                         |
| #id            | \$("#firstname")          | It will select the element with id="firstname"                             |
| .class         | \$(".primary")            | It will select all elements with class="primary"                           |
| class,.class   | \$(".primary,.secondary") | It will select all elements with the class "primary" or "secondary"        |
| element        | \$("p")                   | It will select all p elements.                                             |
| el1,el2,el3    | \$("h1,div,p")            | It will select all h1, div, and p elements.                                |
| :first         | \$("p:first")             | This will select the first p element                                       |
| :last          | \$("p:last")              | This will select the last p element                                        |
| :even          | \$("tr:even")             | This will select all even tr elements                                      |
| :odd           | \$("tr:odd")              | This will select all odd tr elements                                       |
| :first-child   | \$("p:first-child")       | It will select all p elements that are the first child of their parent     |
| :first-of-type | \$("p:first-of-type")     | It will select all p elements that are the first p element of their parent |
| :last-child    | \$("p:last-child")        | It will select all p elements that are the last child of their parent      |
| :last-of-type  | \$("p:last-of-type")      | It will select all p elements that are the last p element of their parent  |

|                                   |                                          |                                                                                                          |
|-----------------------------------|------------------------------------------|----------------------------------------------------------------------------------------------------------|
| <code>:nth-child(n)</code>        | <code>\$("p:nth-child(2)")</code>        | This will select all p elements that are the 2nd child of their parent                                   |
| <code>:nth-last-child(n)</code>   | <code>\$("p:nth-last-child(2)")</code>   | This will select all p elements that are the 2nd child of their parent, counting from the last child     |
| <code>:nth-of-type(n)</code>      | <code>\$("p:nth-of-type(2)")</code>      | It will select all p elements that are the 2nd p element of their parent                                 |
| <code>:nth-last-of-type(n)</code> | <code>\$("p:nth-last-of-type(2)")</code> | This will select all p elements that are the 2nd p element of their parent, counting from the last child |
| <code>:only-child</code>          | <code>\$("p:only-child")</code>          | It will select all p elements that are the only child of their parent                                    |
| <code>:only-of-type</code>        | <code>\$("p:only-of-type")</code>        | It will select all p elements that are the only child, of its type, of their parent                      |
| <code>parent &gt; child</code>    | <code>\$("div &gt; p")</code>            | It will select all p elements that are a direct child of a div element                                   |
| <code>parent descendant</code>    | <code>\$("div p")</code>                 | It will select all p elements that are descendants of a div element                                      |
| <code>element + next</code>       | <code>\$("div + p")</code>               | It selects the p element that are next to each div elements                                              |
| <code>element ~ siblings</code>   | <code>\$("div ~ p")</code>               | It selects all p elements that are siblings of a div element                                             |
| <code>:eq(index)</code>           | <code>\$("ul li:eq(3)")</code>           | It will select the fourth element in a list (index starts at 0)                                          |
| <code>:gt(no)</code>              | <code>\$("ul li:gt(3)")</code>           | Select the list elements with an index greater than 3                                                    |
| <code>:lt(no)</code>              | <code>\$("ul li:lt(3)")</code>           | Select the list elements with an index less than 3                                                       |

|                    |                             |                                                                        |
|--------------------|-----------------------------|------------------------------------------------------------------------|
| :not(selector)     | \$("input:not(:empty)")     | Select all input elements that are not empty                           |
| :header            | \$(":header")               | Select all header elements h1, h2 ...                                  |
| :animated          | \$(":animated")             | Select all animated elements                                           |
| :focus             | \$(":focus")                | Select the element that currently has focus                            |
| :contains(text)    | \$(":contains('Hello')")    | Select all elements which contains the text "Hello"                    |
| :has(selector)     | \$("div:has(p)")            | Select all div elements that have a p element                          |
| :empty             | \$(":empty")                | Select all elements that are empty                                     |
| :parent            | \$(":parent")               | Select all elements that are a parent of another element               |
| :hidden            | \$("p:hidden")              | Select all hidden p elements                                           |
| :visible           | \$("table:visible")         | Select all visible tables                                              |
| :root              | \$(":root")                 | It will select the document's root element                             |
| :lang(language)    | \$("p:lang(de)")            | Select all p elements with a lang attribute value starting with "de"   |
| [attribute]        | \$("[href]")                | Select all elements with a href attribute                              |
| [attribute=value]  | \$("[href='default.htm']")  | Select all elements with a href attribute value equal to "default.htm" |
| [attribute!=value] | \$("[href!='default.htm']") | It will select all elements with a href attribute value not equal to   |

|                     |                           |                                                                                                                        |
|---------------------|---------------------------|------------------------------------------------------------------------------------------------------------------------|
|                     |                           | "default.htm"                                                                                                          |
| [attribute\$=value] | \$("[href\$='.jpg']")     | It will select all elements with a href attribute value ending with ".jpg"                                             |
| [attribute =value]  | \$("[title ='Tomorrow']") | Select all elements with a title attribute value equal to 'Tomorrow', or starting with 'Tomorrow' followed by a hyphen |
| [attribute^=value]  | \$("[title^='Tom']")      | Select all elements with a title attribute value starting with "Tom"                                                   |
| [attribute~=value]  | \$("[title~='hello']")    | Select all elements with a title attribute value containing the specific word "hello"                                  |
| [attribute*=value]  | \$("[title*='hello']")    | Select all elements with a title attribute value containing the word "hello"                                           |
| :input              | \$(":input")              | It will select all input elements                                                                                      |
| :text               | \$(":text")               | It will select all input elements with type="text"                                                                     |
| :password           | \$(":password")           | It will select all input elements with type="password"                                                                 |
| :radio              | \$(":radio")              | It will select all input elements with type="radio"                                                                    |
| :checkbox           | \$(":checkbox")           | It will select all input elements with type="checkbox"                                                                 |
| :submit             | \$(":submit")             | It will select all input elements with type="submit"                                                                   |
| :reset              | \$(":reset")              | It will select all input elements with type="reset"                                                                    |

|           |                 |                                                      |
|-----------|-----------------|------------------------------------------------------|
| :button   | \$(":button")   | It will select all input elements with type="button" |
| :image    | \$(":image")    | It will select all input elements with type="image"  |
| :file     | \$(":file")     | It will select all input elements with type="file"   |
| :enabled  | \$(":enabled")  | Select all enabled input elements                    |
| :disabled | \$(":disabled") | It will select all disabled input elements           |
| :selected | \$(":selected") | It will select all selected input elements           |
| :checked  | \$(":checked")  | It will select all checked input elements            |

.

## jQuery Events

jQuery events are the actions that can be detected by your web application. They are used to create dynamic web pages. An event shows the exact moment when something happens

These are some examples of events.

- A mouse click
- An HTML form submission
- A web page loading
- A keystroke on the keyboard
- Scrolling of the web page etc.

These events can be categorized on the basis their types:

### Mouse Events

- click
- dblclick
- mouseenter
- mouseleave

---

### Keyboard Events

- keyup
- keydown
- keypress

---

#### Form Events

- submit
- change
- blur
- focus

---

#### Document/Window Events

- load
- unload
- scroll
- resize

---

#### Syntax for event methods

Most of the DOM events have an equivalent jQuery method. To assign a click events to all paragraph on a page, do this:

1. `$("#p").click ();`  
The next step defines what should happen when the event fires. You must pass a function to the event.
1. `$("#p").click(function(){`
2. `// action goes here!!`
3. `});`

---

### AJAX for data exchange with server

AJAX (Asynchronous JavaScript and XML) is a technique for creating dynamic, asynchronous web applications. It allows the browser to send and receive data from the server without reloading the entire web page. While XML was initially used for data exchange, JSON is now the most commonly used format due to its simplicity.

Here's an overview of how AJAX is used for data exchange with a server:

---

### Steps in an AJAX Request

1. **Create an XMLHttpRequest Object:**
  - This object is used to interact with the server.
2. **Open a Connection:**
  - Specify the HTTP method (GET/POST), URL, and whether the request is asynchronous.
3. **Send the Request:**
  - Send the request to the server.
4. **Handle the Response:**
  - Use a callback function to process the server's response.

---

Example: AJAX with JSON



## Using Vanilla JavaScript

javascript

Copy code

```
// Create a new XMLHttpRequest object
var xhr = new XMLHttpRequest();

// Configure the request: GET method, endpoint URL, and asynchronous
xhr.open("GET", "https://api.example.com/data", true);

// Set up a callback function to handle the response
xhr.onreadystatechange = function () {
 if (xhr.readyState === 4 && xhr.status === 200) {
 // Parse the JSON response
 var responseData = JSON.parse(xhr.responseText);
 console.log(responseData);
 }
};

// Send the request
xhr.send();
```

## Using Fetch API (Modern Approach)

The Fetch API provides a more modern and cleaner way to handle AJAX requests.

javascript

Copy code

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

---

## Using POST with AJAX

To send data to the server using POST:

### Vanilla JavaScript

javascript

Copy code

```
var xhr = new XMLHttpRequest();
xhr.open("POST", "https://api.example.com/submit", true);
xhr.setRequestHeader("Content-Type", "application/json");

// Callback for response
```

```

xhr.onreadystatechange = function () {
 if (xhr.readyState === 4 && xhr.status === 200) {
 console.log("Response:", JSON.parse(xhr.responseText));
 }
};

// Send JSON data
var data = JSON.stringify({ name: "John", age: 30 });
xhr.send(data);

```

Fetch API

```

javascript
Copy code
fetch("https://api.example.com/submit", {
 method: "POST",
 headers: {
 "Content-Type": "application/json"
 },
 body: JSON.stringify({ name: "John", age: 30 }) // Send JSON data
})
 .then(response => response.json())
 .then(data => {
 console.log("Response:", data);
 })
 .catch(error => {
 console.error("Error:", error);
 });

```

---

## Key Components of AJAX

1. **HTTP Methods:**
  - GET: Retrieve data from the server.
  - POST: Send data to the server for processing.
2. **Response Formats:**
  - JSON (most common), XML, HTML, or plain text.
3. **Asynchronous:**
  - AJAX requests are typically asynchronous, meaning the page doesn't freeze while waiting for a server response.
4. **Error Handling:**
  - Check status and readyState (for XMLHttpRequest) or handle errors with .catch (for fetch).

---

## JSON data format

JSON (JavaScript Object Notation) is a lightweight data format commonly used for data interchange. It's easy for humans to read and write, and easy for machines to parse and generate. Below is an overview of the JSON format:

### Structure of JSON

- **Objects:** Represented as a collection of key-value pairs enclosed in curly braces {}.
- **Arrays:** Represented as an ordered list of values enclosed in square brackets [].
- **Values:** Can be strings, numbers, booleans, null, objects, or arrays.

## Syntax Rules

1. **Key-Value Pair:** Keys must be strings enclosed in double quotes ("), followed by a colon (:), and then the value.
2. **String:** Enclosed in double quotes (").
3. **Number:** Can be an integer or a floating-point value.
4. **Boolean:** true or false.
5. **Null:** null.
6. **Nested Objects/Arrays:** Can be embedded inside one another.

### Example JSON

json

Copy code

```
{
 "name": "John Doe",
 "age": 30,
 "isMarried": false,
 "children": ["Jane", "Jack"],
 "address": {
 "street": "123 Main St",
 "city": "Springfield",
 "zipCode": "12345"
 },
 "hobbies": null
}
```

### Use Cases

- Data transfer in APIs (e.g., REST APIs).
- Configuration files (e.g., package.json in Node.js).
- Storing data in NoSQL databases like MongoDB.

### JSON vs. Other Formats

- **Advantages:** Simple, lightweight, widely supported.
- **Drawbacks:** No comments allowed, limited data types.

## UNIT - III

**Angular:** importance of Angular, Understanding Angular, creating a Basic Angular Application, Angular Components, Expressions, Data Binding, Built-in Directives, Custom Directives, Implementing AngularServices in Web Applications.

**React:**

Need of React, Simple React Structure, The Virtual DOM, React Components, Introducing React Components, Creating Components in React, Data and Data Flow in React, Rendering and Life Cycle Methods in React, Working with forms in React, integrating third party libraries, Routing in React.

### Angular

#### Getting Started with Angular

**a. Defination:-**

Angular is an open-source web application framework maintained by Google and a community of developers. It is designed to build dynamic and interactive single-page applications (SPAs) efficiently. With Angular, developers can create robust, scalable, and maintainable web applications.

(Or)

Angular is an open-source, JavaScript framework written in TypeScript. Google maintains it, and its primary purpose is to develop single-page applications. As a framework, Angular has clear advantages while also providing a standard structure for developers to work with. It enables users to create large applications in a maintainable manner.

**b. History**

Angular, initially released in 2010 by Google, has undergone significant transformations over the years. The first version, AngularJS, introduced concepts like two-way data binding and directives. However, as web development evolved, AngularJS faced limitations in terms of performance and flexibility.

In 2016, Angular 2 was released, which was a complete rewrite of AngularJS, focusing on modularity and performance. Since then, Angular has continued to evolve, with regular updates and improvements to meet the demands of modern web development.

**c. Why Angular?**

JavaScript is the most commonly used client-side scripting language. It is written into HTML documents to enable interactions with web pages in many unique ways. As a relatively easy-to-learn language with pervasive support, it is well-suited to develop modern applications.

But is JavaScript ideal for developing single-page applications that require modularity, testability, and developer productivity? Perhaps not.

These days, we have a variety of frameworks and libraries designed to provide alternative solutions. With respect to front-end web development, Angular addresses many, if not all, of the issues developers face when using JavaScript on its own.

**d. Here are some of the features of Angular**

**1. Custom Components**

Angular enables users to build their components that can pack functionality along with rendering logic into reusable pieces.

**2. Data Binding**

Angular enables users to effortlessly move data from JavaScript code to the view, and react to user events without having to write any code manually.

**3. Dependency Injection**

Angular enables users to write modular services and inject them wherever they are needed. This improves the testability and reusability of the same services.

**4. Testing**

- Tests are first-class tools, and Angular has been built from the ground up with testability in mind. You will have the ability to test every part of your application—which is highly recommended.

### 5. Comprehensive

Angular is a full-fledged JavaScript framework and provides out-of-the-box solutions for server communication, routing within your application, and more.

### 6. Browser Compatibility

Angular works cross-platform and compatible with multiple browsers. An Angular application can typically run on all browsers (Eg: Chrome, Firefox) and operating systems, such as Windows, macOS, and Linux.

**7. Two-Way Data Binding:** Angular provides seamless synchronization between the model and the view, allowing for easy management of user inputs.

**8. Directives:** Angular offers a rich set of built-in directives for manipulating the DOM, such as `*ngIf*`, `*ngFor*`, and `*ngSwitch*`.

**9. Routing:** Angular's powerful routing module enables to build SPAs with multiple views and navigation between them.

**10. HTTP Client:** Angular includes an HTTP client module for making server requests, simplifying data fetching and manipulation.

#### e. Advantages of Angular

- **Productivity:** Angular's extensive tooling and ecosystem streamline development tasks, enabling faster project completion.
- **Maintainability:** Angular's modular architecture and clear separation of concerns promote code organization and maintainability.
- **Scalability:** Angular is well-suited for building large-scale applications, thanks to its component-based architecture and robust performance.
- **Community Support:** Being backed by Google and a vast community of developers, Angular enjoys strong community support and continuous improvement.

#### f. Disadvantages of Angular

- **Learning Curve:** Angular has a steep learning curve, especially for beginners, due to its complex concepts and extensive documentation.
- **Performance Overhead:** Angular's powerful features come with a performance cost, and poorly optimized applications may suffer from performance issues.
- **Size:** Angular applications tend to have larger file sizes compared to other frameworks, which may impact load times, especially on mobile devices.
- **Migration:** Upgrading between major Angular versions can be challenging and time-consuming, requiring significant changes to existing codebases.

### g. Angular Prerequisites

There are three main prerequisites.

#### NodeJS

Angular uses Node.js for a large part of its build environment. As a result, to get started with Angular, you will need to have Node.js installed on your system. You can head to the NodeJS official website to download the software. Install the latest version and confirm them on your command prompt by running the following commands:

**Node --version npm --v**



## Angular CLI

The Angular team has created a command-line interface (CLI) tool to make it easier to bootstrap and develop your Angular applications. As it significantly helps to make the process of development easier, we highly recommend using it for your initial angular projects at the least.

To install the CLI, in the command prompt, type the following

commands Installation:

```
npm install -g @angular/cli
```

Confirmation -

```
ng--version
```



## Text Editor

You need a text editor to write and run your code. The most popularly used integrated development environment (IDE) is Visual Studio Code (VS Code). It is a powerful source code editor that is available on Windows, macOS, and Linux platforms.



Now, Let's create our first Angular HelloWorld Application.

## Creating an Angular Application

**Step 1: Install Angular CLI:** Angular CLI (Command Line Interface) is a powerful tool for

scaffolding and managing Angular applications. Install it globally using npm:

```
Npm install -g @angular/cli
```

**Step 2: Create a New Angular Project:** Use Angular CLI to create a new Angular project.

Navigate to the desired directory and run:

```
ng new my-angular-app //creating standalone application
```

(or)

**Ng new my-angular-app --standalone false** //creating non-standalone application project structure is different adding two more files app.module.ts and app-routing.module.ts

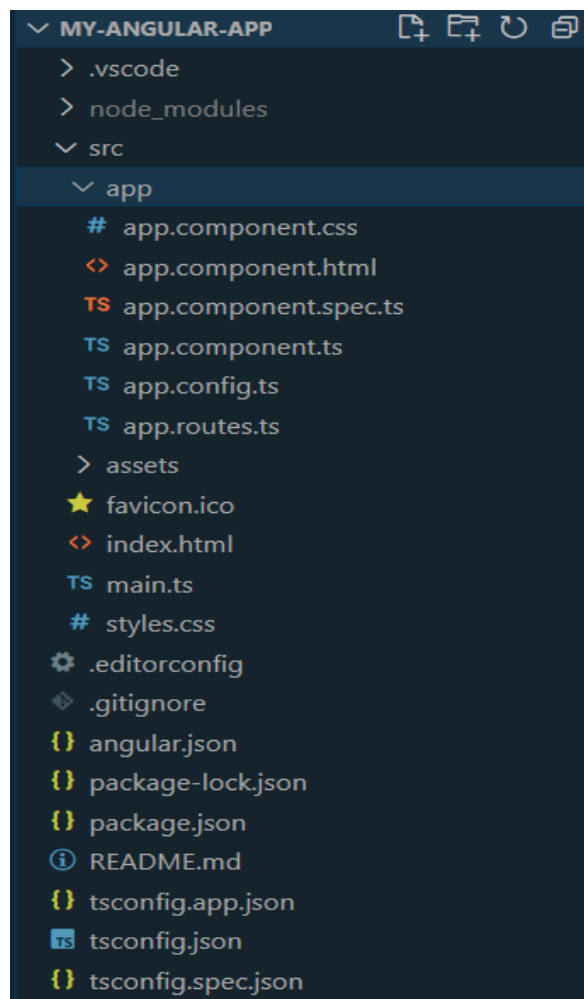
**Step 3: Navigate to the Project Directory:** Move into the newly created project directory:

```
cd my-angular-app
```

**Step 4: Serve the Application:** Launch the development server to see your app in action:

```
ng serve
```

**Folder Structure:**



**Dependencies:**

```
- "dependencies": {
 "@angular/animations":
 "^17.3.0",
 "@angular/common":
 "^17.3.0",
 "@angular/compiler":
 "^17.3.0",
 "@angular/core":
 "^17.3.0",
 "@angular/forms":
 "^17.3.0",
 "@angular/platform-browser":
 "^17.3.0", "@angular/platform-
 browser-dynamic": "^17.3.0",
}
```

### Example:

```
<!-- app.component.html -->

<h1>Hello Angular</h1>
//app.component.ts

import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';

@Component({
 selector: 'app-root',
 standalone: true,
 imports: [RouterOutlet],
 templateUrl:
 './app.component.html', styleUrls:
 ['./app.component.css']
})
export class AppComponent {
 title = 'my-angular-app';
}
```

### Root HTML - index.html(default code)

```
<!doctype html>

<html lang="en">

<head>

 <meta charset="utf-8">

 <title>HelloWorld</title>

 <base href="/">

 <meta name="viewport" content="width=device-width, initial-scale=1">
```



- `<link rel="icon" type="image/x-icon" href="favicon.ico">`

`</head>`

`<body>`

`<app-root></app-root>`

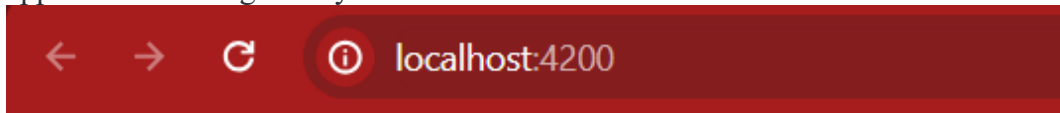
`</body>`

`</html>`

The only main thing in this file is the `<app-root>` element. This is the marker for loading the application. All the application code, styles, and inline templates are dynamically injected into the `index.html` file at run time by the **ng serve** command.

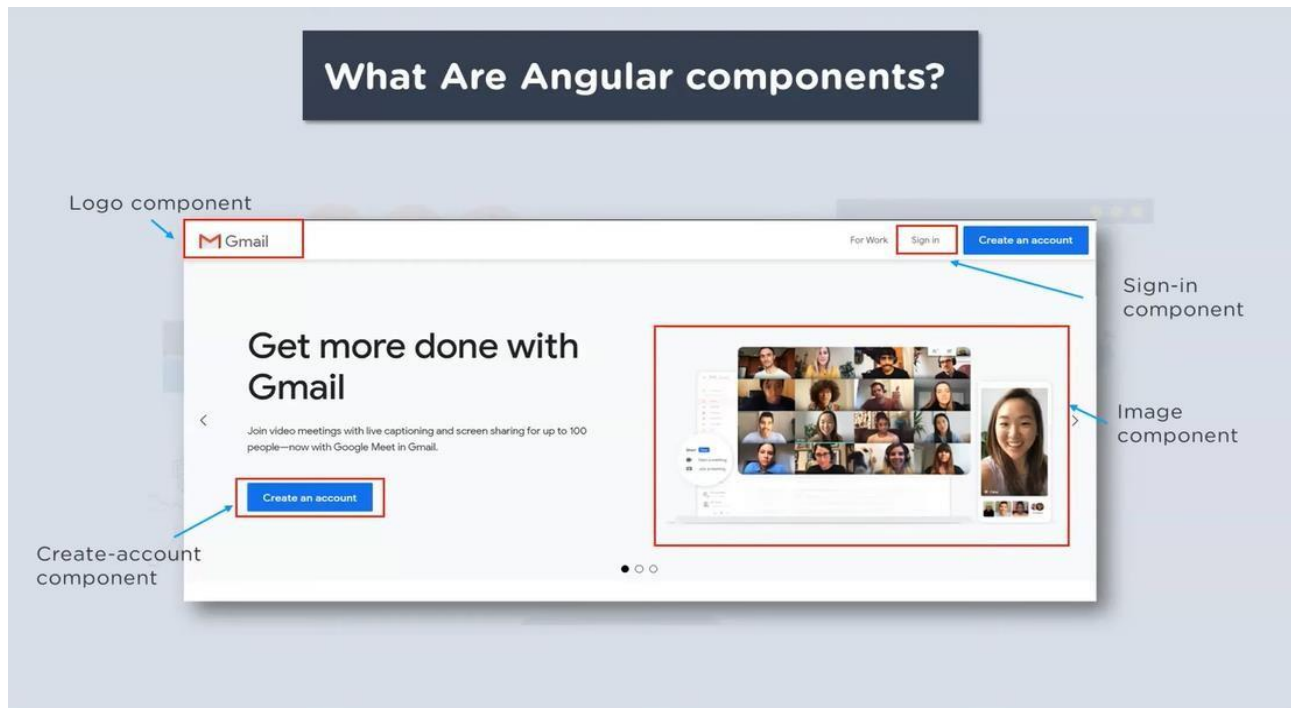
### Output:

Upon running `ng serve`, the Angular CLI will compile the application and launch a development server. Open a web browser and navigate to `http://localhost:4200` to view the application running locally.



# Hello Angular

## \_ 2. Angular Components:-



The above image showing Gmail is a Single Page Application each part consider like a component like logo component, sign-in component etc.,

### Defination:-

The component is the basic building block of Angular. It has a selector, template, style, and other properties, and it specifies the metadata required to process the component.

```
import { Component } from '@angular/core';

@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})

export class AppComponent {
 title = 'MyAngularApp';
}
```

Importing the component decorator from angular core library

Decorating the class with @Component decorator and providing the metadata

Creating class to define data and logic for the view

## - parts of an Angular Component

**An Angular component has several parts, such as:**

### **Selector**

It is the CSS selector that identifies this component in a template. This corresponds to the HTML tag that is included in the parent component. You can create your own [HTML tag](#). However, the same has to be included in the parent component.

### **Template**

It is an inline-defined template for the view. The template can be used to define some markup. The markup could typically include some headings or paragraphs that are displayed on the UI.

### **TemplateUrl**

It is the URL for the external file containing the template for the view.

### **Styles**

These are inline-defined styles to be applied to the component's view

### **styleUrls**

List of URLs to stylesheets to be applied to the component's view.

Before Creating Angular Component create Angular Project using this command `ng new Projectname` or `na new projectname --standalone false`

Creating a Component in Angular 8:

To create a component in any angular application, follow the below steps:

- Get to the angular app via your terminal.(`ng new project_name`)
- Create a component using the following command:

```
ng gc <component_name>
```

OR

```
ng generate component <component_name>
```

- Following files will be created after generating the component:

Note:-write below picture four files in exam important to explaining component

```
Microsoft Windows [Version 10.0.18362.1082]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\hp\demo>ng g c gfg
CREATE src/app/gfg/gfg.component.html (18 bytes)
CREATE src/app/gfg/gfg.component.spec.ts (607 bytes)
CREATE src/app/gfg/gfg.component.ts (263 bytes)
CREATE src/app/gfg/gfg.component.css (0 bytes)
UPDATE src/app/app.module.ts (363 bytes)

C:\Users\hp\demo>
```

Using a component in Angular 8:

- Go to the component.html file and write the necessary HTML code.  
gfg.component.html:

```
<h1>GeeksforGeeks</h1>
```

- Go to the component.css file and write the necessary CSS code.

gfg.component.css:

```
h1{

 color: green;

 font-size: 30px;

}
```

- Write the corresponding code in component.ts file.

**gfg.component.ts:**

```
import { Component, OnInit } from
```

```
@Component({
```

```
 selector: 'app-gfg',
```

```
templateUrl:
-
'./gfg.component.html',

styleUrls:

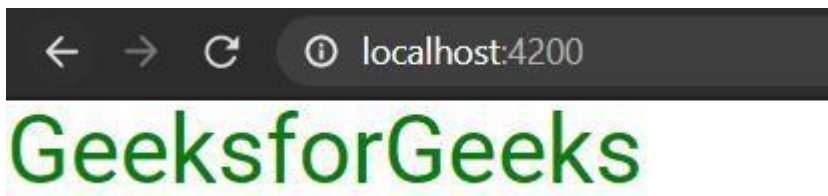
['./gfg.component.css']

})

export class
```

- Run the Angular app using **ng serve**

–open Output:



## Properties, Events & Binding with ngModel

### Data Binding

Data binding is the core concept of Angular 8 and used to define the communication between a component and the DOM. It is a technique to link your data to your view layer. In simple words, you can say that data binding is a communication between your typescript code of your component and your template which user sees. It makes easy to define interactive applications without worrying about pushing and pulling data.

Data binding can be either one-way data binding or two-way data binding.

### One-way databinding

One way databinding is a simple one way communication where HTML template is changed when we make changes in TypeScript code.

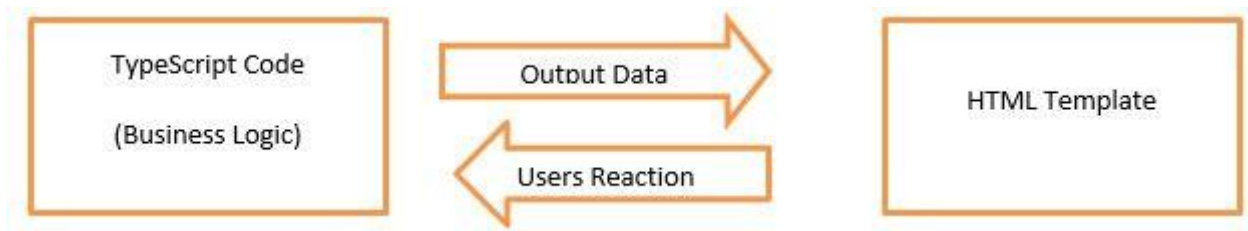
Or

In one-way databinding, the value of the Model is used in the View (HTML page) but you can't update Model from the View. Angular Interpolation / String Interpolation, Property Binding, and Event Binding are the example of one-way databinding.

### Two-way databinding

In two-way databinding, automatic synchronization of data happens between the Model and the View. Here, change is reflected in both components. Whenever you make changes in the Model, it will be reflected in the View and when you make changes in View, it will be reflected in Model.

- This happens immediately and automatically, ensures that the HTML template and the TypeScript code are updated at all times.



Angular provides four types of data binding and they are different on the way of data flowing.

- o [String Interpolation](#)
- o [Property Binding](#)
- o [Event Binding](#)
- o Class binding
- o Style binding
- o [Two-way binding](#)

One way Data Binding:-

1. [String Interpolation](#)
2. [Property Binding](#)
3. [Event Binding](#)
4. Class binding
5. Style binding

### 1. String Interpolation

String Interpolation is a **one-way databinding** technique which is used to output the data from a TypeScript code to HTML template (view). It uses the template expression in **double curly braces** to display the data from the component to the view. String interpolation adds the value of a property from the component.

**For example:**

```
{{ data }}
```

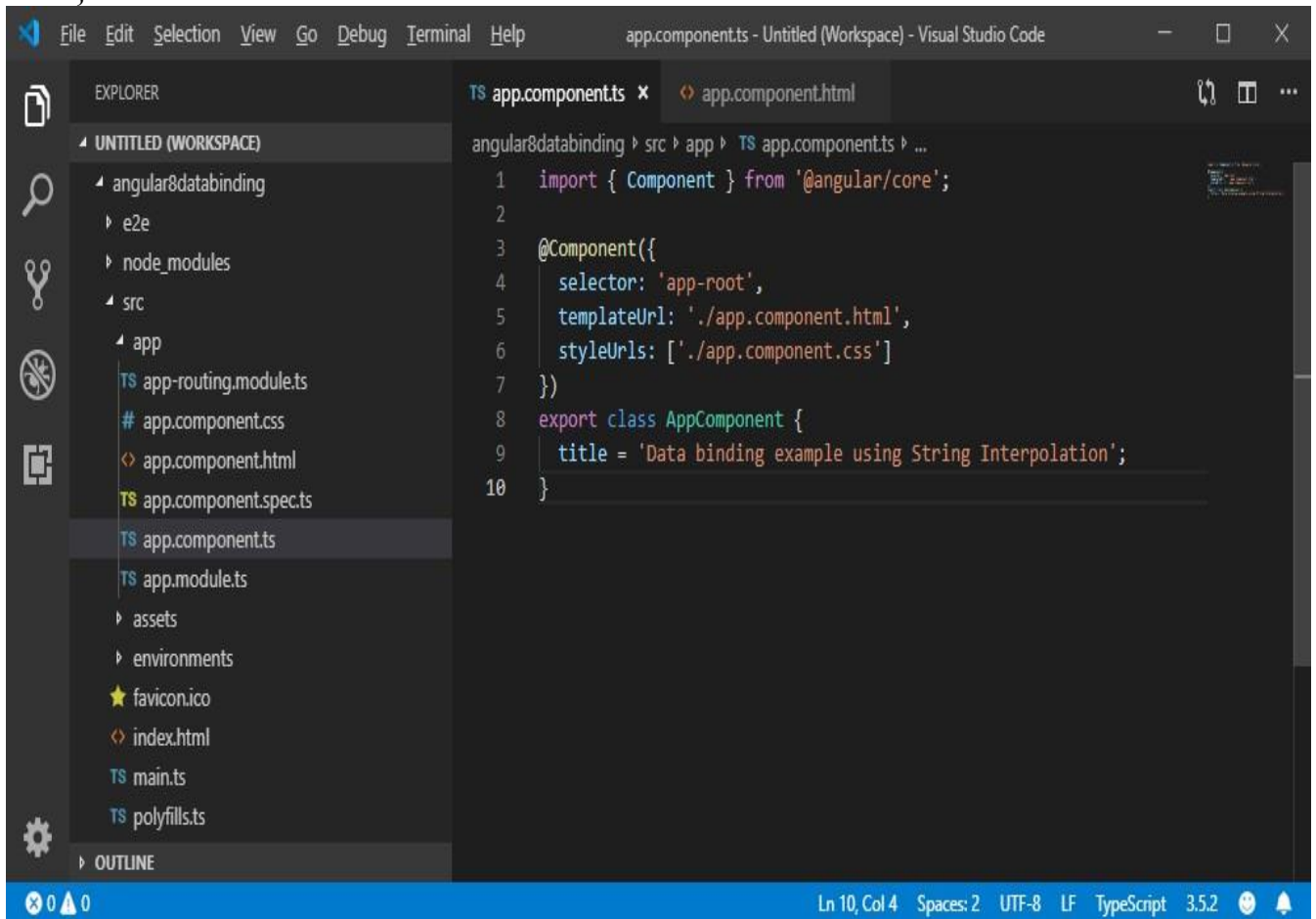
We have already created an Angular project using

Angular CLI. Here, we are using the same project

for this example.

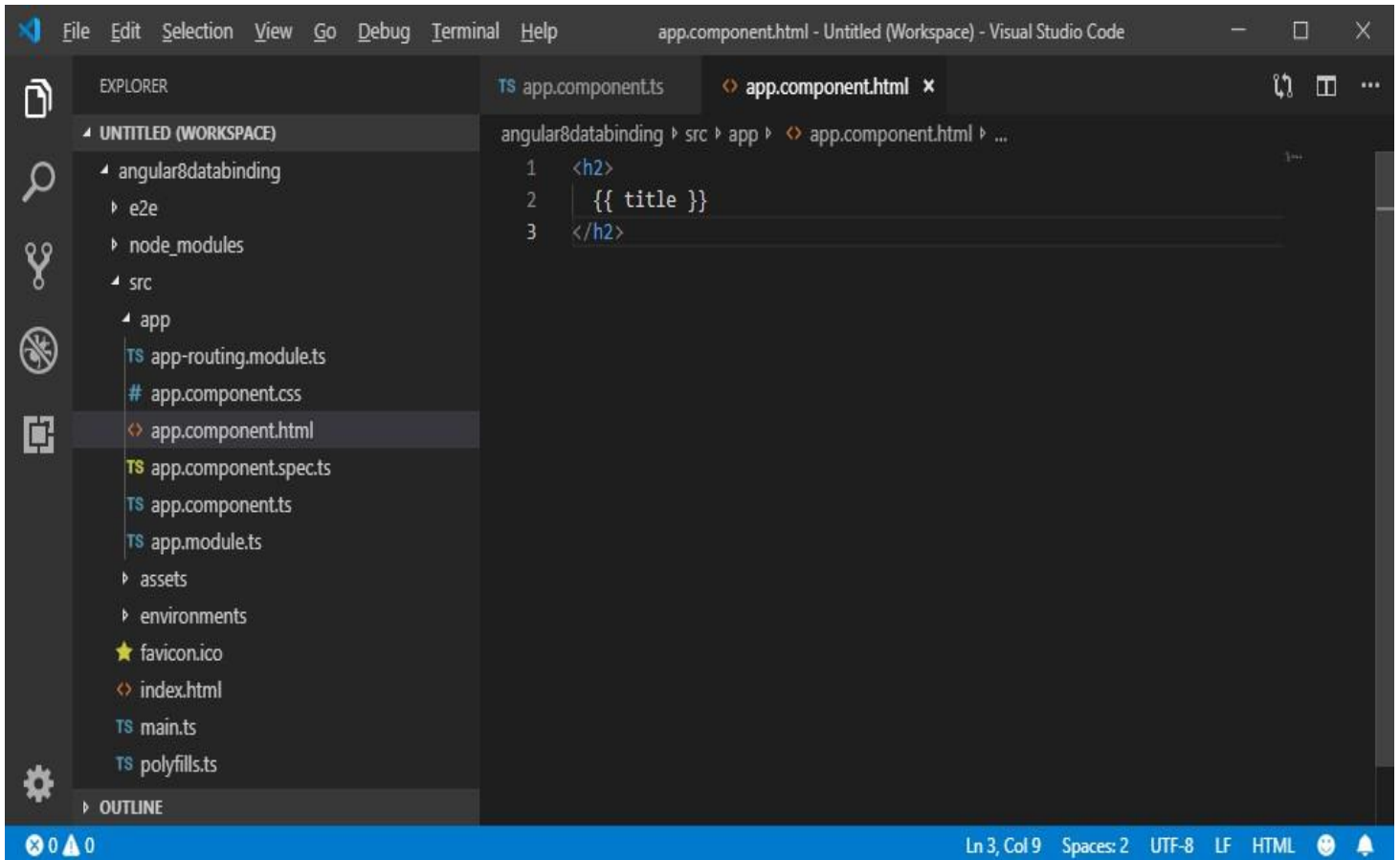
Open **app.component.ts** file and use the following code within the file:

1. `import { Component } from '@angular/core';`
2. `@Component({`
3. `selector: 'app-root',`
4. `templateUrl: './app.component.html',`
5. `styleUrls: ['./app.component.css']`
6. `})`
7. `export class AppComponent {`
8. `title = 'Data binding example using String Interpolation';`
9. `}`



Now, open **app.component.html** and use the following code to see string interpolation.

1. `<h2>`
2. `{{ title }}`
3. `</h2>`



Now, open Node.js command prompt and run the **ng serve** command to see the result.

### Output:

ADVERTISEMENT





- String Interpolation can be used to resolve some other expressions too.

Let's see an example.

### Example:

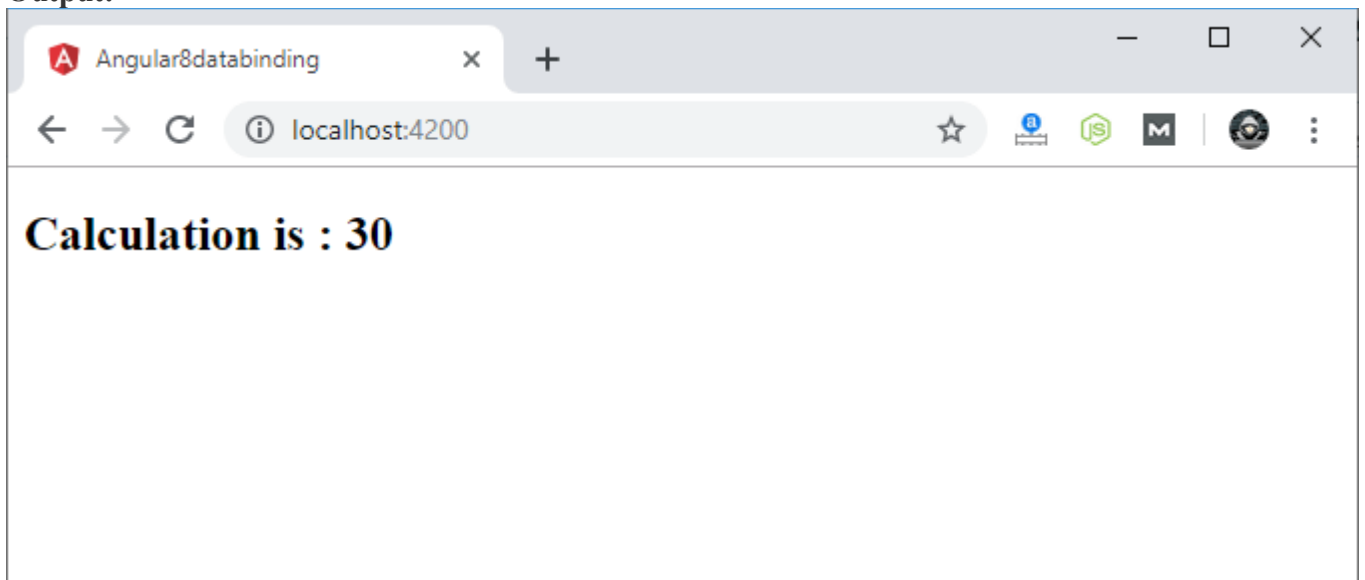
Update the **app.component.ts** file with the following code:

```
1. import { Component } from '@angular/core';
2. @Component({
3. selector: 'app-root',
4. templateUrl: './app.component.html',
5. styleUrls: ['./app.component.css']
6. })
7. export class AppComponent {
8. title = 'Data binding example using String Interpolation';
9. numberA: number = 10;
10. numberB: number = 20;
11. }
```

### app.component.html:

```
1. <h2>Calculation is : {{ numberA + numberB }}</h2>
```

Output:



## 2. Property Binding in Angular 8

Property Binding is also a **one-way data binding** technique. In property binding, we bind a property of a DOM element to a field which is a defined property in our component TypeScript code. Actually Angular internally converts string interpolation into property binding.

**For example:**

```

```

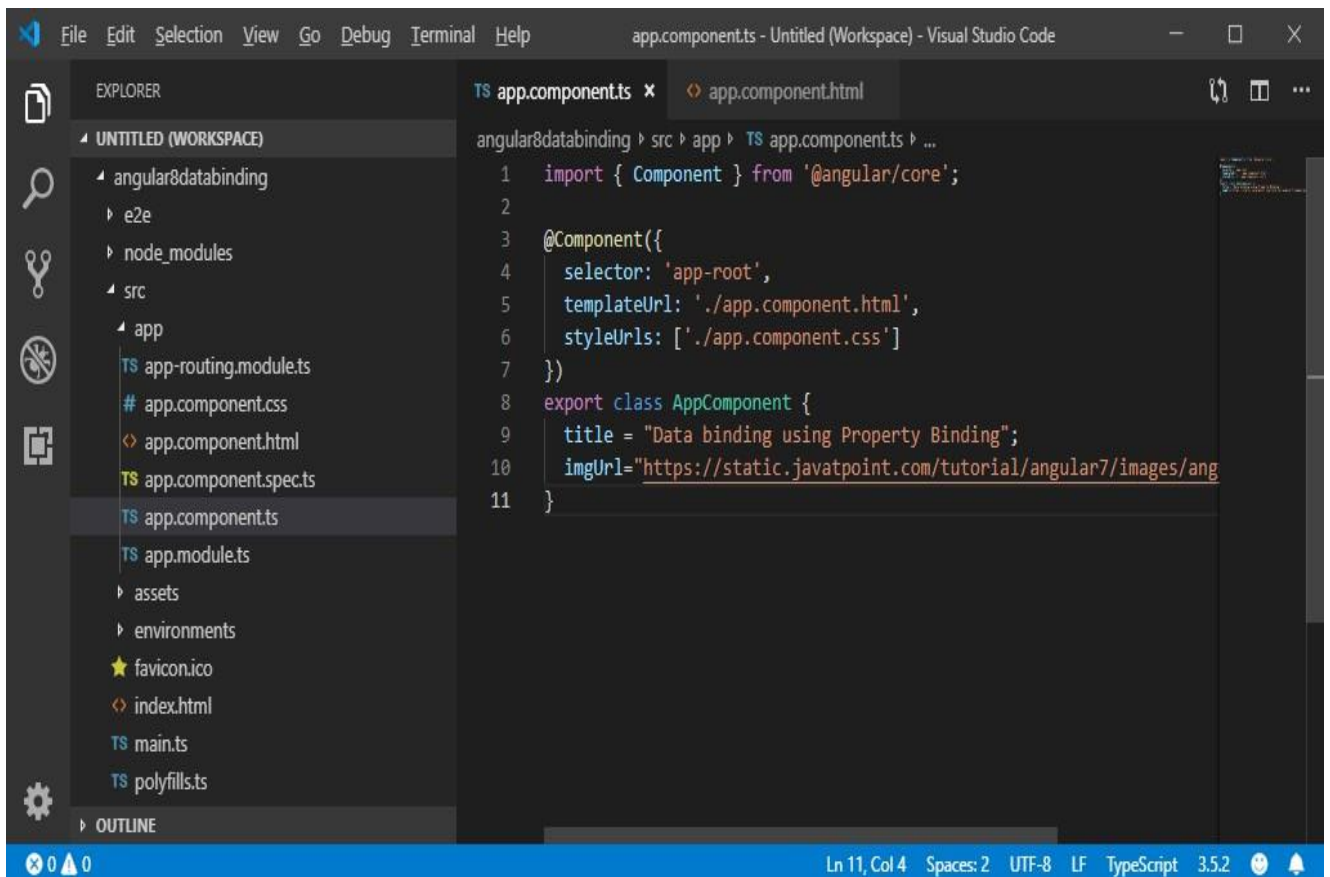
Property binding is preferred over string interpolation because it has shorter and cleaner code. String interpolation should be used when you want to simply display some dynamic data from a component on the view between headings like h1, h2, p etc.

*Note: String Interpolation and Property binding both are one-way binding. Means, if field value in the component changes, Angular will automatically update the DOM. But any changes in the DOM will not be reflected back in the component.*

### Property Binding Example

Open **app.component.ts** file and add the following code:

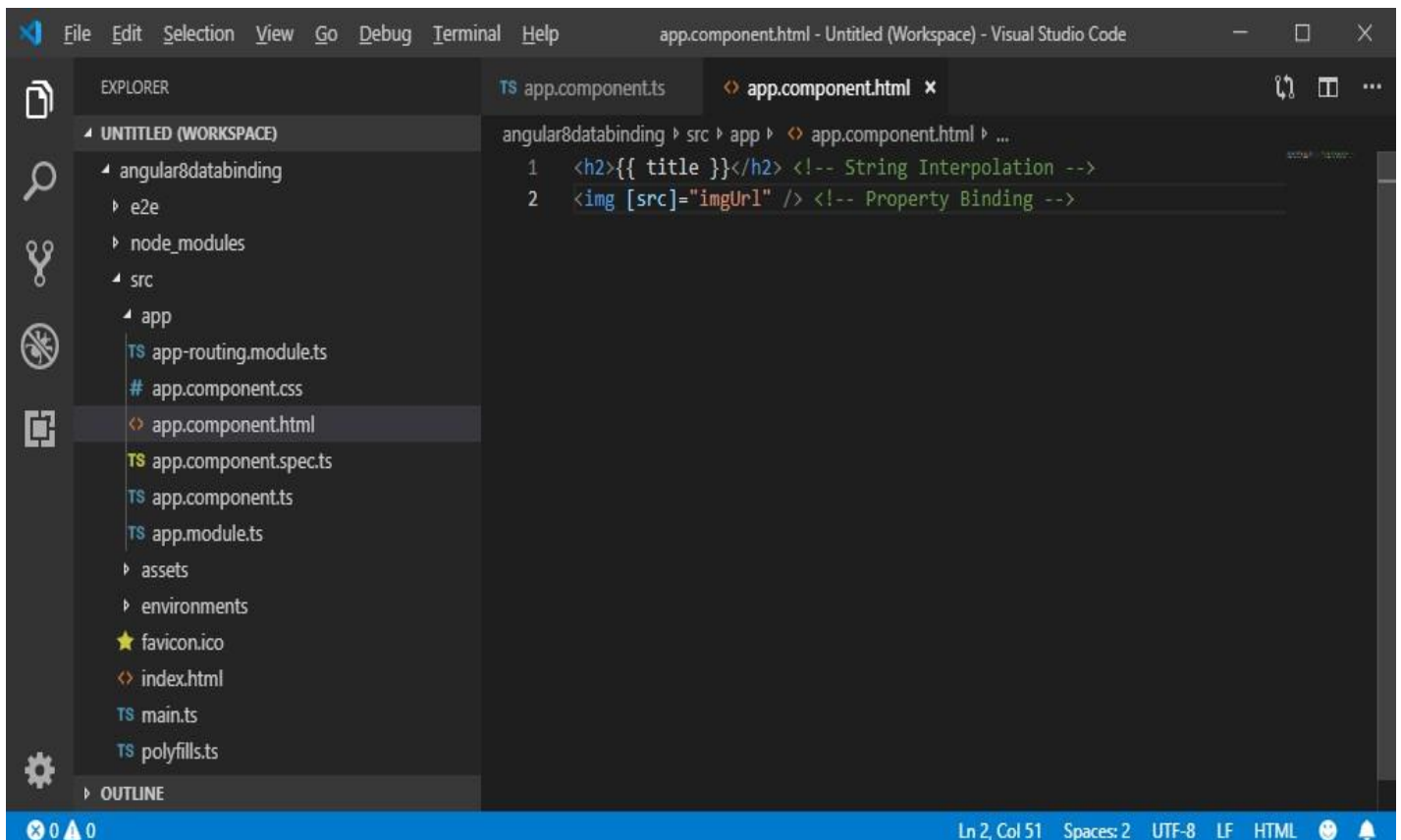
```
1. import { Component } from '@angular/core';
2. @Component({
3. selector: 'app-root',
4. templateUrl: './app.component.html',
5. styleUrls: ['./app.component.css']
6. })
7. export class AppComponent {
8. title = "Data binding using Property Binding";
9. imgUrl="https://static.javatpoint.com/tutorial/angular7/images/angular-7-logo.png";
10. }
```



```
1 import { Component } from '@angular/core';
2
3 @Component({
4 selector: 'app-root',
5 templateUrl: './app.component.html',
6 styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9 title = "Data binding using Property Binding";
10 imgUrl="https://static.javatpoint.com/tutorial/angular7/images/ang
11 }
```

Now, open **app.component.html** and use the following code for property binding:

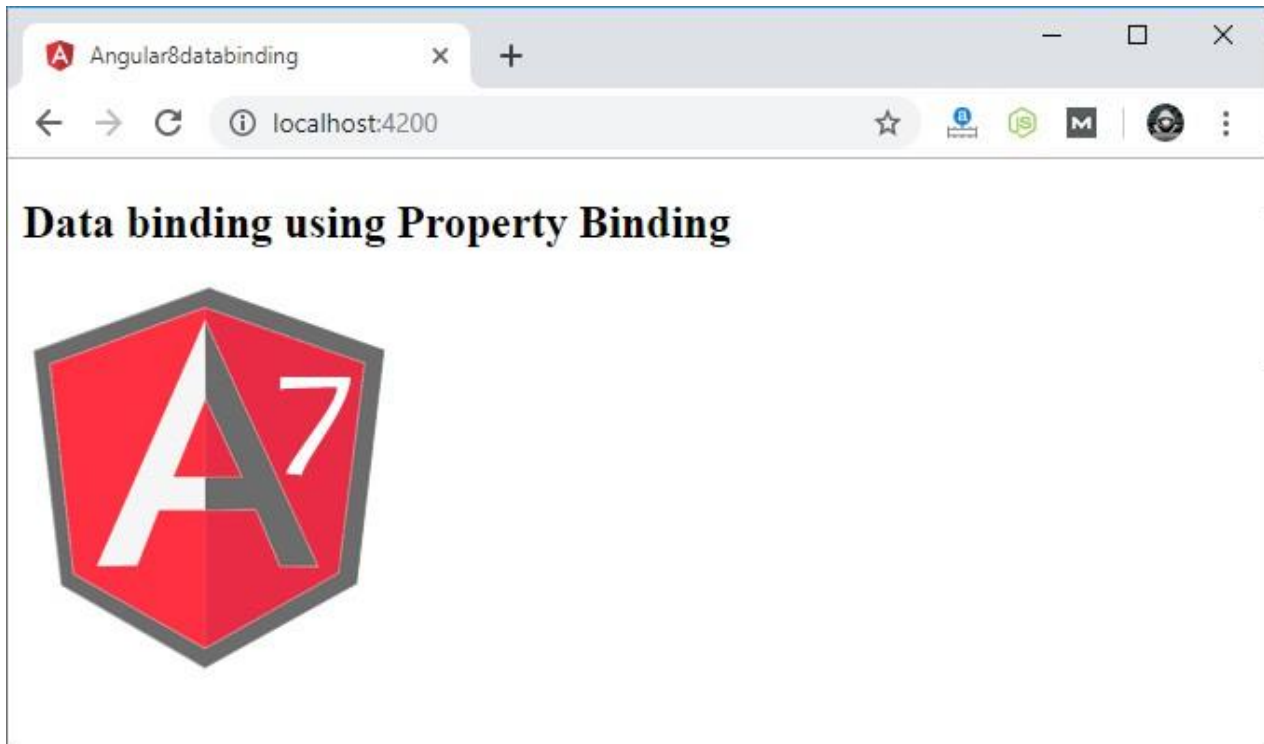
1. `<h2>{{ title }}</h2> <!-- String Interpolation -->`
2. `<img [src]="imgUrl" /> <!-- Property Binding -->`



```
1 <h2>{{ title }}</h2> <!-- String Interpolation -->
2 <!-- Property Binding -->
```

Run the ng serve command and open local host to see the result.

**Output:**



### Event Binding in Angular 8

In Angular 8, event binding is used to handle the events raised from the DOM like button click, mouse move etc. When the DOM event happens (eg. click, change, keyup), it calls the specified method in the component. In the following example, the cookBacon() method from the component is called when the button is clicked:

**For example:**

1. `<button (click)="cookBacon()"></button>`

### Event Binding Example

Let's take a button in the HTML template and handle the click event of this button. To implement event binding, we will bind click event of a button with a method of the component.

Now, open the **app.component.ts** file and use the following code:

Backward Skip 10sPlay VideoForward Skip 10s

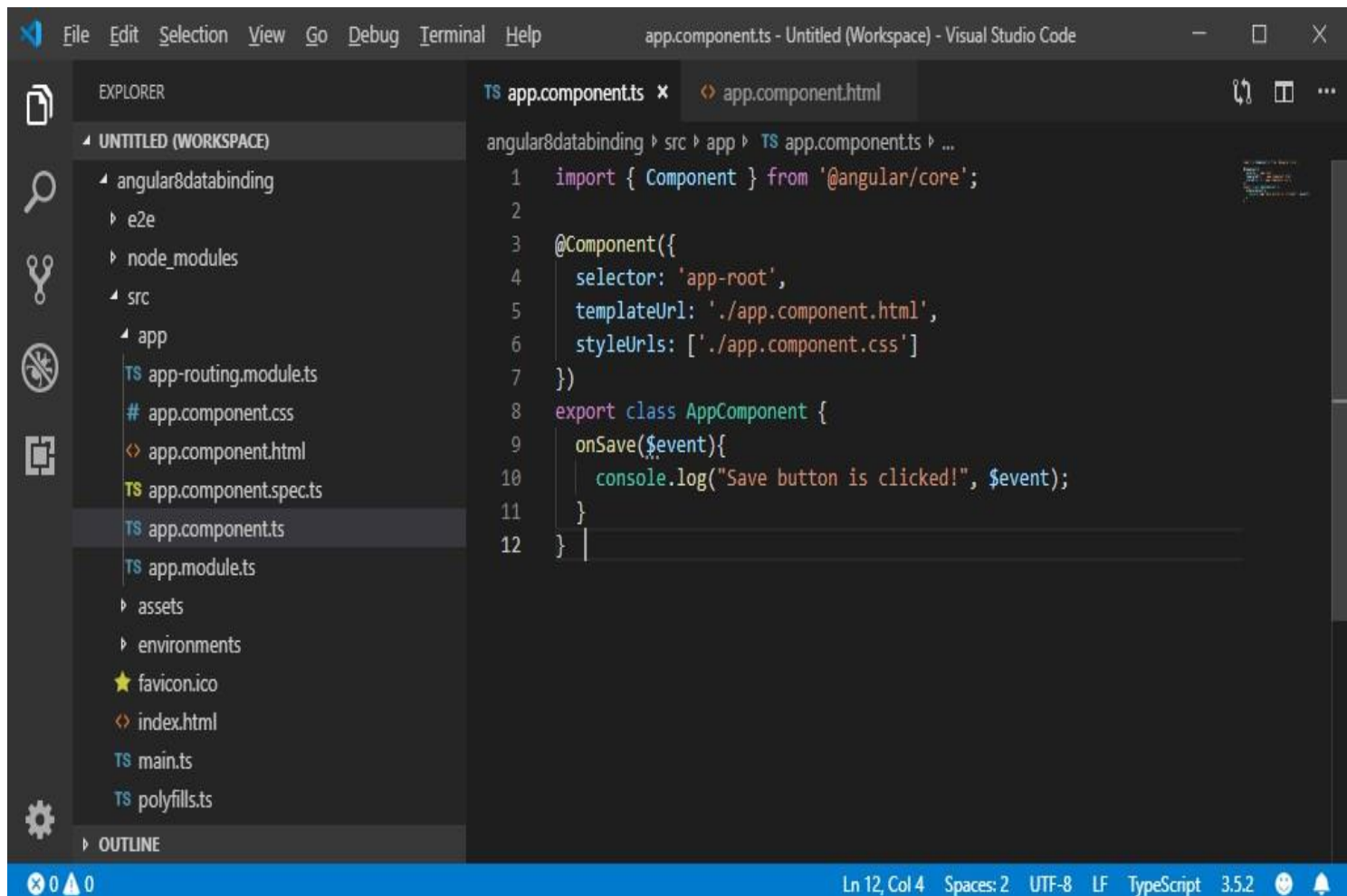
### ADVERTISEMENT

1. `import { Component } from '@angular/core';`
2. `@Component({`
3. `selector: 'app-root',`
4. `templateUrl: './app.component.html',`
5. `styleUrls: ['./app.component.css']`
6. `})`

```

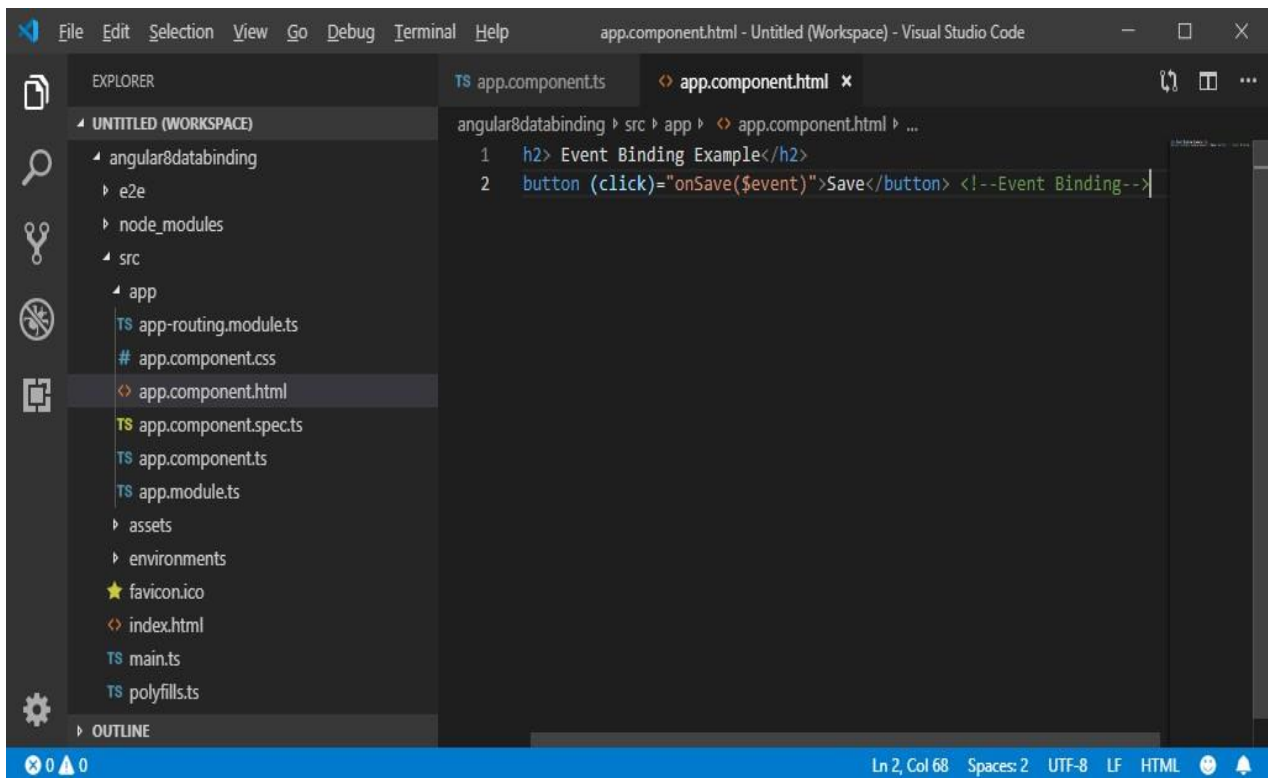
7. _ export class AppComponent {
8. onSave($event){
9. console.log("Save button is clicked!", $event);
10. }
11. }

```



### app.component.html:

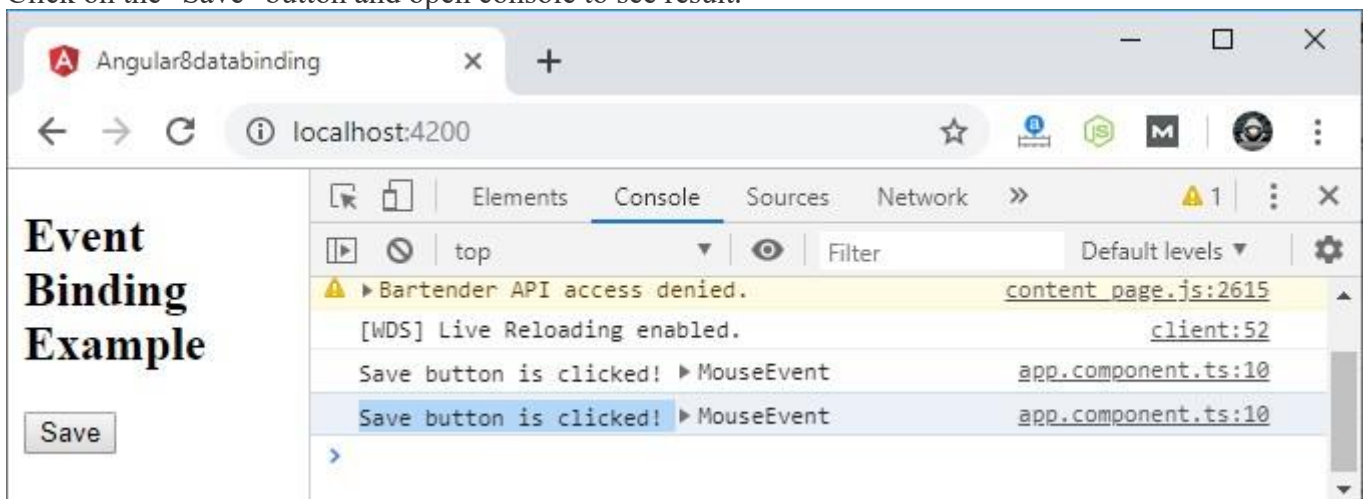
1. `<h2> Event Binding Example</h2>`
2. `<button (click)="onSave($event)">Save</button> <!--Event Binding-->`
- 3.



Output:



Click on the "Save" button and open console to see result.



Now, you can see that the "Save" button is clicked.

#### 4. Class Binding

**Class binding** in Angular makes it very easy to set the class property of a view element. We can set or remove the CSS class names from an element's class attribute with the help of class binding. We bind a class of a DOM element to a field that is a defined property in our Typescript Code. Its syntax is like that of property binding.

Syntax:

```
<element [class] = "typescript_property">
```

##### Approach:

- Define a property element in the app.component.ts file.
- In the app.component.html file, set the class of the HTML element by assigning the property value to the app.component.ts file's element.

**Example 1:** Setting the class element using class binding.

app.component.html

- HTML

```
<h1[class]="geeky"> GeeksforGeeks
```

```
</h1>
```

Upper Heading's class is : "{{ g[0].className }}"

app.component.ts

- Javascript

```
import { Component, OnInit } from '@angular/core';
```

```
@Component({
 selector: 'app-root',
 templateUrl: './app.component.html'
})
export class AppComponent {
 geeky = "GeekClass";

 g = document.getElementsByClassName(this.geeky);
}
```

Output:



Upper Heading's class is : "GeekClass"

## 5. Style Binding

It is very easy to give the CSS styles to HTML elements using style binding in Angular 8. Style binding is used to set a style of a view element. We can set the inline styles of an HTML element using the style binding in angular. You can also add styles conditionally to an element, hence creating a dynamically styled element.

Syntax:

```
<h1 <element [style.style-property] = "'style-value'">
```

**Example 1: app.component.html:**

- HTML

```
[style.color] = "'green'"
```

```
[style.text-align] = "'center'" >
```

GeeksforGeeks

**Output:**

GeeksforGeeks

### b. Two way Data Binding using ngmodel

We have seen that in one-way data binding any change in the template (view) were not be reflected in the component TypeScript code. To resolve this problem, Angular provides two-way data binding. The two-way binding has a feature to update data from component to view and vice-versa.

In two-way databinding, automatic synchronization of data happens between the Model and the View. Here, change is reflected in both components. Whenever you make changes in the Model, it will be reflected in the View and when you make changes in View, it will be reflected in Model.

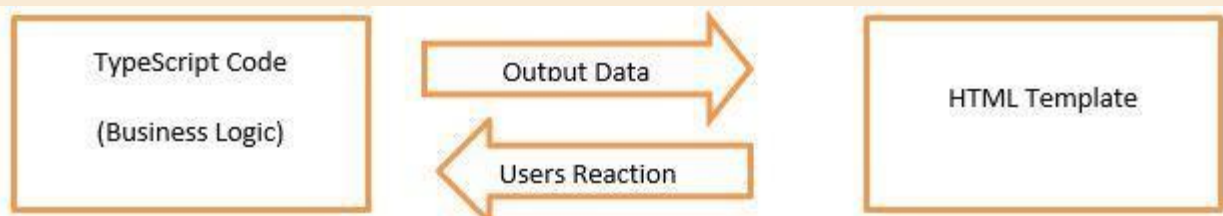
This happens immediately and automatically, ensures that the HTML template and the TypeScript code are updated at all times.

In two way data binding, **property binding and event binding** are combined together.

**Syntax:**

1. [(ngModel)] = "[property of your component]"

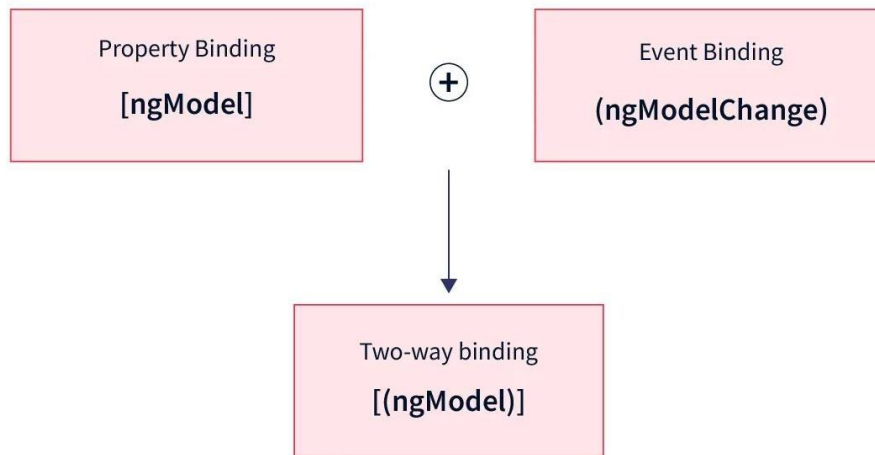
**Note: For two way data binding, we have to enable the ngModel directive. It depends upon FormsModule in angular/forms package, so we have to add FormsModule in imports[] array in the AppModule.**





Let's take an example to understand it better.

**Note:-when you are using ngmodel import FormsModule**



**`[property binding] + (event binding) = [(property)]`**

---

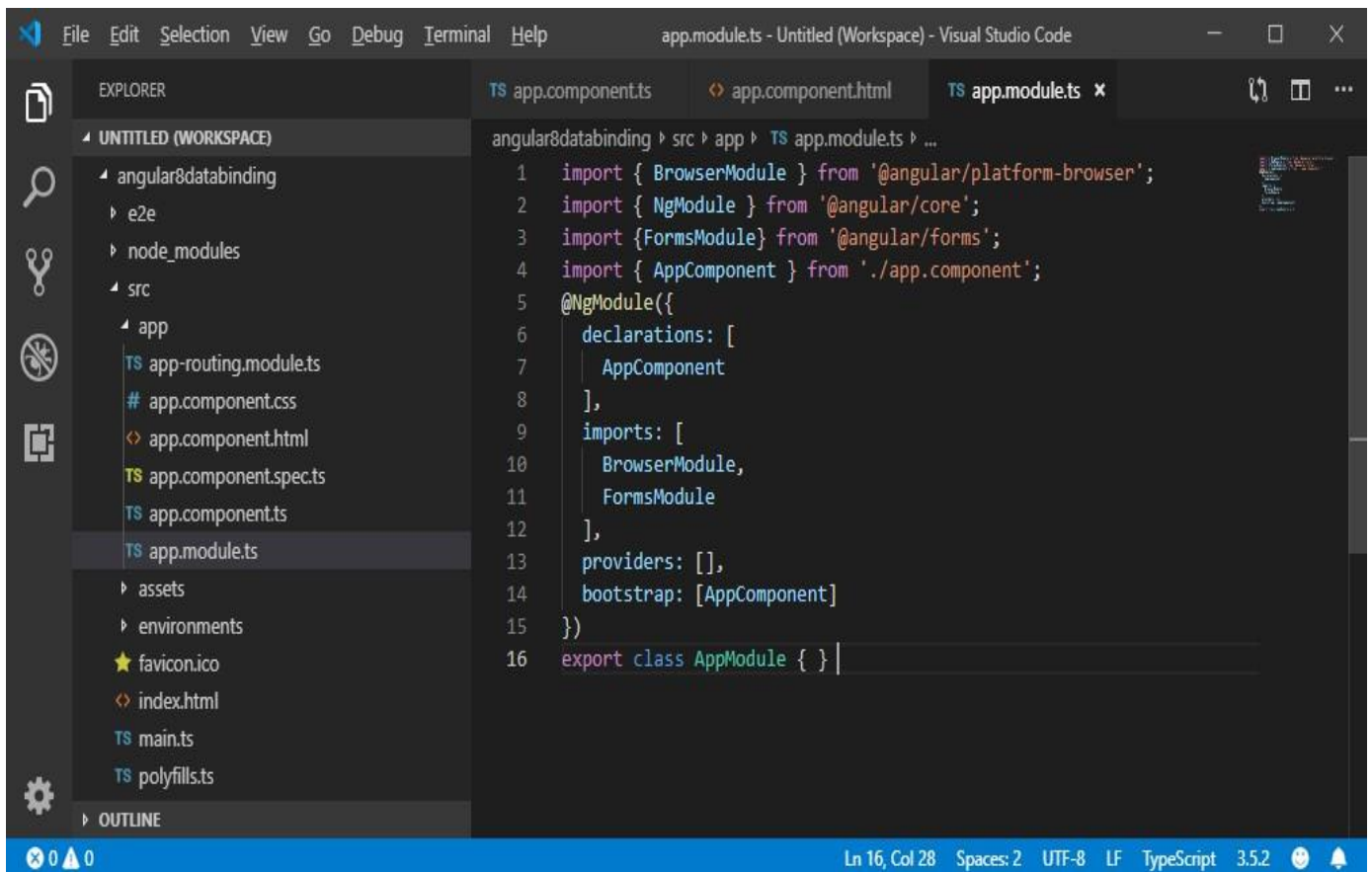
**`[ngModel] + (ngModelChange) = [(ngModel)]`**

---

**`[text] + (textChange) = [(text)]`**

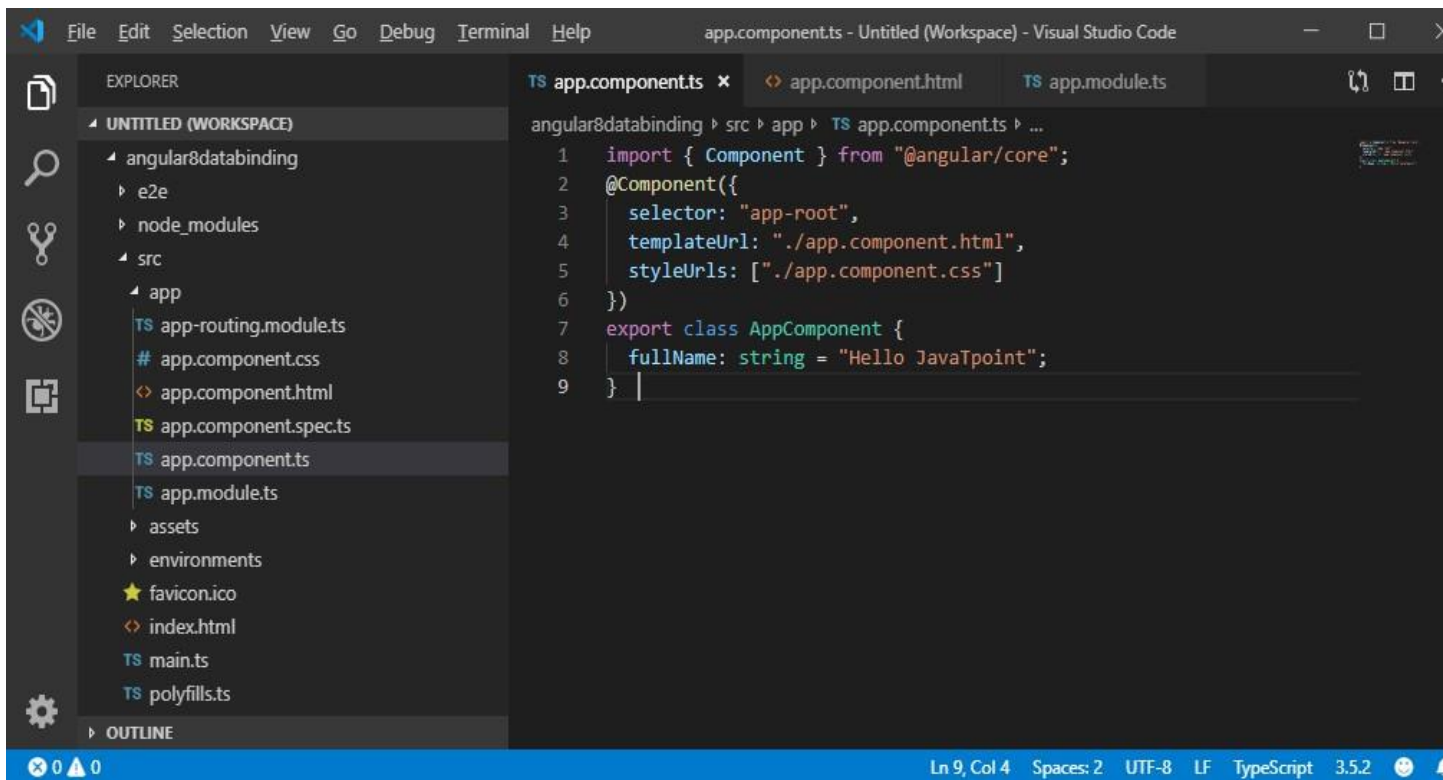
Open your project's **app.module.ts** file and use the following code:

```
1. import { BrowserModule } from '@angular/platform-browser';
2. import { NgModule } from '@angular/core';
3. import { FormsModule } from '@angular/forms';
4. import { AppComponent } from './app.component';
5. @NgModule({
6. declarations: [
7. AppComponent
8.],
9. imports: [
10. BrowserModule,
11. FormsModule
12.],
13. providers: [],
14. bootstrap: [AppComponent]
15. })
16. export class AppModule { }
```



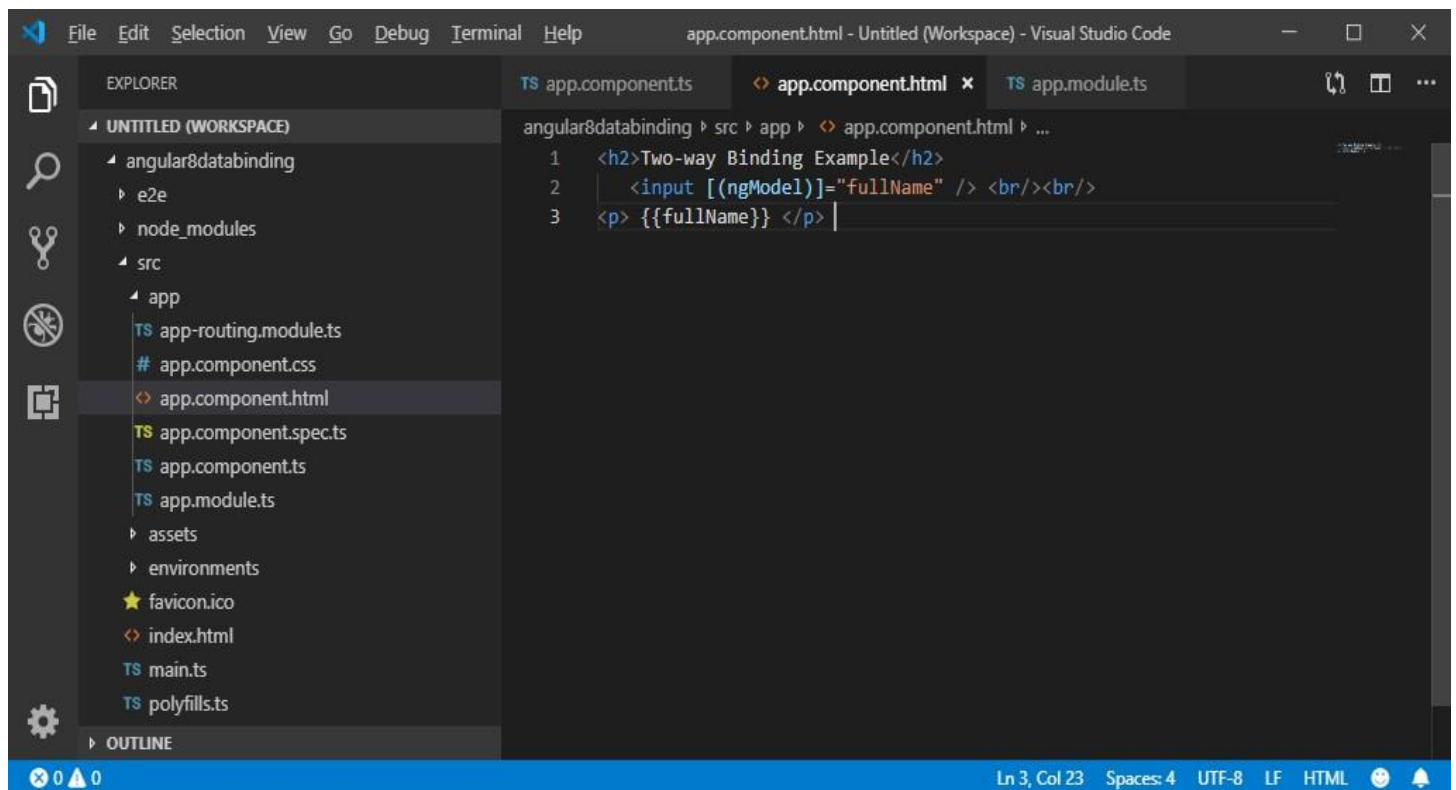
### app.component.ts file:

1. import { Component } from "@angular/core";
2. @Component({
3.   selector: "app-root",
4.   templateUrl: "./app.component.html",
5.   styleUrls: ["./app.component.css"]
6. })
7. export class AppComponent {
8.   fullName: string = "Hello JavaTpoint";
9. }



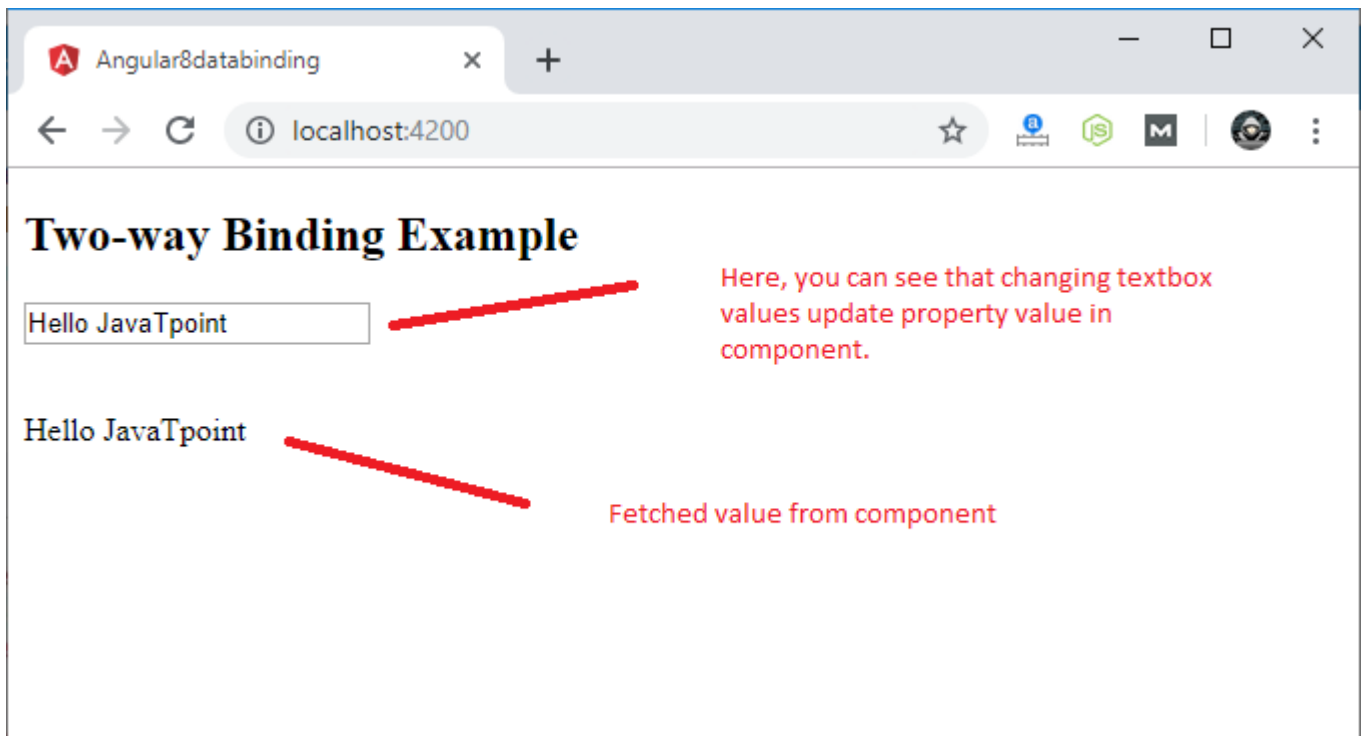
**app.component.html file:**

1. `<h2>Two-way Binding Example</h2>`
2. `<input [(ngModel)]="fullName" /> <br/><br/>`
3. `<p> {{fullName}} </p>`



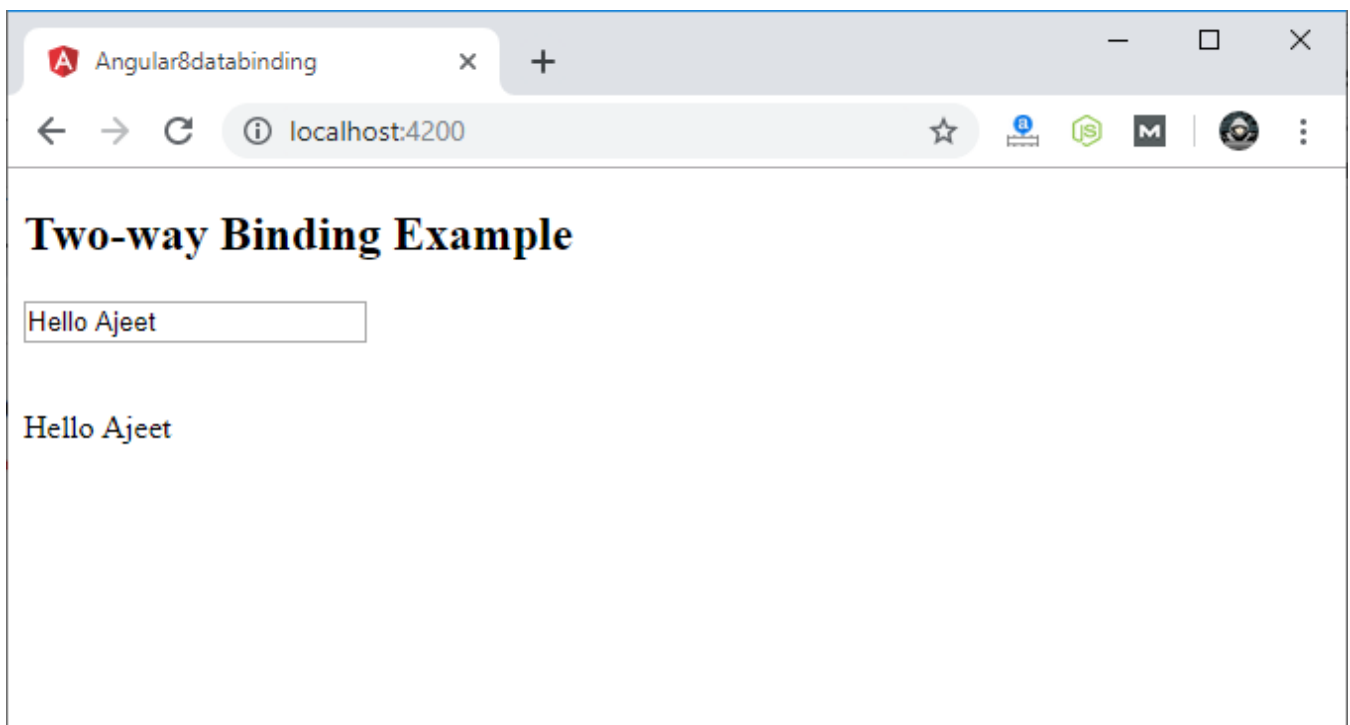
Now, start your server and open local host browser to see the result.

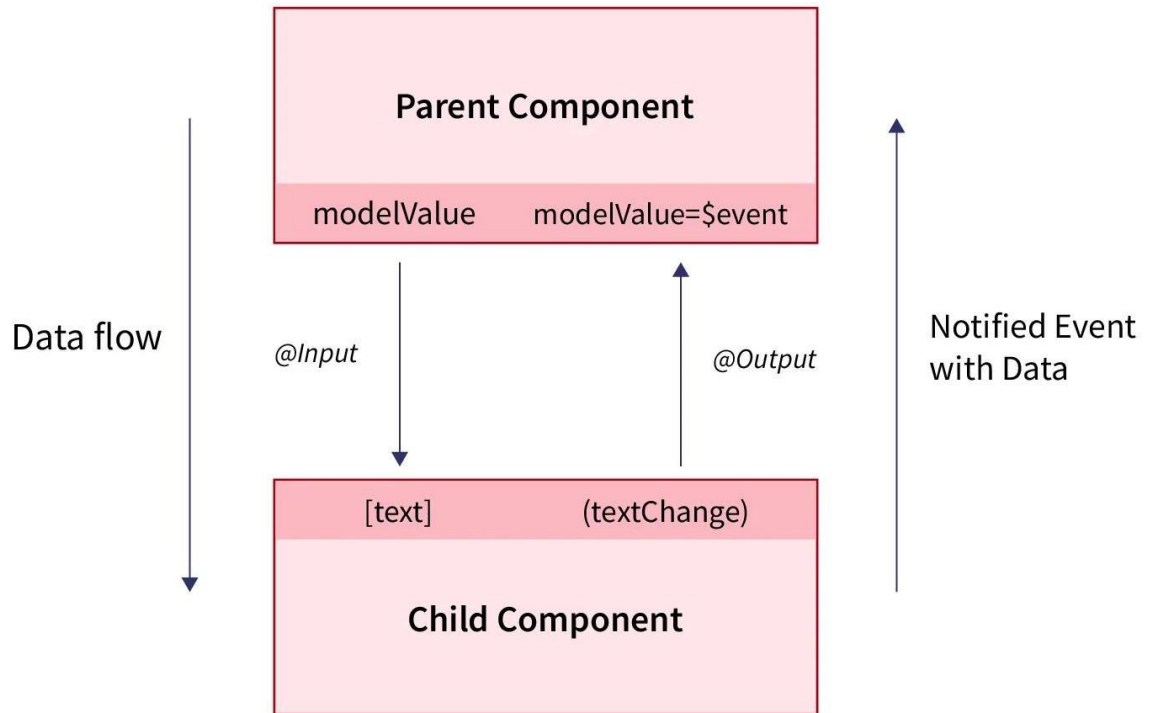
**Output:**



You can check it by changing textbox value and it will be updated in component as well.

**For example:**





(OR)

### Without using ngmodel

**[property binding] + (event binding) = [(property)]**

#### app.component.html

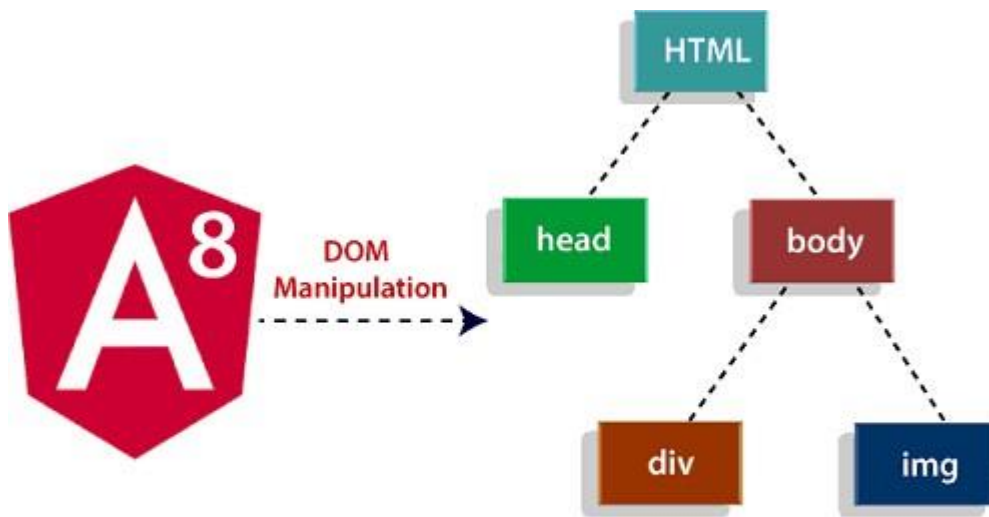
```
<label>User Name</label>
<input type="text" [value]="text" (input)="updateValue
($event)">
<h1>{{text}}</h1>
```

#### app.component.ts

```
fullName: string = "Hello JavaTpoint";
```

### Angular Directives

The Angular 8 directives are used to manipulate the DOM. By using Angular directives, you can change the appearance, behavior or a layout of a DOM element. It also helps you to extend HTML.



## Angular 8 Directive

**Angular 8 directives can be classified in 3 categories based on how they behave:**

- Component Directives
- Structural Directives
- Attribute Directives

**Component Directives:** Component directives are used in main class. They contain the detail of how the component should be processed, instantiated and used at runtime.

**Structural Directives:** Structural directives start with a \* sign. These directives are used to manipulate and change the structure of the DOM elements. For example, \*ngIf directive, \*ngSwitch directive, and \*ngFor directive.

- **\*ngIf Directive:** The ngIf allows us to Add/Remove DOM Element.
- **\*ngSwitch Directive:** The \*ngSwitch allows us to Add/Remove DOM Element. It is similar to switch statement of C#.
- **\*ngFor Directive:** The \*ngFor directive is used to repeat a portion of HTML template once per each item from an iterable list (Collection).

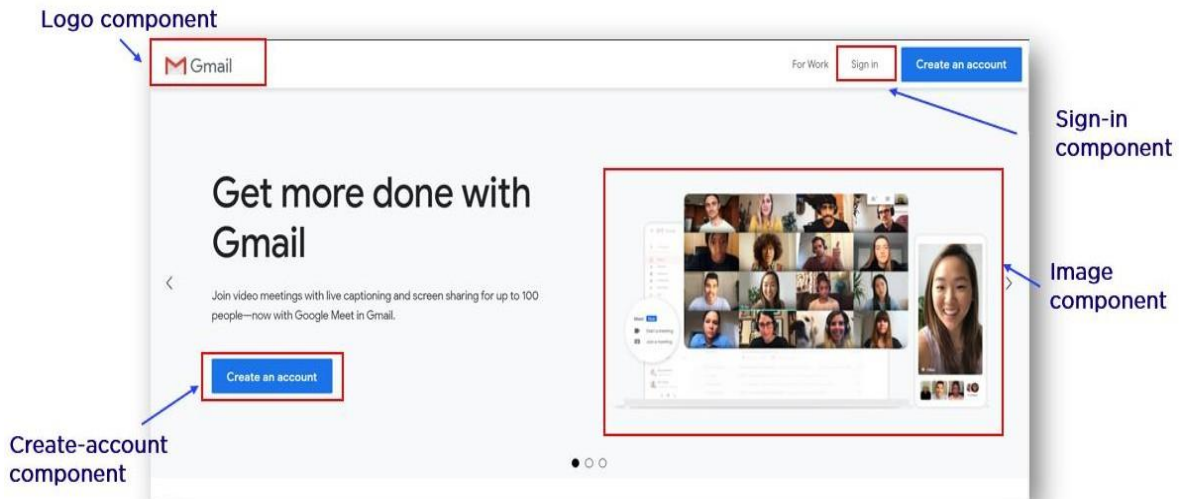
**Attribute Directives:** Attribute directives are used to change the look and behavior of the DOM elements. For example: ngClass directive, and ngStyle directive etc.

- **ngClass Directive:** The ngClass directive is used to add or remove CSS classes to an HTML element.
- **ngStyle Directive:** The ngStyle directive facilitates you to modify the style of an HTML element using the expression. You can also use ngStyle directive to dynamically change the style of your HTML element.

## 5. Fetch Data from a Service

What is the Need for Angular Services?

We're sure you are aware of the concept of components in Angular. The user interface of the application is developed by embedding several components into the main component.



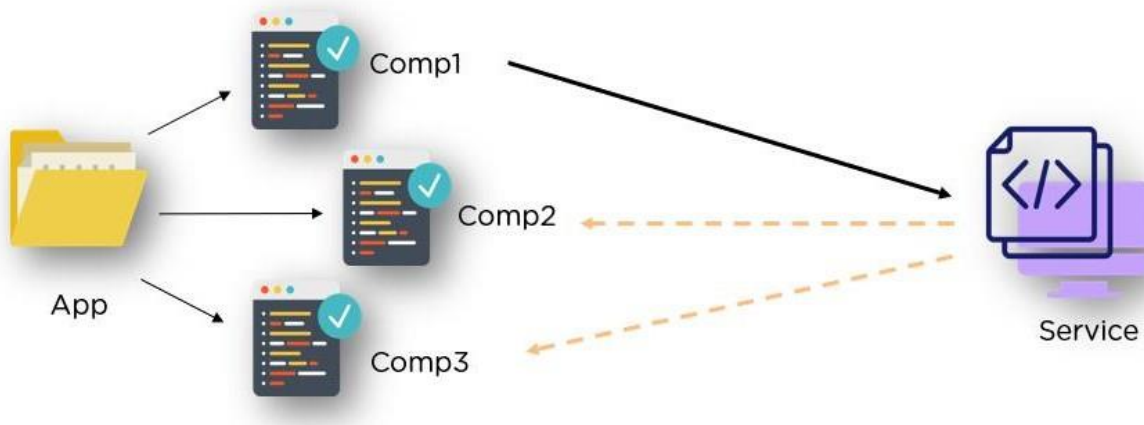
However, these components are generally used only for rendering purposes. They are only used to define what appears on the user interface. Ideally, other tasks, like data and image fetching, network connections, database management, are not performed. Then how are these tasks achieved? And what if more than one component performs similar tasks? Well, Services take care of this. They perform all the operational tasks for the components.

- **Services avoid rewriting of code. A service can be written once and injected into all the components that use that service**
- A service could be a function, variable, or feature that an application needs

### What Are Angular Services?

Angular services are objects that get instantiated just once during the lifetime of an application. They contain methods that maintain data throughout the life of an application, i.e., data is available all the time.

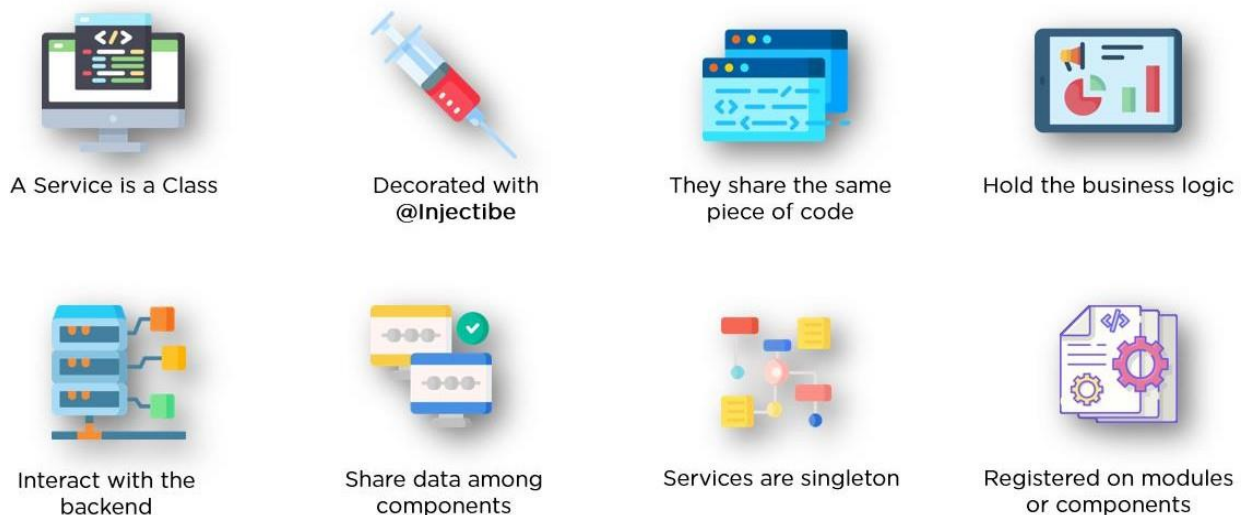




The main objective of a service is to organize and share business logic, models, or data and functions with different components of an Angular application. They are usually implemented through dependency injection.

### Features of Angular Services

- Services in Angular are simply typescript classes with the `@injectable` decorator. This decorator tells angular that the class is a service and can be injected into components that need that service. They can also inject other services as dependencies.
- As mentioned earlier, these services are used to share a single piece of code across multiple components. These services are used to hold business logic.
- Services are used to interact with the backend. For example, if you wish to make AJAX calls, you can have the methods to those calls in the service and use it as a dependency in files.



- In angular, the components are singletons, meaning that only one instance of a service that gets created, and the same instance is used by every building block in the application.
- A service can be registered as a part of the module, or as a part of the component. To register it as a part of the component, you'll have to specify it in the providers' array of the module.

### Fetch data from service Example:-



- Use this command **ng g s service\_name**

### Creating Angular Project Use below Commands

- npm install -g @angular/cli //creating cli
- ng version
- ng new prog9 --standalone false //creating angular project SPA(Single page application)  
//app component is a default component.
- cd prog9
- ng g c header //creating header component
- ng g c home //creating home component
- ng g c profile //creating profile component
- ng serve //running angular project
- Use this command for creating service **ng g s service\_name**

Creating body of about, contact, home and header. Header is a navbar this page creating routerLinks about and contact. Home is a default link when header loaded it is displayed. App.module.ts file creating url paths for each page.

Here test.service.ts file is creating for displaying fruits names you can access service data any component here fetching data service to about.

#### 1)about.component.html:-

```
<h1>This is About Component</h1>
<h3>Which Fruit You Like?</h3>

<div *ngFor="let m of names">
 {{m}}
</div>
```

#### About.component.ts:-

```
import { Component } from '@angular/core';

import { TestService } from '../test.service'; @Component({selector: 'app-about', templateUrl:
'./about.component.html', styleUrls: ['./about.component.css']
})
export class AboutComponent { constructor(private ts:TestService){

}
names=this.ts.names;

}
```

#### 2)contact.component.html:-

```
<h1>This is Contact Component</h1>
```

#### 3)header.component.css:-

```

ul li{
 list-style: none;
}
ul li a{
 text-decoration: none;
}
ul{
 display: flex;
 justify-content: flex-start; gap: 20px;
 background-color: aqua; height: 50px;
}
a{
 line-height: 50px; color: black; margin: 0 20px; font-weight: bold; font-size: 30px;
}

```

#### 4) header.component.html:-

```


 about

 contact


```

#### 5) home.component.html:-

```

<h1>This is Home Component</h1>

```

#### 6) notfound.component.html:-

```

<p>notfound works!</p>

```

#### 7) app.component.html:-

```

<app-header></app-header>
<router-outlet></router-outlet>

```

## 8) app.module.ts:-

```
import { AppComponent } from './app.component';
import { HeaderComponent } from './header/header.component';
import { AboutComponent } from './about/about.component';
import { ContactComponent } from './contact/contact.component';
import { HomeComponent } from './home/home.component';
import { NotfoundComponent } from './notfound/notfound.component';
import { RouterModule, Routes } from '@angular/router';
```

```
const routes:Routes=[
 {
 path:'',component:HomeComponent
 },
 {
 path:'about',component:AboutComponent
 },
 {
 path:'contact',component:ContactComponent
 },
 {
 path:'**',component:NotfoundComponent
 }
]
```

```
imports: [
```

```
 RouterModule.forRoot(routes)
```

```
],
```

## Test.service.ts:-

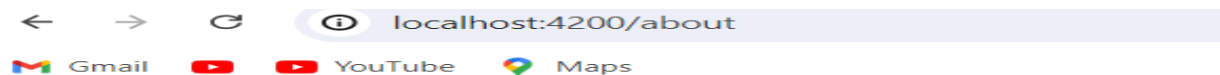
```
import { Injectable } from '@angular/core';
```

```
@Injectable({ providedIn: 'root' })
```

```
export class TestService {
```

```
 constructor() { } names=['Mango','Banana','Watermelon','Apple'];
```

}  
- **Output:**



# This is About Component

## Which Fruit You Like?

Mango  
Banana  
Watermelon  
Apple

### 6. Submit data to service:-

*npm install bootstrap --save*

When Bootstrap is installed open angular.json file and add bootstrap.min.css file reference under "styles":

```
1. "styles": [
2. "src/styles.css",
3. "node_modules/bootstrap/dist/css/bootstrap.min.css"
4.]
```

Now we need to create components and service. Use the following commands to create the same.

*ng g c header ng g c reg*

### Note

g stands for generate | c stands for Component | s stands for Service

Open app.modules.ts file and add these lines:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { RegComponent } from './reg/reg.component';
import { HeaderComponent } from './header/header.component';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { RouterModule, Routes } from '@angular/router';
```

```
const routes: Routes = [
```

```

- { path: "reg", component: RegComponent }
];

imports: [BrowserModule, AppRoutingModule, FormsModule, ReactiveFormsModule,
 RouterModule.forRoot(routes)
]

```

In app.component.html replace the existing code with the below code:

```

1. <app-header></app-header>
2. <router-outlet></router-outlet>

```

Let's start with components now.

Open header.component.html file and replace with the code below.

```


 create


```

Open header.component.css file and replace with the code below.

```

ul{
 background-color:aqua;
}

```

Now let's create a function in Service.

Open data.service.ts file and replace with the code below.

```

1. public SaveEmployee(empdata) {
2. console.log("Full Name : " + empdata.regFullName);
3. console.log("Email Id : " + empdata.regEmail);
4. }

```

Open reg.component.html file and replace with the code below.

```

1. <div class="container" style="margin-top: 150px;">
2. <form [formGroup]="frmRegister" (ngSubmit)="SaveEmployee(frmRegister.valu
 e)">
3. <div class="panel panel-primary">
4. <div class="panel-heading">
5. <h3 class="panel-title">Employee Registration</h3>
6. </div>
7. <div class="panel-body">
8. <div class="form-group">
9. <label for="fullName">Full Name</label>
10. <input id="fullName" formControlName="regFullName" type=
 "text" class="form-control" required />
11. </div>
12. <div class="form-group">

```

```

13. <label for="email">Email</label>
14. <input id="email" formControlName="regEmail" type="email
" class="form-control" required />
15. </div>
16. </div>
17. <div class="panel-footer">
18. <button type="submit" class="btn btn-
primary">Save</button>
19. </div>
20. </div>
21. </form>
22. </div>

```

Open reg.component.ts file and replace with the code below.

```

1. import {
2. Component,
3. OnInit
4. } from '@angular/core';
 import {
5. FormGroup,
6. FormBuilder
7. } from '@angular/forms';
8. import {
9. DataService
10. } from '../data.service';
11. @Component({
12. selector: 'app-reg',
13. templateUrl: './reg.component.html',
14. styleUrls: ['./reg.component.css']
16. })
17. export class RegComponent implements OnInit {
18. frmRegister: FormGroup;
19. constructor(private _fb: FormBuilder, private dataservice: DataService)
 {}
20. ngOnInit(): void {
21. this.frmRegister = this._fb.group({
22. regFullName: '',
23. regEmail: ''
24. });
25. }
26. SaveEmployee(value) {
27. this.dataservice.SaveEmployee(value);
28. }
29. }

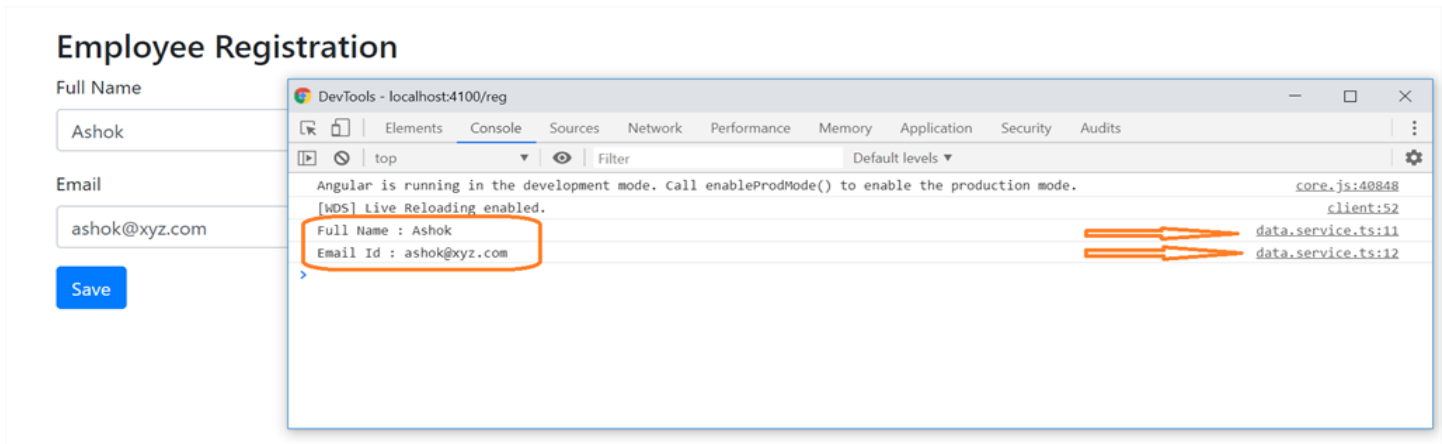
```

Now build your application by ng build. Run application by **ng serve**.

**Output:-**

## 7. Http Module:-

### Defination:-



**\$http** is an AngularJS service for reading data from remote servers. Implements an HTTP client API for Angular apps that relies on the XMLHttpRequest interface exposed by browsers. Includes testability features, typed request and response objects, request and response interception, observable APIs, and streamlined error handling.

These components are self-sufficient and can be used on their own without being tied to a specific **NgModule**. But, sometimes, when you're working with these standalone components, you might need to fetch data from servers or interact with APIs using HTTP requests.

We need to import the http module to make use of the http service. Let us consider an example to understand how to make use of the http service.

### Example1:-Fetching data from API and displayed console

To start using the http service, we need to import the module in **app.module.ts** as shown below –

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { BrowserModuleAnimationsModule } from '@angular/platform-browser/animations';
import { HttpClientModule } from '@angular/common/http';
import { AppComponent } from './app.component';
@NgModule({
 declarations: [
 AppComponent
],
 imports: [
 BrowserModule,
 BrowserModuleAnimationsModule,
 HttpClientModule
],
 providers: [],
 bootstrap: [AppComponent]
})
export class AppModule { }
```

-

If you see the highlighted code, we have imported the `HttpClientModule` from `@angular/common/http` and the same is also added in the imports array.

Let us now use the http client in the **app.component.ts**.

```
import { Component } from '@angular/core';
import { HttpClient } from
'@angular/common/http'; @Component({
 selector: 'app-root',
 templateUrl:
'./app.component.html',
 styleUrls:
['./app.component.css']
})
export class AppComponent {
 constructor(private http:
HttpClient) {} ngOnInit() {
 this.http.get("http://jsonplaceholder.typicode.com/users").
 subscribe((data) => console.log(data))
 }
}
```

Let us understand the code highlighted above. We need to import http to make use of the service, which is done as follows –

```
import { HttpClient } from '@angular/common/http';
```

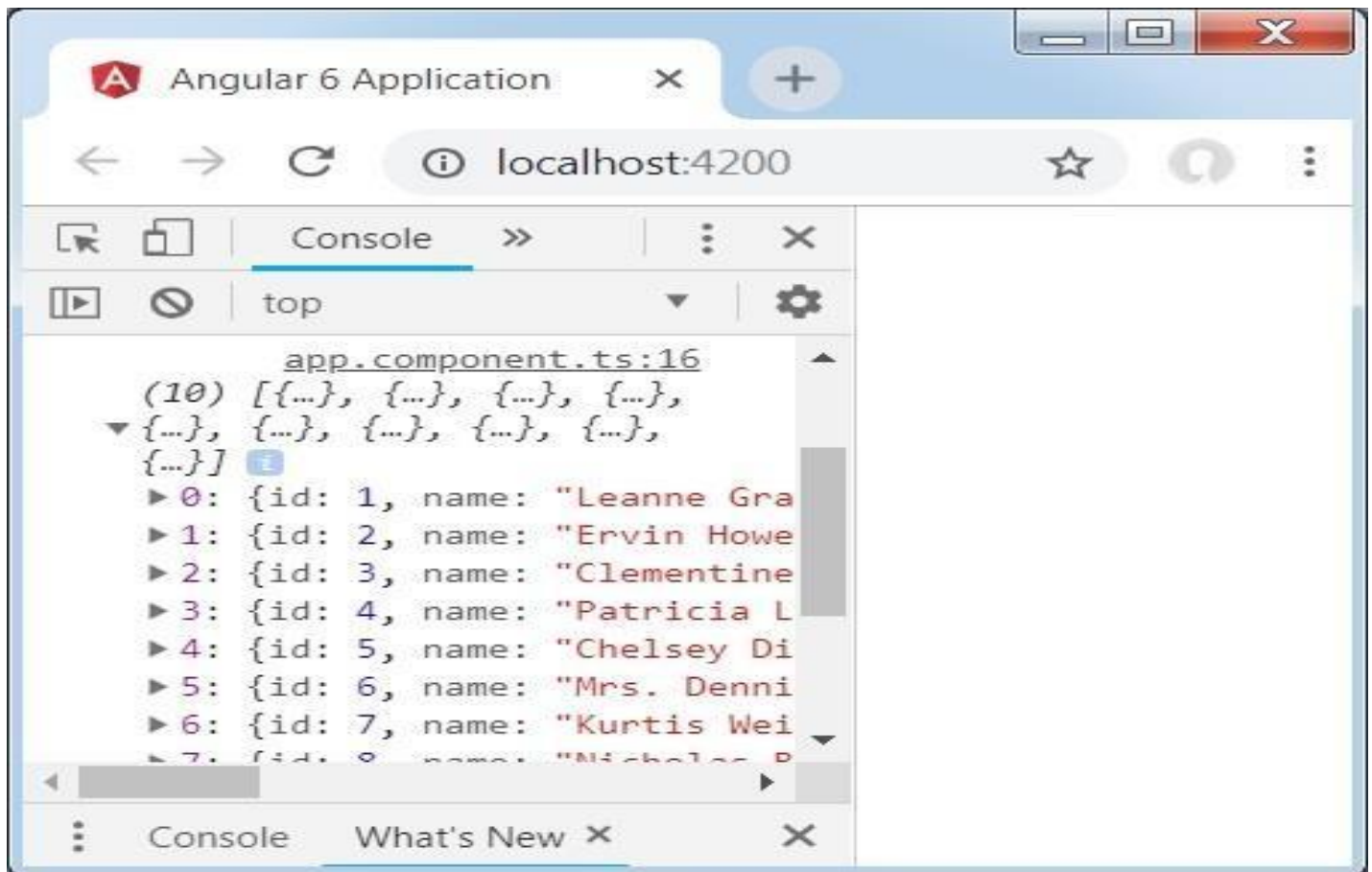
In the class **AppComponent**, a constructor is created and the private variable `http` of type `Http`. To fetch the data, we need to use the **get API** available with `http` as follows

```
this.http.get();
```

It takes the url to be fetched as the parameter as shown in the code.



- We will use the test url – <https://jsonplaceholder.typicode.com/users> to fetch the json data. The subscribe will log the output in the console as shown in the browser –



If you see, the json objects are displayed in the console. The objects can be displayed in the browser too.

Example2:-

For the objects to be displayed in the browser, update the codes in **app.component.html** and **app.component.ts** as follows –

```
import { Component } from '@angular/core';
import { HttpClient } from '@angular/common/http';
```

```

@Component({
 selector: 'app-root',
 templateUrl:
 './app.component.html',
 styleUrls:
 ['./app.component.css']
})
export class AppComponent {
 constructor(private http: HttpClient) {}
 httpdata;
 ngOnInit() {
 this.http.get("http://jsonplaceholder.typicode.com/users")
 .subscribe((data) => this.displaydata(data));
 }
}

```

In **app.component.ts**, using the subscribe method we will call the display data method and pass the data fetched as the parameter to it.

In the display data method, we will store the data in a variable httpdata. The data is displayed in the browser using **for** over this httpdata variable, which is done in the **app.component.html** file.

```

<ul *ngFor = "let data of httpdata">
 Name : {{data.name}} Address: {{data.address.city}}


```

The json object is as follows –

```

{
 "id": 1,
 "name": "Leanne Graham", "username": "Bret",
 "email": "Sincere@april.biz",

 "address": {
 "street": "Kulas Light",
 "suite": "Apt. 556",
 "city": "Gwenborough",
 "zipcode": "92998-3874",
 "geo": {
 "lat": "-37.3159",
 "lng": "81.1496"
 }
 },

 "phone": "1-770-736-8031 x56442",

```

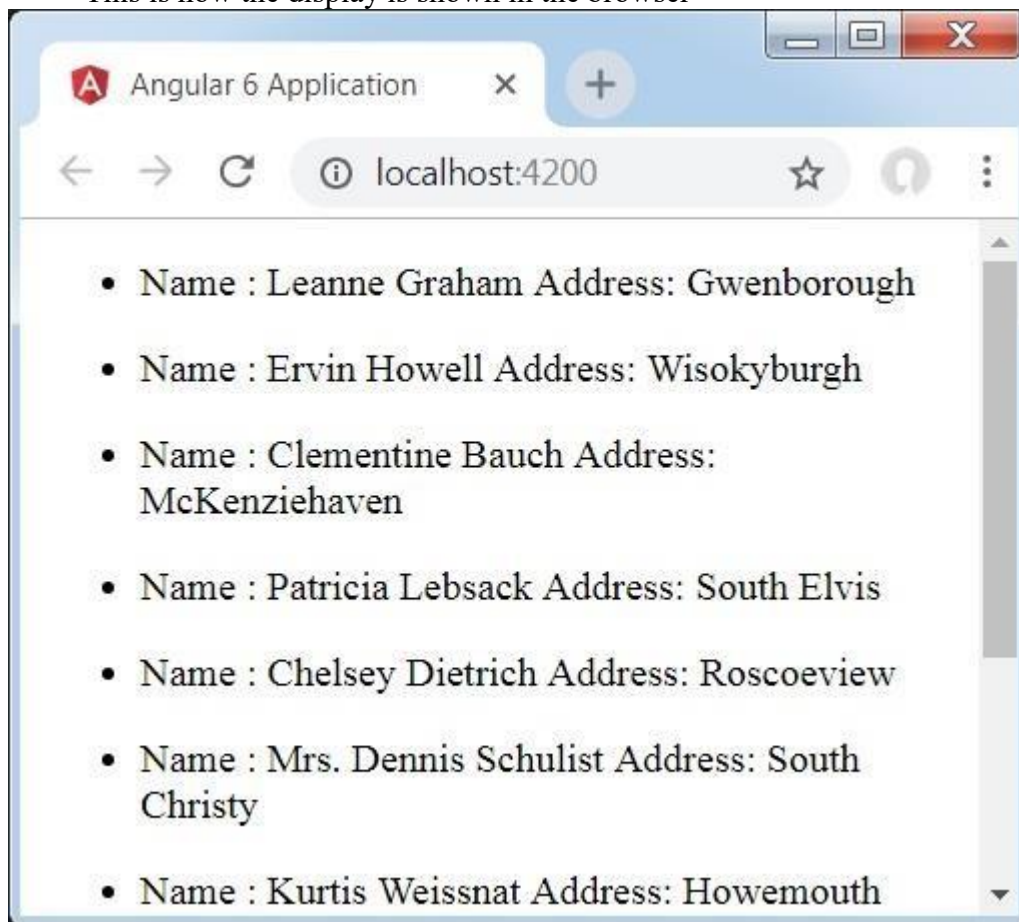
```

- "website": "hildegard.org",
 "company": {
 "name": "Romaguera-Crona",
 "catchPhrase": "Multi-layered client-
server neural-net", "bs": "harness real-time
e-markets"
 }
}

```

The object has properties such as id, name, username, email, and address that internally has street, city, etc. and other details related to phone, website, and company. Using the **for** loop, we will display the name and the city details in the browser as shown in the **app.component.html** file.

This is how the display is shown in the browser –



Let us now add the search parameter, which will filter based on specific data.

Example 3:-

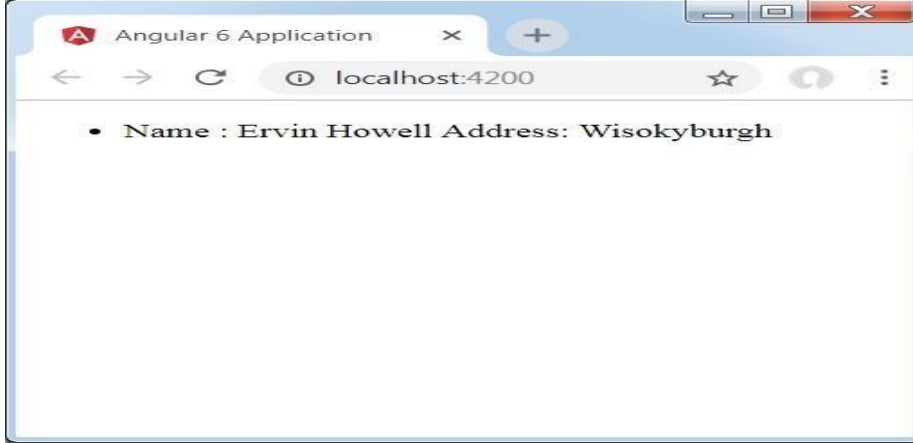
We need to fetch the data based on the search param passed.

Following are the changes done in **app.component.html** and

```

import { Component } from '@angular/core';
import { HttpClient } from '@angular/common/http';
@Component({
 selector: 'app-root',
 templateUrl:
 './app.component.html',
 styleUrls:

```



**app.component.ts** files – app.component.ts

```
export class AppComponent {
 constructor(private http:
 HttpClient) {} httpdata;
 nam
 e;
 sear
 chpa
 ram
 = 2;
```

For the **get api**, we will add the search param `id = this.searchparam`. The searchparam is equal to 2. We need the details of **id = 2** from the json file.

This is how the browser is displayed –

We have console'd the data in the browser, which is received from the http. The same is displayed in the browser console. The name from the json with **id = 2** is displayed in the browser.

#### Example4:-

##### Creating Angular Project Use below Commands

- `npm install -g @angular/cli //creating cli`
- `ng version`
- `ng new prog9 //creating angular project SPA(Single page application) //app component is a default component.`
- `cd prog9`
- `ng g c header //creating header component`
- `ng g c home //creating home component`
- `ng g c profile //creating profile component`
- `ng serve //running angular project`

#### Step1: header.component.css

```
ul li{
 list-style: none;
}
```

```

ul li a{
 text-decoration: none;
}
ul{
 background-color: aqua; height: 50px;
}
a{
 line-height:50px ; font-weight: bold; font-size:20px;
}

```

**Step 2:-** Create navbar in **header.component.html**

```


 Profile


```

**Step 3:-** Create navbar in **home.component.html**

```

<h1>Welcome to Home Page</h1>

```

**Step 4:-**Configure route links in **app.module.ts**

```

import { ProfileComponent } from './profile/profile.component';
import { HeaderComponent } from './header/header.component';
import { HomeComponent } from './home/home.component';
import { RouterModule,Routes } from '@angular/router';
import { HttpClientModule } from '@angular/common/http'; const
routes:Routes=[
 {
 path:"",component:HomeComponent
 },
 {
 path:'profile',component:ProfileComponent
 },

```

\_ ]

```
imports: [RouterModule.forRoot(routes), HttpClientModule]
```

**Step 5:-**Use header selector in the **app.component.html** along with <router-outlet>

```
<app-header></app-header>
<router-outlet></router-outlet>
```

#### Step6:Profile.component.css

```
img {
 border-radius: 50%;
}
```

#### Step7:Profile.component.html

```
<h1>Welcome to profile page</h1>

<button (click)="getData()">Get Profile</button>

<div *ngIf="data">

<table>
 <tr><th>ID</th>
 <td>{{data.id}}</td></tr>
 <tr><th>Name</th>
 <td>{{data.name}}</td></tr>
 <tr><th>Email</th>
 <td>{{data.email}}</td></tr>
 <tr><th>Phone</th>
 <td>{{data.phone}}</td></tr>
</table>
</div>
```

#### Step8:Profile.component.ts

```
import { Component } from '@angular/core';
import { HttpClient } from '@angular/common/http';
@Component({ selector: 'app-profile',
 templateUrl:
```

```

- './profile.component.
html', styleUrls:
'./profile.component.
css'
}))

export class ProfileComponent {
 imagePath:any;
 constructor(private http:HttpClient){

 this.imagePath = 'https://static.javatpoint.com/tutorial/angular7/images/angular-7-logo.png';
 }
 data:any;

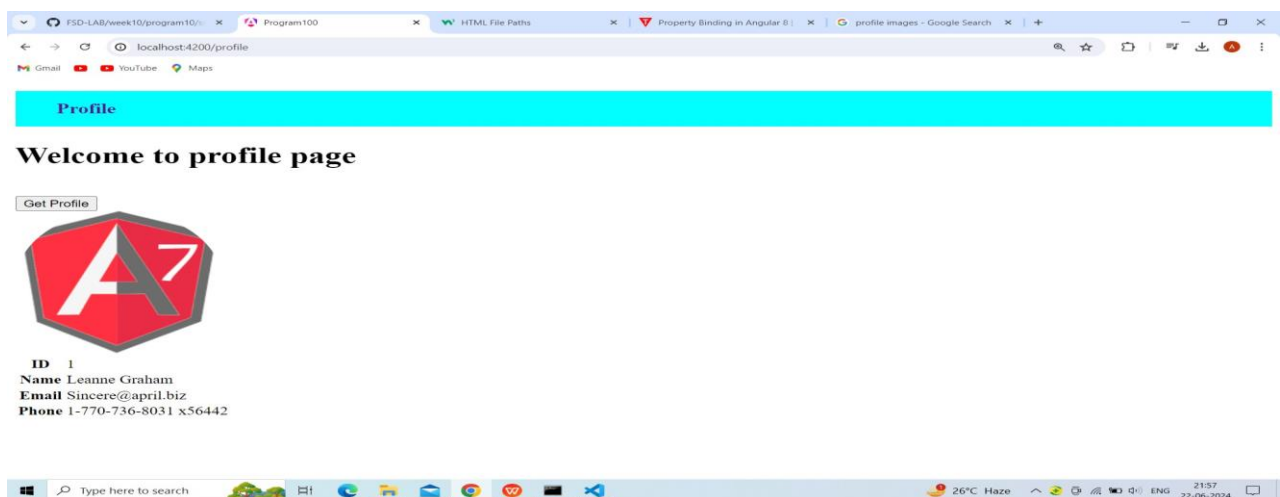
 getData(){

 this.http.get('https://jsonplaceholder.typicode.com/users/1')
 .subscribe((data)=>{ this.data=data;

 })
 }
}

```

## Output:



## React JS

Need of React, Simple React Structure, The Virtual DOM, React Components, Introducing React

# Components, Creating Components in React, Data and Data Flow in React, Rendering and Life Cycle Methods in React, Working with forms in React, integrating third party libraries, Routing in React.

Introduction:-

## 1. Defination:-

ReactJS is a declarative, efficient, and flexible JavaScript library for building reusable UI components. It is an open-source, component-based front end library responsible only for the view layer of the application. It was created by **Jordan Walke**, who was a software engineer at **Facebook**. It was initially developed and maintained by Facebook and was later used in its products like **WhatsApp & Instagram**. Facebook developed ReactJS in **2011** in its newsfeed section, but it was released to the public in the month of **May 2013**.

Today, most of the websites are built using MVC (model view controller) architecture. In MVC architecture, React is the 'V' which stands for view, whereas the architecture is provided by the Redux or Flux.

- A ReactJS application is made up of multiple components, each component responsible for outputting a small, reusable piece of HTML code. The components are the heart of all React applications. These Components can be nested with other components to allow complex applications to be built of simple building blocks. ReactJS uses virtual DOM based mechanism to fill data in HTML DOM. The virtual DOM works fast as it only changes individual DOM elements instead of reloading complete DOM every time.
- To create React app, we write React components that correspond to various elements. We organize these components inside higher level components which define the application structure. For example, we take a form that consists of many elements like input fields, labels, or buttons. We can write each element of the form as React components, and then we combine it into a higher-level component, i.e., the form component itself. The form components would specify the structure of the form along with elements inside of it.

## 2. why learn ReactJS?

- Today, many JavaScript frameworks are available in the market(like angular, node), but still, React came into the market and gained popularity amongst them. The previous frameworks follow the traditional data flow structure, which uses the DOM (Document Object Model). DOM is an object which is created by the browser each time a web page is loaded. It dynamically adds or removes the data at the back end and when any modifications were done, then each time a new DOM is created for the same page. This repeated creation of DOM makes unnecessary memory wastage and reduces the performance of the application.
- Therefore, a new technology ReactJS framework invented which remove this drawback. ReactJS allows you to divide your entire application into various components. ReactJS still used the same traditional data flow, but it is not directly operating on the browser's Document Object Model (DOM) immediately; instead, it operates on a virtual DOM. It means rather than manipulating the document in a browser after changes to our data, it resolves changes on a DOM built and run entirely in memory. After the virtual DOM has been updated, React determines what changes made to the actual browser's DOM. The React Virtual DOM exists entirely in memory and is a representation of the web browser's DOM. Due to this, when we write a React component, we did not write directly to the DOM; instead, we are writing virtual components that react will turn into the DOM.



### 3.React.JS History

Current version of React.JS is V18.0.0 (April 2022). Initial Release to the Public (V0.3.0) was in July 2013.

React.JS was first used in 2011 for Facebook's Newsfeed feature. Facebook Software Engineer, Jordan Walke, created it. Current version of create-react-app is v5.0.1 (April 2022). create-react-app includes built tools such as webpack, Babel, and ESLint.

### 4. Features of React

React offers some outstanding features that make it the most widely adopted library for frontend app development. Here is the list of those salient features.

- **JSX**

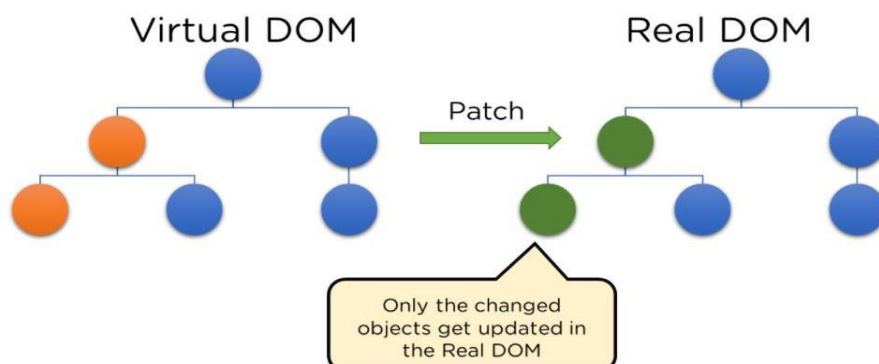


JSX is a JavaScript syntactic extension. It's a term used in React to describe how the user interface should seem. You can write HTML structures in the same file as JavaScript code by utilizing JSX.

```
const name = 'Simplilearn';
const greet = <h1>Hello, {name}</h1>;
```

The above code shows how JSX is implemented in React. It is neither a string nor HTML. Instead, it embeds HTML into JavaScript code.

### Virtual Document Object Model (DOM)



- The Virtual DOM is React's lightweight version of the Real DOM. Real DOM manipulation is substantially slower than virtual DOM manipulation. When an object's state changes, Virtual DOM updates only that object in the real DOM rather than all of them.

- What is the Document Object Model (DOM)?

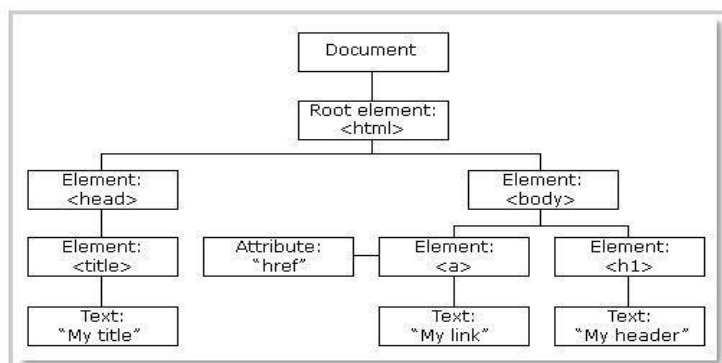


Fig: DOM of a Webpage

DOM (Document Object Model) treats an XML or HTML document as a tree structure in which each node is an object representing a part of the document.

- How do Virtual DOM and React DOM interact with each other?

When the state of an object changes in a React application, VDOM gets updated. It then compares its previous state and then updates only those objects in the real DOM instead of updating all of the objects. This makes things move fast, especially when compared to other front-end technologies that have to update each object even if only a single object changes in the web application.

- Architecture

In a Model View Controller(MVC) architecture, React is the 'View' responsible for how the app looks and feels.

MVC is an architectural pattern that splits the application layer into Model, View, and Controller. The model relates to all data-related logic; the view is used for the UI logic of the application, and the controller is an interface between the Model and View.

- Extensions



- React goes beyond just being a UI framework; it contains many extensions that cover the entire application architecture. It helps the building of mobile apps and provides server-side rendering. Flux and Redux, among other things, can extend React.

- Data Binding

Since React employs one-way data binding, all activities stay modular and quick. Moreover, the unidirectional data flow means that it's common to nest child components within parent components when developing a React project.



Fig: One-way data binding

- Debugging

Since a broad developer community exists, React applications are straightforward and easy to test. Facebook provides a browser extension that simplifies and expedites React debugging.

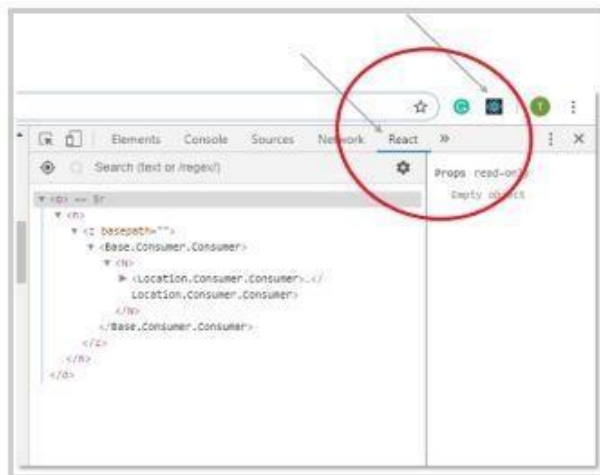
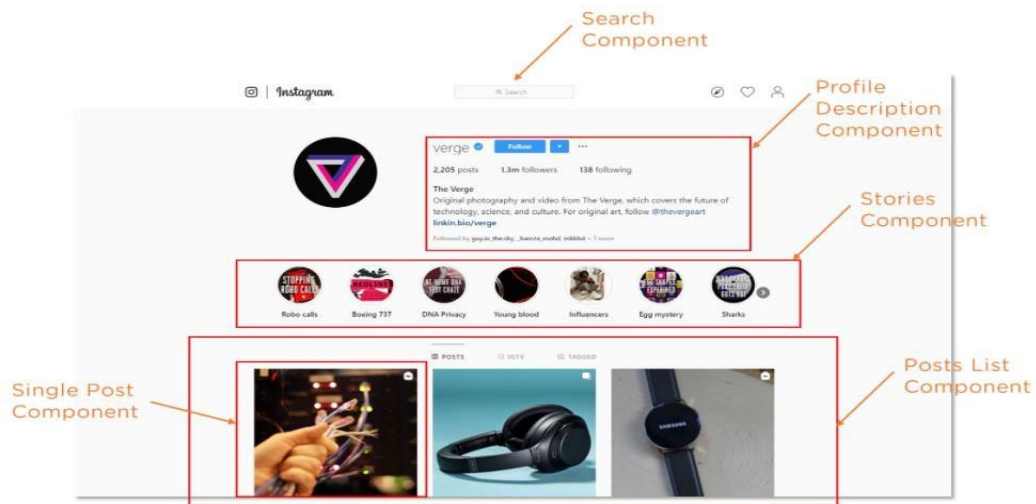


Fig: React Extension

This extension, for example, adds a React tab in the developer tools option within the Chrome web browser. The tab makes it easy to inspect React components directly.

- Components in React

Components are the building blocks that comprise a React application representing a part of the user interface.



React separates the user interface into numerous components, making debugging more accessible, and each component has its own set of properties and functions.

- **Single-Page Applications (SPAs)**

React is recommended in creating SPAs, allowing smooth content updates without page reloads. Its focus on reusable components makes it ideal for real-time applications.

## Install React JS

### How To Install React on Windows

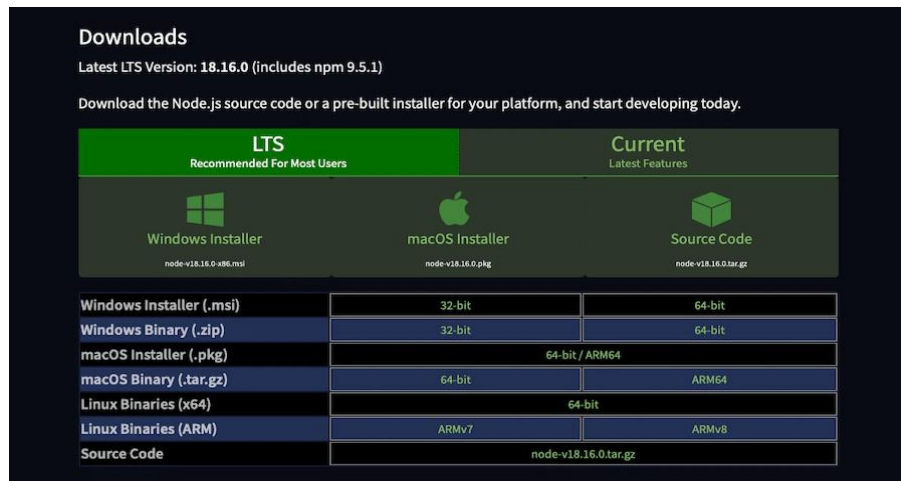
In this section, we'll guide you through the process of installing React on a Windows machine. Follow these steps to get started:

1. [Step 1: Install Node.js and npm](#)
2. [Step 2: Install Create React App](#)
3. [Step 3: Create a New React Project](#)
4. [Step 4: Go To the Project Directory and Start the Development Server](#)

## Step 1: Install Node.js and npm

Before installing React, you need to have Node.js and npm (Node Package Manager) installed on your system. If you haven't already installed them, follow these steps:

1. Visit the Node.js download page at: <https://nodejs.org/en/download/>
2. Download the installer for your Windows system (either the LTS or Current version is fine, but the LTS version is recommended for most users)
3. To install Node.js and npm, please run the installer and carefully follow the provided



After the installation is complete, you can verify that Node.js and npm are installed by opening a command prompt and running the following commands:

Downloading the Node.js installer for Windows.

- `node -v`
- `npm -v`

These commands should display the version numbers for Node.js and npm, respectively.

## Step 2: Install Create React App

Create React App is a command-line tool that simplifies the process of setting up a new React project with a recommended project structure and configuration. To install Create React App globally, open a command prompt and run the following command:

- `npm install -g create-react-app`

This command installs Create React App on your system, making it available to use in any directory.

## Step 3: Create a New React Project

Now that you have Create React App installed, you can use it to create a new React project. To do this, open a command prompt, go to the directory where you want the project to live, and run the

- `create-react-app my-app`

Replace “my-app” with the desired name for your project. Create React App will create a new directory with the specified name and generate a new React project with a recommended project structure and configuration.

#### Step 4: Go To the Project Directory and Start the Development Server

Once the project is created, head over to the project directory by running the following

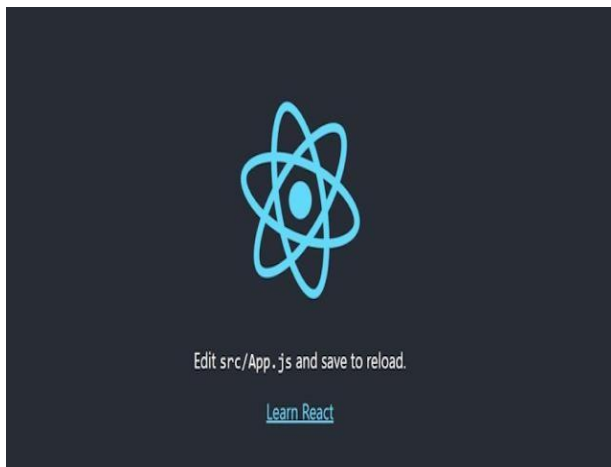
```
cd my-app
```

Replace “my-app” with the name of your project directory. Now, start the development server by running the following command:

```
npm start
```

This command launches the development server, which watches for changes to your project files and automatically reloads the browser when changes are detected.

A new browser window should open with your React application running at `http://localhost:3000/` that looks like this:



React has been successfully installed on Windows.

Congratulations! You have successfully installed React on your Windows machine and created a new React project. You can now begin building your user interfaces with React.

React create-react-app

Starting a new React project is very complicated, with so many build tools. It uses many dependencies, configuration files, and other requirements such as Babel, Webpack, ESLint before writing a single line of React code. Create React App CLI tool removes all that complexities and makes React app simple. For this, you need to install the package using NPM, and then run a few simple commands to get a new React project.

The **create-react-app** is an excellent tool for beginners, which allows you to create and run React

- project very quickly. It does not take any configuration manually. This tool is wrapping all of the required dependencies like **Webpack**, **Babel** for React project itself and then you need to focus on writing React code only. This tool sets up the development environment, provides an excellent developer experience, and optimizes the app for production.

Create React App is a command-line tool that simplifies the process of setting up a new React project with a recommended project structure and configuration.

### Requirements

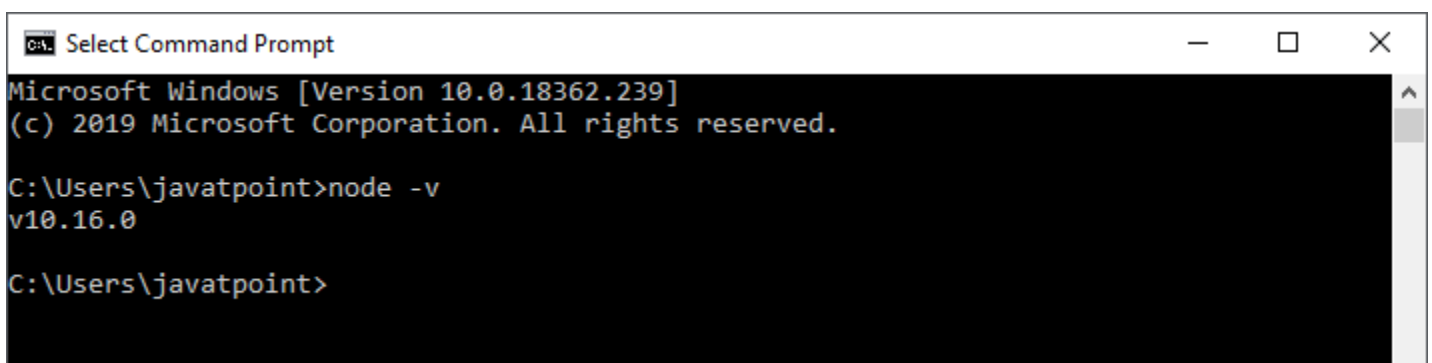
The Create React App is maintained by **Facebook** and can work on any **platform**, for example, macOS, Windows, Linux, etc. To create a React Project using create-react-app, you need to have installed the following things in your system.

1. Node version
2. NPM version

Let us check the current version of **Node** and **NPM** in the system.

Run the following command to check the Node version in the command prompt.

1. \$ node -v



```
Microsoft Windows [Version 10.0.18362.239]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\javatpoint>node -v
v10.16.0

C:\Users\javatpoint>
```

Run the following command to check the NPM version in the command prompt.

1. \$ npm -v



```
C:\Users\javatpoint>npm -v
6.9.0

C:\Users\javatpoint>
```

### Install React

We can install React using npm package manager by using the following command. There is no need to worry about the complexity of React installation. The create-react-app npm package

manager will manage everything, which needed for React project.

```
C:\Users\javatpoint> npm install -g create-react-app
```

**Note:**-If you've previously installed create-react-app globally, it is recommended that you uninstall the package to ensure npx always uses the latest version of create-react-app.

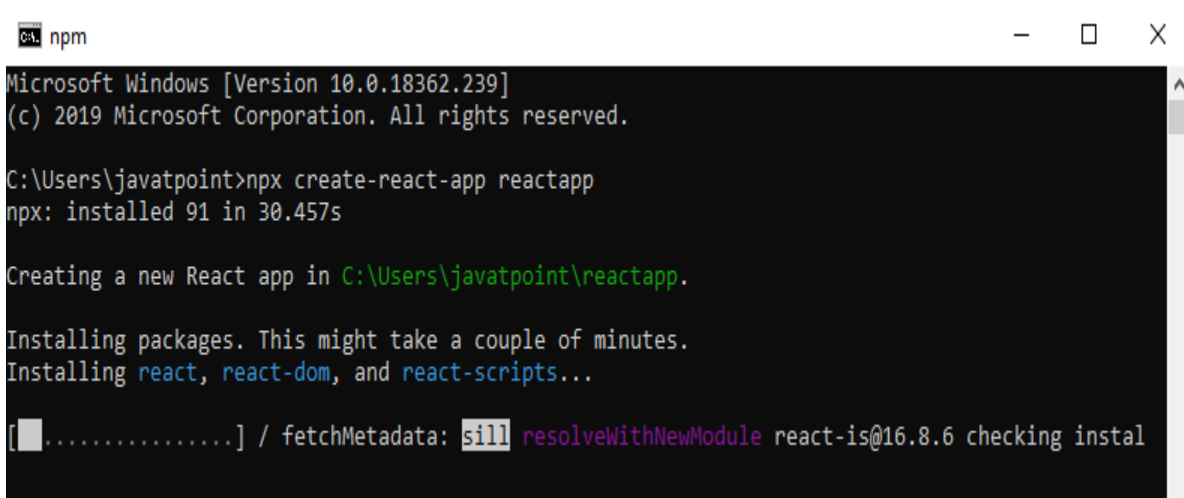
Create a new React project

Once the React installation is successful, we can create a new React project using create-react-app command. Here, I choose "reactproject" name for my project.

```
C:\Users\javatpoint> create-react-app reactproject
```

**NOTE:** *We can combine the above two steps in a single command using npx. The npx(Node Package eXecute) is a package runner tool which comes with npm 5.2 and above version.*

```
C:\Users\javatpoint> npx create-react-app reactproject
```



The above command will take some time to install the React and create a new project with the name "reactproject." Now, we can see the terminal as like below.



```
Command Prompt
added 1385 packages from 675 contributors and audited 902050 packages in 305.218s
found 0 vulnerabilities

Success! Created reactapp at C:\Users\javatpoint\reactapp
Inside that directory, you can run several commands:

 npm start
 Starts the development server.

 npm run build
 Bundles the app into static files for production.

 npm test
 Starts the test runner.

 npm run eject
 Removes this tool and copies build dependencies, configuration files
 and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

 cd reactapp
 npm start

Happy hacking!
C:\Users\javatpoint>
```

The above screen tells that the React project is created successfully on our system. Now, we need to start the server so that we can access the application on the browser. Type the following command in the terminal window.

Run the React Application

Now you are ready to run your first *real* React application! Run this command to move to the

```
cd my-react-app
```

my-react-app directory:

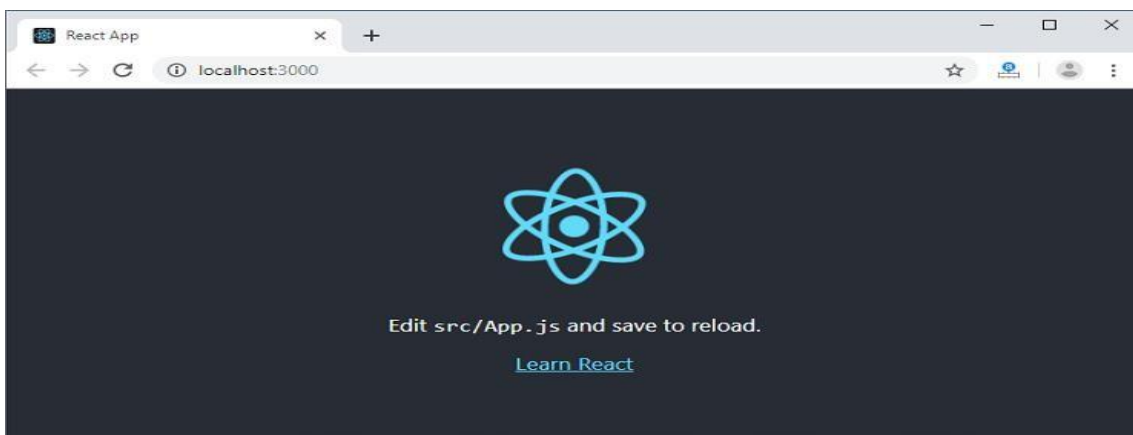
Run this command to run the React application my-react-app:

```
npm start
```

A new browser window will pop up with your newly created React App! If not, open your browser and type localhost:3000 in the address bar.

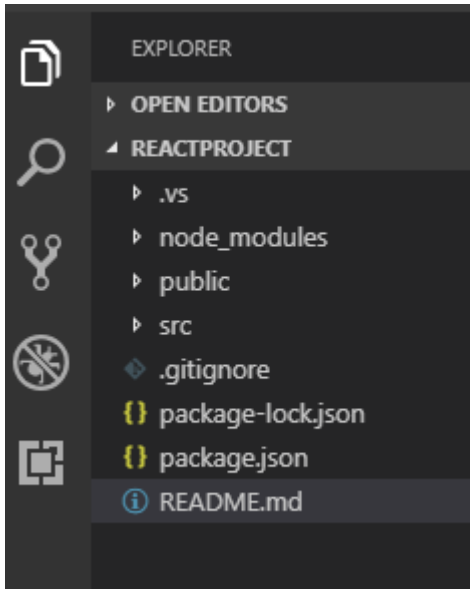
The result:

NPM is a package manager which starts the server and access the application at default



Next, open the project on Code editor. Here, I am using Visual Studio Code. Our project's default

structure looks like as below image.



In React application, there are several files and folders in the root directory. Some of them are as follows:

1. **node\_modules:** It contains the React library and any other third party libraries needed.
2. **public:** It holds the public assets of the application. It contains the index.html where React will mount the application by default on the `<div id="root"></div>` element.
3. **src:** It contains the App.css, App.js, App.test.js, index.css, index.js, and serviceWorker.js files. Here, the App.js file always responsible for displaying the output screen in React.
4. **package-lock.json:** It is generated automatically for any operations where npm package modifies either the node\_modules tree or package.json. It cannot be published. It will be ignored if it finds any other place rather than the top-level package.
5. **package.json:** It holds various metadata required for the project. It gives information to npm, which allows to identify the project as well as handle the project's dependencies.
6. **README.md:** It provides the documentation to read about React topics.

### Modify the React Application

Now, open the **src >> App.js** file and make changes which you want to display on the screen. After making desired changes, **save** the file. As soon as we save the file, Webpack recompiles the code, and the page will refresh automatically, and changes are reflected on the browser screen. Now, we can create as many components as we want, import the newly created component inside the **App.js** file and that file will be included in our main **index.html** file after compiling by Webpack.

Look in the my-react-app directory, and you will find a src folder. Inside the src folder there is a file called App.js, open it and it will look like this:

```
/myReactApp/src/App.js:
```

```
import './App.css';
```

```

function App() {
 return (
 <div className="App">
 <header className="App-header">

 <p>
 Edit <code>src/App.js</code> and save to reload.
 </p>
 <a
 className="App-link" href="https://reactjs.org"
 target="_blank"
 rel="noopener noreferrer">
 Learn React

 </header>
 </div>
);
 }
 export default App;

```

Try changing the HTML content and save the file.

Notice that the changes are visible immediately after you save the file, you do not have to reload the browser!

### Example

Replace all the content inside the `<div className="App">` with a `<h1>` element. See the changes in the browser when you click Save.

```

function App()

{

 return (
 <div className="App">
 <h1>Hello World!</h1>

```

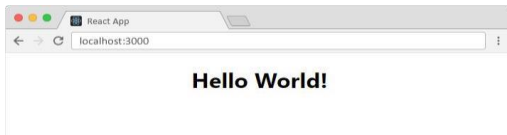
```

- </div>
-);
- }
- export default App;

```

Notice that we have removed the imports we do not need (logo.svg and App.css).

The result:



Next, if we want to make the project for the production mode, type the following command. This command will generate the production build, which is best optimized.

1. \$ npm build

## React Components

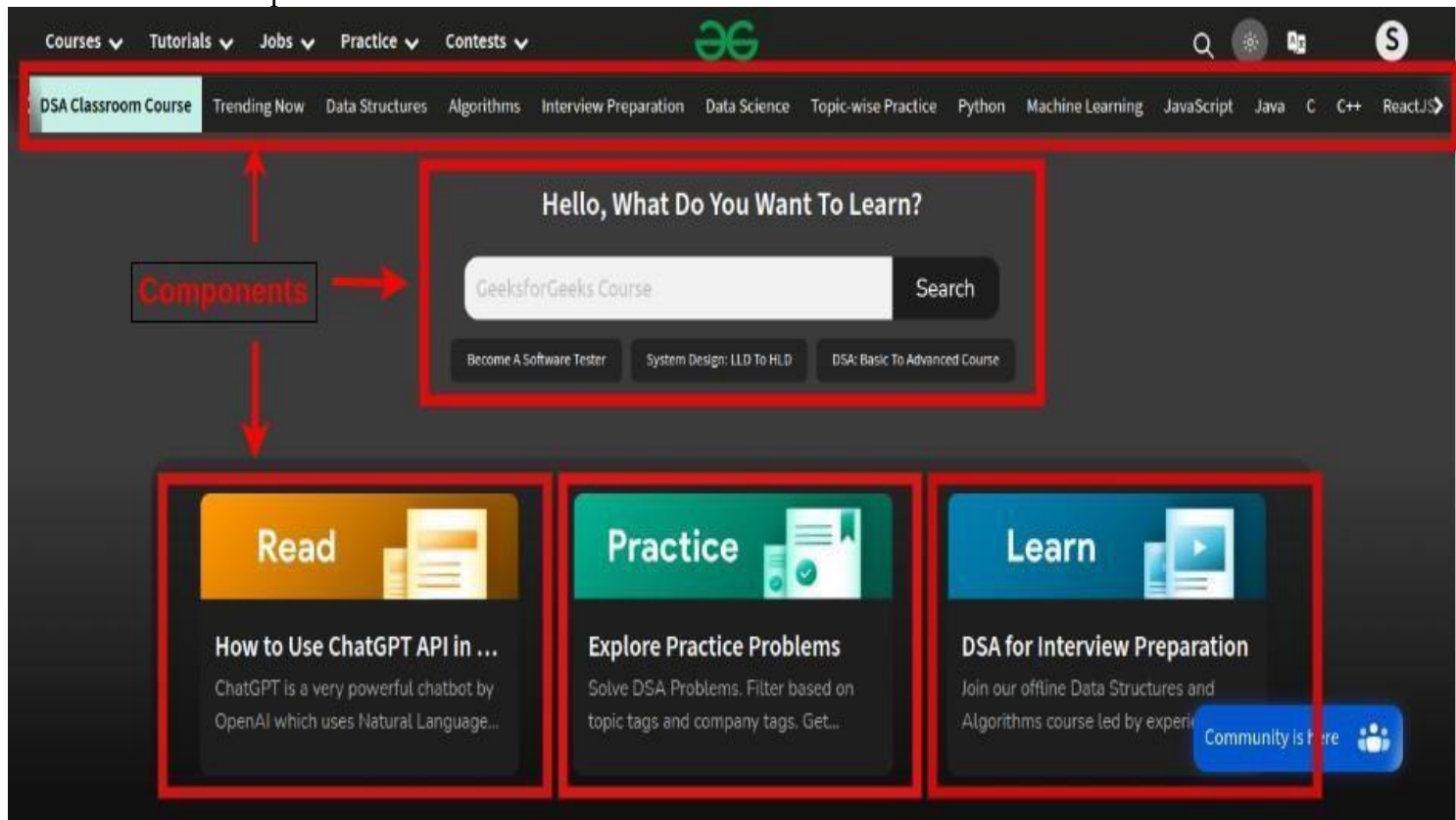
1. What are React Components?

**React Components** are the building block of React Application. They are the reusable code blocks containing logics and UI elements. They have the same purpose as **JavaScript functions** and return **HTML**. Components make the task of building UI much easier.

A UI is broken down into multiple individual pieces called components. You can work on components independently and then merge them all into a **parent component** which will be your final UI.

Components promote **efficiency** and **scalability** in web development by allowing developers to compose, combine, and customize them as needed.

- You can see in the below image we have broken down the UI of GeeksforGeeks's homepage into individual components.



Components in React return a piece of JSX code that tells what should be **rendered on the screen**.

## Types of Components in React

In React, we mainly have two types of components:

- **Functional Components**
- **Class Components**

- Functional Component

**Functional components** are just like JavaScript functions that accept properties and return a React element. We can create a functional component in React by writing a JavaScript function. These functions may or may not receive data as parameters. The below example shows a valid functional component in React:

**Syntax:**

```
function demoComponent() {
 return (<h1>
 Welcome Message!
 </h1>);
}
```

**Example:** Create a function component called welcome. Javascript

```
function welcome() {
 return <h1>Hello, Welcome to GeeksforGeeks!</h1>;
}
```

- **Class Component**

The [class components](#) are a little more complex than the functional components. A class component can show **inheritance** and access data of other components.

Class Component must include the line “**extends React.Component**” to pass data from one class component to another class component. We can use JavaScript ES6 classes to create class-based components in React.

**Syntax:**

```
class Democomponent extends
 React.Component { render() {
 return <h1>Welcome Message!</h1>;
 }
}
```

The below example shows a valid class-based component in React:

**Example:** Create a class component called welcome. Javascript

```
class Welcome extends
 Component { render() {
 return <h1>Hello, Welcome to GeeksforGeeks!</h1>;
 }
}
```

The components we created in the above two examples are equivalent, and we also have stated the basic difference between a functional component and a class component.

#### Functional Component vs Class Component

- A functional component is best suited for cases where the component doesn't need to interact with other components or manage complex states.
- Functional components are ideal for **presenting static UI elements** or composing multiple simple components together under a single parent component.
- While class-based components can achieve the same result, they are generally **less efficient** compared to functional components. Therefore, it's recommended to not use class components for general use.

#### Rendering React Components

**Rendering Components** means turning your component code into the UI(User Interface) that users see on the screen.

React is capable of rendering user-defined components. To render a component in React we can initialize an element with a user-defined component and pass this element as the first parameter to [ReactDOM.render\(\)](#) or directly pass the component as the first argument to the ReactDOM.render() method.

The below syntax shows how to initialize a component to an element:

```
const elementName = <ComponentName/>;
```

In the above syntax, the *ComponentName* is the name of the user-defined component.

**Note:** The name of a component should always start with a capital letter. This is done to differentiate a component tag from an [HTML tag](#).

**Example:** This example renders a component named Welcome to the Screen. javascript

```
// Filename - src/index.js:

import React from "react";
import ReactDOM from "react-dom";

// This is a functional component
const Welcome = () => {
 return (
 <>
 <h1>Hello World!</h1>;
 </>
)
};
ReactDOM.render(
 <Welcome />,
 document.getElementById("root")
);
```

**Output:** This output will be visible on the **http://localhost:3000/** on the browser window.

# Hello World!

### Explanation:

Let us see step-wise what is happening in the above example:

- We call the ReactDOM.render() as the first parameter.
- React then calls the component Welcome, which returns <h1>Hello World!</h1>; as the result.
- Then the ReactDOM efficiently updates the DOM to match with the returned element and renders that element to the DOM element with id as “root”.

### Components in Components

We can call components inside another component

### Example:

Javascript

```
// Filename - src/index.js:

import React from "react";
import ReactDOM from "react-dom";

const Greet = () => {
 return <h1>Hello Geek</h1>
}

// This is a functional component
const Welcome = () => {
 return <Greet />;
};

ReactDOM.render(
 <Welcome />,
 document.getElementById("root")
);
```

The above code will give the same output as other examples but here we have called the Greet component inside the Welcome Component.

## React State:-

### a. What Is 'State' in ReactJS?

The state is a built-in React object that is used to contain data or information about the component. A component's state can change over time; whenever it changes, the component re-renders. The change in state can happen as a response to user action or system-generated events and these changes determine the behavior of the component and how it will render.

### b. Creating State Object

Creating a state is essential to building dynamic and interactive components. We can create a state object within the **constructor** of the class component.

Example:-

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Car extends React.Component { constructor(props) {
 super(props);
 this.state = {
 brand: "Ford",
 model: "Mustang",
 color: "red",
 year: 1964
 };
}
 render()
 { return (
 <div>
```



```

- <h1>My {this.state.brand}</h1>
 <p>
 It is a {this.state.color}
 {this.state.model} from {this.state.year}.
 </p>
 </div>
);
}
}

```

Output:-

# My Ford

It is a red Mustang from 1964.

## Changing the **state** Object

To change a value in the state object, use the **this.setState()** method. When a value in the **state** object changes, the component will re-render, meaning that the output will change according to the new value(s).

### Example:

Add a button with an onClick event that will change the color

```

property: import React from 'react';
import ReactDOM from 'react-dom/client';

class Car extends
 React.Component {
 constructor(props) {
 super(props);
 this.state = {
 brand: "Ford",
 model: "Mustang",
 color: "red",
 year: 1964
 };
 }
 changeColor = () => { this.setState({color: "blue"}); }
 render() { return (
 <div>
 <h1>My {this.state.brand}</h1>
 <p>
 It is a {this.state.color}
 {
 this.state.model} from {this.state.year}.
 </p>
 <button type="button"
 onClick={this.changeColor}

```

```

- >Change color</button>
 </div>
);
}
}

```

```

Const root = ReactDOM.createRoot(document.getElementById('root')); root.render(<Car
/>);

```

**Output:-**

# My Ford

It is a red Mustang from 1964.

Change color

# My Ford

It is a blue Mustang from 1964.

Change color

## React Hook

Hooks were added to React in version 16.8.

Hooks allow function components to have access to state and other React features. Because of this, class components are generally no longer needed.

Although Hooks generally replace class components, there are no plans to remove classes from React.

What is a Hook?

Hooks allow us to "hook" into React features such as state and lifecycle methods.

### Example:

Here is an example of a Hook.

```
import React, { useState } from "react";
```

```
import ReactDOM from "react-dom/client"; function FavoriteColor() {
```

```
 const [color, setColor] = useState("red"); return (
```

```
 <
```

```
 <h1>My favorite color is {color}!</h1>
```

```
 <button type="button"
```

```
 onClick={() => setColor("blue")}

```

```
 >Blue</button>

```

```
 </>
)
}

```

```

 _);
 }

 const root = ReactDOM.createRoot(document.getElementById('root')); root.render(<FavoriteColor />);

```

**Output:-**

**My favorite color is green!**

Blue When click button it change blue

You must **import** Hooks from **react**.

Here we are using the **useState** Hook to keep track of the application state. State generally refers to application data or properties that need to be tracked.

### Hook Rules

There are 3 rules for hooks:

- Hooks can only be called inside React function components.
- Hooks can only be called at the top level of a component.
- Hooks cannot be conditional

**Note:** Hooks will not work in React class components.

**React Props:-**

**Defination:-**

Props stand for "**Properties**." They are **read-only** components. It is an object which stores the value of attributes of a tag and work similar to the HTML attributes. It gives a way to pass data from one component to other components. It is similar to function arguments. Props are passed to the component in the same way as arguments passed in a function.

Props are **immutable** so we cannot modify the props from inside the component. Inside the components, we can add attributes called props. These attributes are available in the component as **this.props** and can be used to render dynamic data in our render method.

When you need immutable data in the component, you have to add props to **ReactDOM.render()** method in the **index.js** file of your ReactJS project and used it inside the component in which you need.

**Syntax:**

```

// Passing Props
<DemoComponent sampleProp = "HelloProp" />

```

**Syntax:**

```

//Accessing props
this.props.propName;

```

**Example:-**

```

import React from 'react';
import ReactDOM from 'react-dom/client';

function Car(props) {
 return <h2>I am a { props.brand.model }!</h2>;
}

function Garage() {
 const carInfo = { name: "Ford", model:
 "Mustang" }; return (
 <
 <h1>Who lives in my garage?</h1>
 <Car brand={ carInfo } />
 </>
);
}

const root = ReactDOM.createRoot(document.getElementById('root')); root.render(<Garage />);

```

Output:-

**Who lives in my Garage?**

**I am a Mustang!**

## - React Forms

### a. Definition:-

**React Forms** are the components **used to collect and manage the user inputs**. These components includes the input elements like text field, check box, date input, dropdowns etc. In [HTML forms](#) the data is usually handled by the DOM itself but in the case of React Forms data is handled by the react components.

**React forms** are the way to collect the user data in a React application. React typically utilize controlled components to manage form state and handle user input changes efficiently. It provides additional functionality such as preventing the default behavior of the form which refreshes the browser after the form is submitted.

In React Forms, all the form data is stored in the React's component state, so it can handle the form submission and retrieve data that the user entered. To do this we use controlled components.

### **Controlled Components**

In simple HTML elements like input tags, the value of the input field is changed whenever the user type. But, In React, whatever the value the user types we save it in state and pass the same value to the input tag as its value, so here DOM does not change its value, it is controlled by react state. These are known as **Controlled Components**.

### b. Creating Form:-

React offers a stateful, reactive approach to build a form. The component rather than the DOM usually handles the React form. In React, the form is usually implemented by using controlled components.

There are mainly two types of form input in React.

1. **Uncontrolled component**
2. **Controlled component**

### **1. Uncontrolled component**

The uncontrolled input is similar to the traditional HTML form inputs. The DOM itself handles the form data. Here, the HTML elements maintain their own state that will be updated when the input value changes. To write an uncontrolled component, you need to use a ref to get form values from the DOM. In other words, there is no need to write an event handler for every state update. You can use a ref to access the input field value of the form from the DOM.

### **Example**

In this example, the code accepts a field **username** and **company name** in an uncontrolled component.

1. `import React, { Component } from 'react';`
2. `class App extends React.Component {`
3. `constructor(props) {`
4. `super(props);`

```

5. this.updateSubmit = this.updateSubmit.bind(this);
6. this.input = React.createRef();
7. }
8. updateSubmit(event) {
9. alert('You have entered the UserName and CompanyName successfully.');
```

10. event.preventDefault();

11. }

12. render() {

13. return (

14. <form onSubmit={this.updateSubmit}>

15. <h1>Uncontrolled Form Example</h1>

16. <label>Name:

17. <input type="text" ref={this.input} />

18. </label>

19. <label>

20. CompanyName:

21. <input type="text" ref={this.input} />

22. </label>

23. <input type="submit" value="Submit" />

24. </form>

25. );

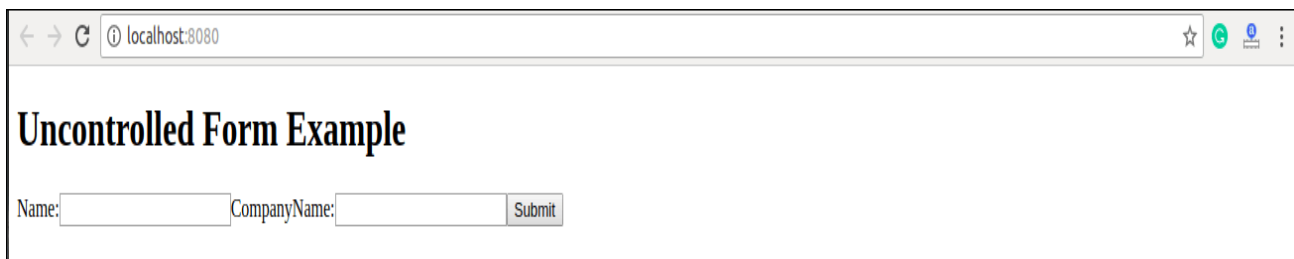
26. }

27. }

28. export default App;

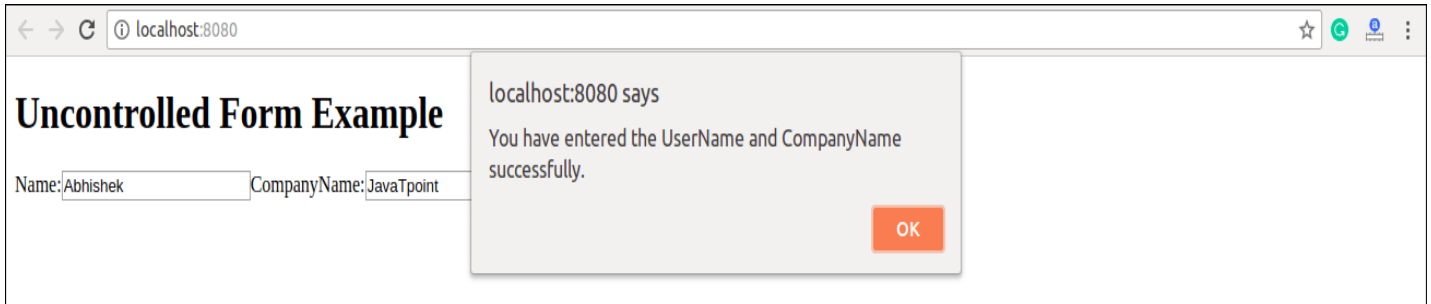
## Output

When you execute the above code, you will see the following screen.



The screenshot shows a web browser window with the address bar set to 'localhost:8080'. The page displays the title 'Uncontrolled Form Example'. Below the title, there is a form with two text input fields. The first field is labeled 'Name:' and the second is labeled 'CompanyName:'. To the right of these fields is a button labeled 'Submit'.

After filling the data in the field, you get the message that can be seen in the below screen.



## Controlled Component

In HTML, form elements typically maintain their own state and update it according to the user input. In the controlled component, the input form element is handled by the component rather than the DOM. Here, the mutable state is kept in the state property and will be updated only with **setState()** method.

Controlled components have functions that govern the data passing into them on every **onChange event**, rather than grabbing the data only once, e.g., when you click a **submit button**. This data is then saved to state and updated with **setState()** method. This makes component have better control over the form elements and data.

A controlled component takes its current value through **props** and notifies the changes through **callbacks** like an **onChange** event. A parent component "controls" this changes by handling the callback and managing its own state and then passing the new values as props to the controlled component. It is also called as a "dumb component."

### Example

1. import React, { Component } from 'react';
2. class App extends React.Component {
3.     constructor(props) {
4.         super(props);
5.         this.state = {value: ""};
6.         this.handleChange = this.handleChange.bind(this);
7.         this.handleSubmit = this.handleSubmit.bind(this);
8.     }
9.     handleChange(event) {
10.         this.setState({value: event.target.value});
11.     }
12.     handleSubmit(event) {
13.         alert('You have submitted the input successfully: ' + this.state.value);
14.         event.preventDefault();
15.     }
16.     render() {
17.         return (
18.             <form onSubmit={this.handleSubmit}>

```

19. <h1>Controlled Form Example</h1>
20. <label>
21. Name:
22. <input type="text" value={this.state.value} onChange={this.handleChange}
 />
23. </label>
24. <input type="submit" value="Submit" />
25. </form>

26.);
27. }
28. }
29. export default App;

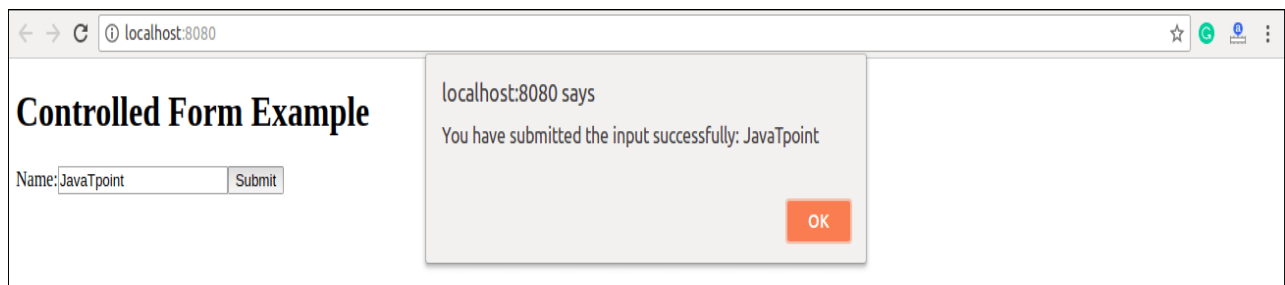
```

Output:-

When you execute the above code, you will see the following screen.



After filling the data in the field, you get the message that can be seen in the below screen.



(or) controlled form

### Submitting Forms:-

We can use the **useState** Hook to keep track of each inputs value and provide a "single source of truth" for the entire application.

You can control the submit action by adding an event handler in the **onSubmit** attribute for the **<form>**:

### Example:-

```

import { useState } from
"react"; import ReactDOM
from 'react-dom/client';

```



```

function MyForm() {
 const [name, setName] = useState("");

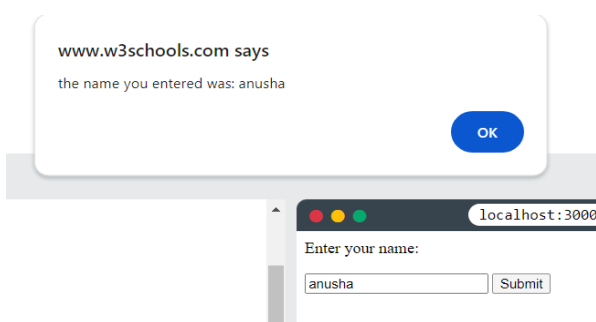
 const handleSubmit = (event)
 => { event.preventDefault();
 alert(`The name you entered was: ${name}`);
 }

 return (
 <form onSubmit={handleSubmit}>
 <label>Enter your name:
 <input
 type
 e="
 tex
 t"
 val
 ue
 ={
 na
 me
 }
 onChange={(e) => setName(e.target.value)}
 />
 </label>
 <input type="submit" />
 </form>
)
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);

```

### Output:-



### Multiple Input Fields:-

You can control the values of more than one input field by adding a name attribute to each element. We will initialize our state with an empty object.

To access the fields in the event handler use the event.target.name and

event.target.value syntax. To update the state, use square brackets [bracket notation]

around the property name.

```
import { useState } from
"react"; import ReactDOM
from "react-dom/client";

function MyForm() {
 const [inputs, setInputs] = useState({});

 const handleChange =
 (event) => { const name
 = event.target.name;
 const value =
 event.target.value;
 setInputs(values => ({...values, [name]: value}))
 }

 const handleSubmit = (event)
 => { event.preventDefault();
 console.log(inputs);
 }

 return (
 <form onSubmit={handleSubmit}>
 <label>Enter your name:
 <input
 type=
 "text"
 name
 ="use
 rname
 "
 value={inputs.username
 || ""}
 onChange={handleChange}
 />
 </label>
 <label>Enter your age:
 <input
 type=
 "number"
 name
 ="age"
 value={inputs.age ||
 ""}
 onChange={handleC
```

```

 hange}
 />
 </label>
 <input type="submit" />
 </form>
)
}

```

```

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);

```

```

/*
Click F12 and navigate to the
"Console view" to see the result
when you submit the form.
*/

```

Output:-

Enter your name:  Enter your age:

**Note:** We use the same event handler function for both input fields, we could write one event handler for each, but this gives us much cleaner code and is the preferred way in React.

### Textarea:-

The textarea element in React is slightly different from ordinary HTML.

In HTML the value of a textarea was the text between the start tag <textarea> and the end tag </textarea>.

```
<textarea>
```

Content of the textarea.

```
</textarea>
```

In React the value of a textarea is placed in a value attribute. We'll use the useStateHook to manage the value of the textarea:

### Example:

```

import { useState } from
"react"; import ReactDOM
from "react-dom/client";

function MyForm() {
 const [textarea, setTextarea] = useState(
 "The content of a textarea goes in the value attribute"
);
}

```

```

);

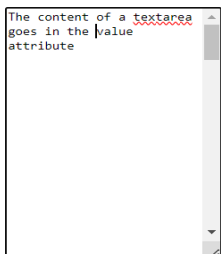
 const handleChange = (event) => {
 setTextarea(event.target.value);
 }

 return (
 <form>
 <textarea value={textarea} onChange={handleChange} />
 </form>
)
 }

 const root = ReactDOM.createRoot(document.getElementById('root'));
 root.render(<MyForm />);

```

### Output:-



### Select:-

A drop down list, or a select box, in React is also a bit different from HTML.

in HTML, the selected value in the drop down list was defined with the selected attribute:

### HTML:

```

<select>
 <option value="Ford">Ford</option>
 <option value="Volvo" selected>Volvo</option>
 <option value="Fiat">Fiat</option>
</select>

```

In React, the selected value is defined with a value attribute on the select tag:

### Example:

A simple select box, where the selected value "Volvo" is initialized in the constructor:

```
import { useState } from
"react"; import ReactDOM
from "react-dom/client";

function MyForm() {
 const [myCar, setMyCar] = useState("Volvo");

 const handleChange = (event)
 => {
 setMyCar(event.target.value)
 }

 return (
 <form>
 <select value={myCar} onChange={handleChange}>
 <option value="Ford">Ford</option>
 <option value="Volvo">Volvo</option>
 <option value="Fiat">Fiat</option>
 </select>
 </form>
)
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);
```

By making these slight changes to <textarea> and <select>, React is able to handle all input elements in the same way.

**Output:-**



### Component Life-Cycle:-

In React, components have a lifecycle that consists of different phases. Each phase has a set of lifecycle methods that are called at specific points in the component's lifecycle. These methods allow you to control the component's behavior and perform specific actions at different stages of its lifecycle.

The lifecycle of the component is divided into **four phases**. They are:

1. Initial Phase
2. Mounting Phase
3. Updating Phase
4. Unmounting Phase

## 1. Initial Phase

It is the **birth** phase of the lifecycle of a ReactJS component. Here, the component starts its journey on a way to the DOM. In this phase, a component contains the default Props and initial State. These default properties are done in the constructor of a component. The initial phase only occurs once and consists of the following methods.

**Example:-**

```
class Clock extends React.Component
{ constructor(props)
{
 // Calling the constructor of
 // Parent Class React.Component
 super(props);

 // Setting the initial state
 this.state = { date : new
 Date() };
}
```

## 2. Mounting

Mounting means putting elements into the DOM.

React has four built-in methods that gets called, in this order, when mounting a component:

- `constructor()`
- `getDerivedStateFromProps()`
- `render()`
- `componentDidMount()`

The `render()` method is required and will always be called, the others are optional and will be called if you define them.

- **Constructor:-**

**Method to initialize state and bind methods. Executed before the component is mounted.**

**Example:-**

```
import React from 'react';
```

```
import ReactDOM from 'react-dom/client';

class Header extends React.Component { constructor(props) {
 super(props);
 this.state = {favoritecolor: "red"};
}
render() { return (
 <h1>My Favorite Color is {this.state.favoritecolor}</h1>
);
}
}

const root = ReactDOM.createRoot(document.getElementById('root')); root.render(<Header
/>);
```

Output:-  
My Favorite Color is red

- **Static getDerivedStateFromProps:**

Used for updating the state based on props. Executed before every render.

Example:-

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component { constructor(props) {
 super(props);
 this.state = {favoritecolor: "red"};
}
static getDerivedStateFromProps(props, state) { return {favoritecolor: props.favcol };
}
render() { return (
 <h1>My Favorite Color is {this.state.favoritecolor}</h1>
);
}
}

const root = ReactDOM.createRoot(document.getElementById('root')); root.render(<Header
favcol="yellow"/>);
```

- **Render**

Responsible for rendering JSX and updating the DOM.

Example:-

```
import React from 'react';
import ReactDOM from 'react-dom/client'; class Header extends React.Component {

render() { return (
 <h1>This is the content of the Header component</h1>
);
}
}

const root = ReactDOM.createRoot(document.getElementById('root')); root.render(<Header
```

`</>);`

Output:-

This is the content of the Header component

- `componentDidMount`

The `componentDidMount()` method is called after the component is rendered.

This is where you run statements that requires that the component is already placed in the DOM.

**Example:-**

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component { constructor(props) {
 super(props);
 this.state = {favoritecolor: "red"};
}
 componentDidMount() { setTimeout(() => {
 this.setState({favoritecolor: "yellow"})
 }, 1000)
}
 render() { return (
 <h1>My Favorite Color is {this.state.favoritecolor}</h1>
);
}
}

const root = ReactDOM.createRoot(document.getElementById('root')); root.render(<Header
/>);
```

Output:- My Favorite Color is yellow

### 3. Updating

The next phase in the lifecycle is when a component is *updated*.

A component is updated whenever there is a change in the component's `state` or `props`. React has five built-in methods that gets called, in this order, when a component is updated:

1. `getDerivedStateFromProps()`
2. `shouldComponentUpdate()`
3. `render()`
4. `getSnapshotBeforeUpdate()`



## 5. `componentDidUpdate()`

The `render()` method is required and will always be called, the others are optional and will be called if you define them.

- `getDerivedStateFromProps`:-

`getDerivedStateFromProps(props, state)` is a static method that is called just before `render()` method in both mounting and updating phase in React. It takes updated props and the current state as arguments.

Example:-

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component { constructor(props) {
 super(props);
 this.state = {favoritecolor: "red"};
}
static getDerivedStateFromProps(props, state)
{ return {favoritecolor: props.favcol };
}
changeColor = () => { this.setState({favoritecolor: "blue"});
}
render() { return (
 <div>
 <h1>My Favorite Color is {this.state.favoritecolor}</h1>
 <button type="button" onClick={this.changeColor}>Change color</button>
 </div>
);
}
}
const root = ReactDOM.createRoot(document.getElementById('root')); root.render(<Header
favcol="yellow" />);

/*
This example has a button that changes the favorite color to blue, but since the
getDerivedStateFromProps() method is called, the favorite color is still rendered as yellow
(because the method updates the state with the color from the favcol attribute).
*/
```

Output:-

My Favorite Color is yellow

Change color

- `shouldComponentUpdate`:-

In the `shouldComponentUpdate()` method you can return a Boolean value that specifies whether React should continue with the rendering or not.

The default value is **true**.

Example:-

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component { constructor(props) {
 super(props);
 this.state = {favoritecolor: "red"};
}
 shouldComponentUpdate() {
 return true;
 }
 changeColor = () => { this.setState({favoritecolor: "blue"});
}
 render() { return (
 <div>
 <h1>My Favorite Color is {this.state.favoritecolor}</h1>
 <button type="button" onClick={this.changeColor}>Change color</button>
 </div>
);
}
}
const root = ReactDOM.createRoot(document.getElementById('root')); root.render(<Header
/>);
```

Output:-

# My Favorite Color is red

Change color

Fig.Before updation

# My Favorite Color is blue

Change color

Fig.After updation

**Note:-**if shouldcomponent is return false means no execution.

- getSnapshotBeforeUpdate:-

In the **getSnapshotBeforeUpdate()** method you have access to the **props** and **state** *before* the update, meaning that even after the update, you can check what the values were *before* the update.

If the **getSnapshotBeforeUpdate()** method is present, you should also include the **componentDidUpdate()** method, otherwise you will get an error.

The example below might seem complicated, but all it does is this:

- When the component is *mounting* it is rendered with the favorite color "red".

When the component *has been mounted*, a timer changes the state, and after one second, the favorite color becomes "yellow".

This action triggers the *update* phase, and since this component has a `getSnapshotBeforeUpdate()` method, this method is executed, and writes a message to the empty DIV1 element.

Then the `componentDidUpdate()` method is executed and writes a message in the empty DIV2 element:

### Example:-

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component { constructor(props) {
 super(props);
 this.state = {favoritecolor: "red"};
}
componentDidMount() { setTimeout(() => {
 this.setState({favoritecolor: "yellow"})
}, 1000)
}
getSnapshotBeforeUpdate(prevProps,prevState)
{
 document.getElementById("div1").innerHTML ="Before the update, the favorite was " +
 prevState.favoritecolor;
}
componentDidUpdate() { document.getElementById("div2").innerHTML = "The updated
favorite is " + this.state.favoritecolor;
}
render() { return (
 <div>
 <h1>My Favorite Color is {this.state.favoritecolor}</h1>
 <div id="div1"></div>
 <div id="div2"></div>
 </div>
);
}
}

const root = ReactDOM.createRoot(document.getElementById('root')); root.render(<Header
/>);
```

Output:-

# My Favorite Color is yellow

Before the update, the favorite was red

The updated favorite is yellow

- **componentDidUpdate**

The `componentDidUpdate` method is called after the component is updated in the DOM. The example below might seem complicated, but all it does is this:

When the component is *mounting* it is rendered with the favorite color "red".

When the component *has been mounted*, a timer changes the state, and the color becomes "yellow".

This action triggers the *update* phase, and since this component has a `componentDidUpdate` method, this method is executed and writes a message in the empty DIV element:

Example:-

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component { constructor(props) {
 super(props);
 this.state = {favoritecolor: "red"};
}
 componentDidMount() { setTimeout(() => {
 this.setState({favoritecolor: "yellow"})
 }, 1000)
}
 componentDidUpdate() {
 document.getElementById("mydiv").innerHTML = "The updated favorite is " +
 this.state.favoritecolor;
 }
 render() { return (
 <div>
 <h1>My Favorite Color is {this.state.favoritecolor}</h1>
 <div id="mydiv"></div>
 </div>
);
}
}
```

```
const root = ReactDOM.createRoot(document.getElementById('root')); root.render(<Header />);
```

**Output:-**

#### 4. Unmounting

# My Favorite Color is yellow

The updated favorite is yellow

The next phase in the lifecycle is when a component is removed from the DOM, or *unmounting* as React likes to call it.

React has only one built-in method that gets called when a component is unmounted:

- `componentWillUnmount()`
- `componentWillUnmount`

The `componentWillUnmount` method is called when the component is about to be removed from the DOM.

**Example:-**

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Container extends React.Component { constructor(props) {
 super(props);
 this.state = {show: true};
}
 delHeader = ()
 => {
 this.setState({
 show: false});
 }
 render() {
 let myheader;
 if (this.state.show) { myheader = <Child />;
 };
 return (
 <div>
 {myheader}
 <button type="button" onClick={this.delHeader}>Delete Header</button>
 </div>
);
 }
}

class Child extends
 React.Component {
 componentWillMount() {
 alert("The component named Header is about to be unmounted.");
 }
 render() { return (
 <h1>Hello World!</h1>
);
}
```

```

}
const root = ReactDOM.createRoot(document.getElementById('root')); root.render(<Container
/>);

```

Output:-

# Hello World!

Delete Header

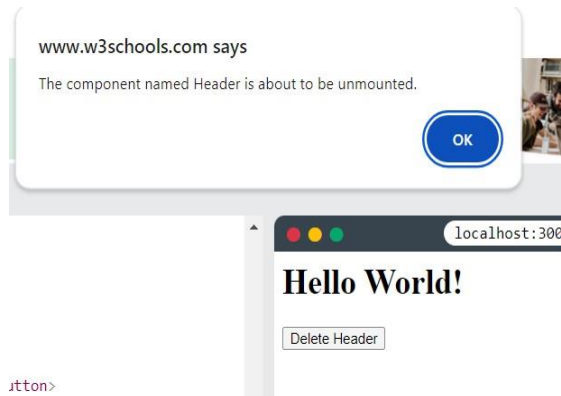


Fig. Before deleting and after deleting alert message

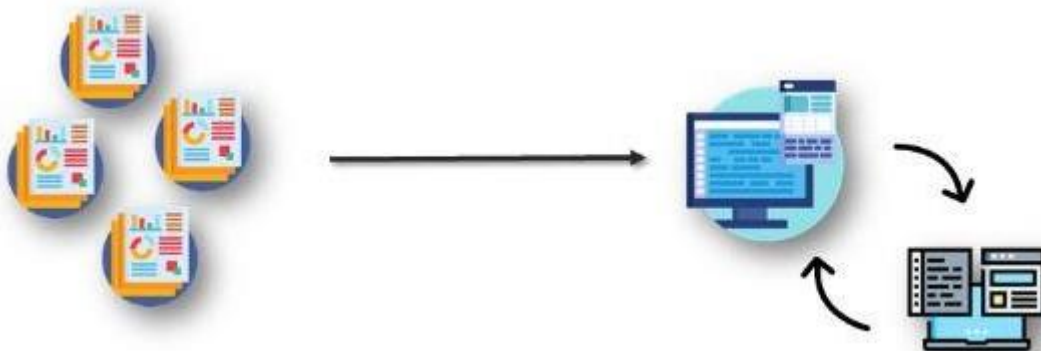
Delete Header

Fig. After Deleting data removed

## React Redux:-

### c. Why Redux?

State transfer between components is pretty messy in React since it is hard to keep track of which component the data is coming from. It becomes really complicated if users are working with a large number of states within an application.



Redux solves the state transfer problem by storing all of the states in a single place called a store. So, managing and transferring states becomes easier as all the states are stored in the same convenient store. Every component in the application can then directly access the required state from that store.

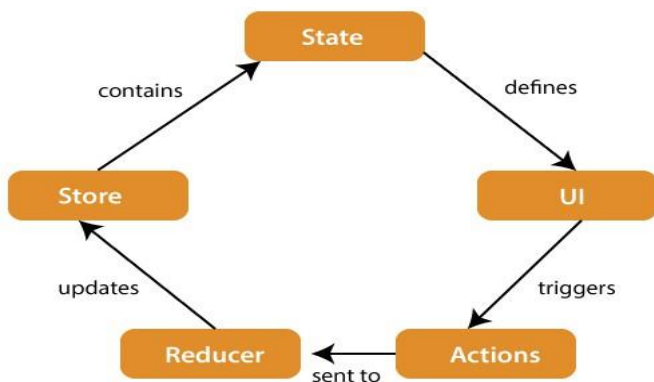
d. Defination:-

Redux is an open-source JavaScript library used to manage application state. React uses Redux for building the user interface. It was first introduced by **Dan Abramov** and **Andrew Clark** in **2015**.

React Redux is the official React binding for Redux. It allows React components to read data from a Redux Store, and dispatch **Actions** to the **Store** to update data. Redux helps apps to scale by providing a sensible way to manage state through a unidirectional data flow model. React Redux is conceptually simple. It subscribes to the Redux store, checks to see if the data which your component wants have changed, and re-renders your component.

Redux was inspired by Flux. Redux studied the Flux architecture and omitted unnecessary complexity.

e. Redux Architecture



The components of Redux architecture are explained below.

**STORE:** A Store is a place where the entire state of your application lists. It manages the status of the application and has a dispatch(action) function. It is like a brain responsible for all moving parts in Redux.

**ACTION:** Action is sent or dispatched from the view which are payloads that can be read by Reducers. It is a pure object created to store the information of the user's event. It includes information such as type of action, time of occurrence, location of occurrence, its coordinates, and which state it aims to change.

**REDUCER:** Reducer read the payloads from the actions and then updates the store via the state accordingly. It is a pure function to return a new state from the initial state.

f. Redux Installation

**Requirements:** React Redux requires React 16.8.3 or later version.

To use React Redux with React application, you need to install the below command.

- npm install redux
- npm install react-redux
- 

### Example Code For Redux Store:-

```
import
{createStore} from
"redux"; const
initialState={
 balance:0, fullName:"", mobile:,
}
function accountReducer(state=initialState,action){
 if(action.type==="deposit")
 {
 return {...state,balance:state.balance+ +action.payload};
 }
 else if(action.type==="withdraw")
 {
 return {...state,balance:state.balance- +action.payload};
 }
 else if(action.type==="mobileUpdate")
 {
 return {...state,mobile:action.payload};
 }
 else if(action.type==="nameUpdate")
 {
 return {...state,fullName:action.payload};
 }
 else
 return state
}

const store=createStore(accountReducer)
console.log(store.getState()); store.dispatch({type:"deposit",payload:1000})
console.log(store.getState());
store.dispatch({type:"withdraw",payload:100})
console.log(store.getState());
store.dispatch({type:"mobileUpdate",payload:8547961256})
console.log(store.getState());
store.dispatch({type:"nameUpdate",payload:"swathi"})
console.log(store.getState());
```

### Output:-

balance:900,



fullName:" swathi ", mobile: 8547961256,

g. Example Code:-Creating store and components Accessing data from store

React Redux includes a <Provider /> component, which makes the Redux store available to the rest of your app components:

- **Index.js**

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import { Provider } from 'react-redux';
import store from './store';
import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(document.getElementById('root')); root.render(
 <React.StrictMode>
 <Provider store={store}>
 <App />
 </Provider>

 </React.StrictMode>
);
```

- Rendering form and account components
- **App.js**

```
import Form from './Forms';
import Account from './Account'; function App() {
 return (<<Form/>
 <Account/>
 </>
);
}
```

export default App;

- Creating state Store data
- **Store.js**

```
import { createStore } from 'redux'; const initialState={
 balance:0, fullName:"Anusha", mobile:7854789689,
}
function accountReducer(state=initialState,action){
 if(action.type==="deposit")
 {
 return {...state,balance:state.balance++action.payload};
 }
}
```

```

else if(action.type==="withdraw")
{
return {...state,balance:state.balance- +action.payload};
}
else if(action.type==="mobileUpdate")
{
return {...state,mobile:action.payload};
}
else if(action.type==="nameUpdate")
{
return {...state,fullName:action.payload};
}
else
return state
}

```

```
const
```

```
store=createStore(accountRed
```

```
ucer) export default store;
```

- Hooks

React Redux provides a pair of custom React hooks that allow your React components to interact with the Redux store.

useSelector reads a value from the store state and subscribes to updates, while useDispatch returns the store's dispatch method to let you dispatch actions.

- Account.js

```
import { useSelector } from "react-redux";
```

```

function Account(){
 let data= useSelector((state)=>{
return state;
 }
)
 return <>
 <div>
 <h1>Account Details</h1>
 <h3>{data.balance}</h3>
 <h3>{data.fullName}</h3>
 <h3>{data.mobile}</h3>
 </div>
 </>;
}
export default Account;

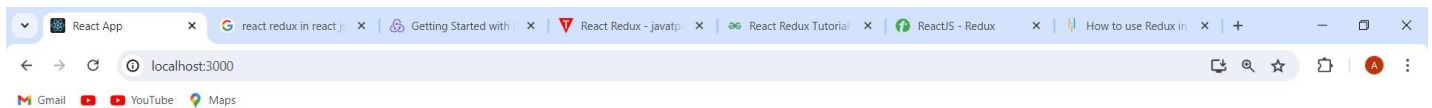
```

- Form.js

```
import { useState } from "react";
import { useDispatch } from "react-redux"; function Form(){
 let dispatch = useDispatch();
 const [amount, setAmount]=useState(""); return <
 <div>
 <h1>Form Component</h1>
 <input type="number" placeholder="Enter Amount" value={amount}
 onChange={(e)=>{
 let data=e.target.value; setAmount(data);
 }}
 />
 <button onClick={()=>{ dispatch({type:"deposit",payload:amount})
 }}>change amount</button>
 </div>
 </>;
}
```

export default Form;

**Output:-**



## Form Component

Enter Amount  change amount

## Account Details

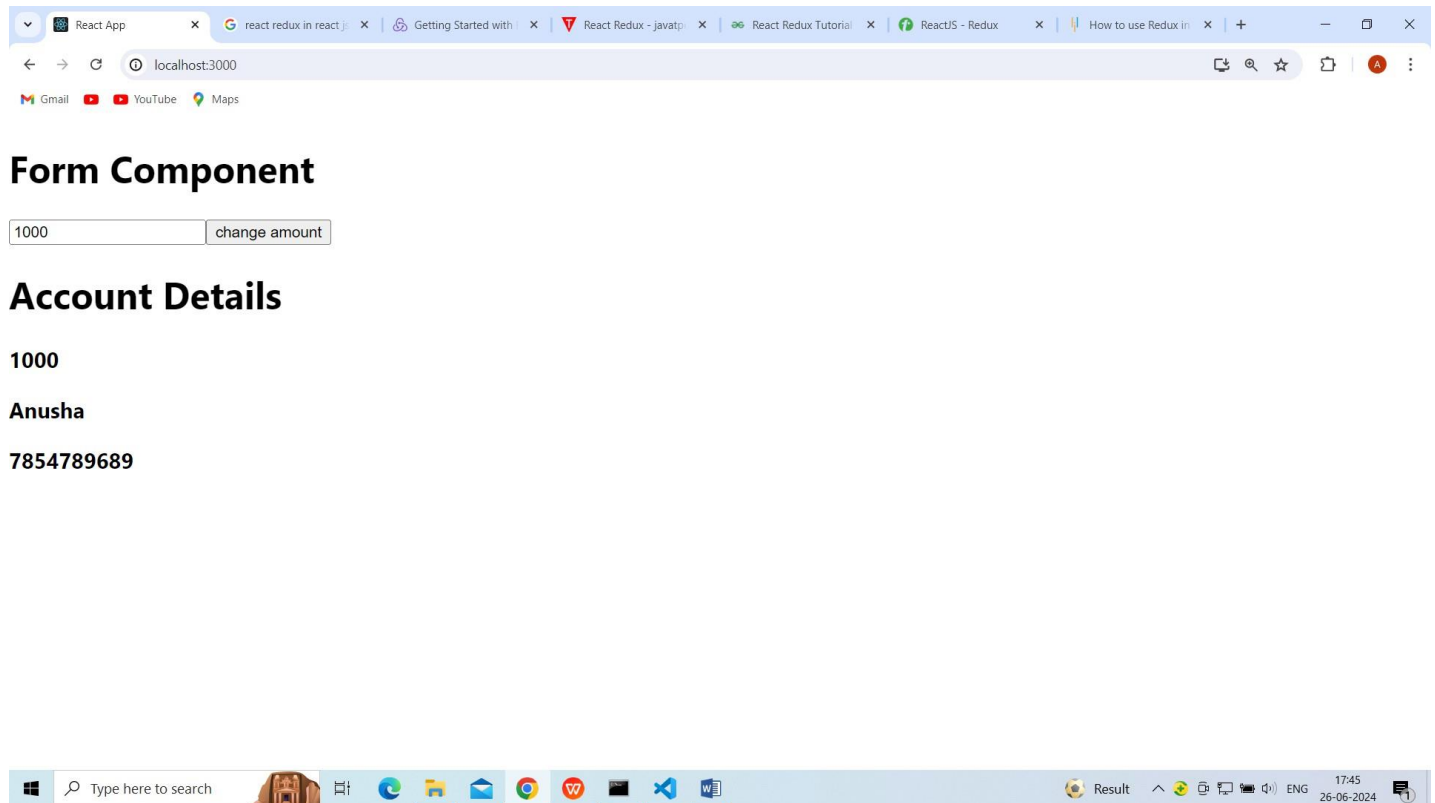
0

Anusha

7854789689



**Fig. Before Updating Data**



**Fig. After Updating Data**

## React Router

What is a React Router?

**React Router** is a standard library for routing in React. It enables the navigation among views of various components in a React Application, allows changing the browser URL, and keeps the UI in sync with the URL. Let us create a simple application to React to understand how the React Router works. The application will contain three components the **home component**, the **Blogs component**, and the **contact component**. We will use React Router to navigate between these components.

Steps to Use React Router

**Step 1:** Initialize a react project.

```
npx create-react-app Reactrouterprogram
```

**Step 2:** Install react-router in your application write the following command in your terminal

```
npm i react-router-dom
```

**Step 3:** Importing React Router

```
import { BrowserRouter, Routes, Route } from "react-router-dom";
```

Folder Structure:

The updated dependencies in **package.json** file.

```
"dependencies": {
 "@testing-library/jest-dom":
 "^5.17.0", "@testing-library/react":
 "^13.4.0", "@testing-library/user-
 event": "^13.5.0", "react":
 "^18.2.0",
 "react-dom": "^18.2.0",
 "react-router-dom": "^6.22.1",
 "react-scripts": "5.0.1",
 "web-vitals": "^2.1.4"
},
```

## React Router Components

The Main Components of React Router are:

- **BrowserRouter:** BrowserRouter is a router implementation that uses the HTML5 history API(pushState, replaceState, and the popstate event) to keep your UI in sync with the URL. It is the parent component that is used to store all of the other components.
- **Routes:** It's a new component introduced in the v6 and an upgrade of the component. The main advantages of Routes over Switch are:
  - Relative s and s
  - Routes are chosen based on the best match instead of being traversed in order.
- **Route:** Route is the conditionally shown component that renders some UI when its path matches the current URL.
- **Link:** The link component is used to create links to different routes and implement navigation around the application. It works like an HTML anchor tag.

### Implementing React Router

**Example:** This example shows navigation using react-router-dom to Home, About and Contact Components.

Within the **src** folder, we'll create a folder named

**pages** with several files: **src\pages\**:

- **Layout.js**
- **Home.js**
- **Blogs.js**
- **Contact.js**
- **NoPage.js**

Each file will contain a very basic React component. Now we will use our Router in our

**index.js** file.

Example:-

Use React Router to route to pages based on URL:

- index.js:

```
import ReactDOM from "react-dom/client";
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Layout from "./pages/Layout";
import Home from "./pages/Home";
import Blogs from "./pages/Blogs";
import Contact from "./pages/Contact";
import NoPage from "./pages/NoPage";

export default function App() {

return (
 <BrowserRouter>
 <Routes>
 <Route path="/" element={<Layout />} />
 <Route index element={<Home />} />
 <Route path="blogs" element={<Blogs />} />
 <Route path="contact" element={<Contact />} />
 <Route path="*" element={<NoPage />} />
 </Routes>
 </BrowserRouter>
);
}

const root = ReactDOM.createRoot(document.getElementById('root')); root.render(<App />);
```

### Example Explained

We wrap our content first with `<BrowserRouter>`.

Then we define our `<Routes>`. An application can have multiple `<Routes>`. Our basic example only uses one.

`<Route>`s can be nested. The first `<Route>` has a path of `/` and renders the `Layout` component.

The nested `<Route>`s inherit and add to the parent route. So the `blogs` path is combined with the parent and becomes `/blogs`.

The `Home` component route does not have a path but has an `index` attribute. That specifies this route as the default route for the parent route, which is `/`.

Setting the `path` to `*` will act as a catch-all for any undefined URLs. This is great for a 404 error page.

### Pages / Components

The `Layout` component has `<Outlet>` and `<Link>` elements. The `<Outlet>` renders the current route selected.

`<Link>` is used to set the URL and keep track of browsing history.

- Anytime we link to an internal path, we will use `<Link>` instead of `<a href="">`.

The "layout route" is a shared component that inserts common content on all pages, such as a navigation menu.

- Layout.js:

```
import { Outlet, Link } from
"react-router-dom"; const
Layout = () => {
 return (
 <div>
 <nav>

 <Link to="/">Home</Link>

 <Link to="/blogs">Blogs</Link>

 <Link to="/contact">Contact</Link>

 </nav>
 <Outlet />
 </div>)
 };
export default Layout;
```

- Home.js:  

```
const Home = () => { return <h1>Home</h1>;
};

export default Home;
```
- Blogs.js:  

```
const Blogs = () => {
 return <h1>Blog Articles</h1>;
};

export default Blogs;
```
- Contact.js:  

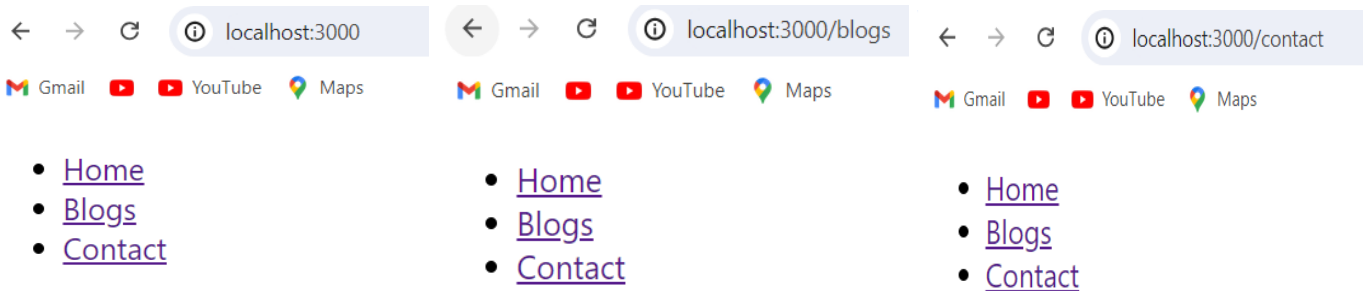
```
const Contact = () => {
 return <h1>Contact Me</h1>;
};

export default Contact;
```
- NoPage.js:  

```
const NoPage = () => { return <h1>404</h1>;
};

export default NoPage;
```

### Output:-



Home

Blog Articles

Contact Me



## What is React?

React is a front-end JavaScript library that allows you to build user interfaces from reusable UI components. React uses server-side rendering to provide a flexible and performance-based solution. It allows developers to create seamless UX and complex UI.

## What is Angular?

Angular is an open-source JavaScript front-end framework developed and managed by

Parameters	Angular	React
Developed By	Google	Facebook
Release Year	2009	2013
Written In	TypeScript	JavaScript
Technology Type	Full-fledged MVC framework written in JavaScript	library (View in MVC; requires Flux to implement architecture)
Concept	Brings JavaScript into HTML Works with the real DOM Client-side rendering	Brings HTML into JavaScript Works with the virtual DOM Server-side rendering
Data Binding	Two-way data binding	One-way data binding
Language	JavaScript + HTML	JavaScript + JSX
Learning Curve	Steep	Moderate
UI Rendering	Client/Server-Side	Client/Server-Side
Best Suited For	Highly active and interactive web apps	Larger apps with recurrent variable data
App Structure	Fixes and complicated MVC	Flexible component-based view
Dependency Injection	Fully supported	Not supported
Performance	High	High
DOM Type	Real	Virtual
Popular Apps	IBM, PayPal, Freelancer, Upwork	Facebook, Skype, Instagram, Walmart

Google's Angular team. Angular is the most popular client-side framework for developing scalable and high-performing mobile and web apps using HTML, CSS, and TypeScript. The latest version of Angular is Angular 13, which offers enterprise-ready web app development solutions and is widely used by companies for web development.

#### UNIT – IV

**Node.js: Getting Started with Node.js, Using Events, Listeners, Timers, and Call backs in Node.js, Handling Data I/O in Node.js, Accessing the File System from Node.js, Implementing Socket Services in Node.js.**

Q) Differentiate React and Node

Feature	React	Node.js
Purpose	Front-end library	Back-end framework
Language	JavaScript	JavaScript (with Node.js runtime)
Architecture	Component-based	Modular
Execution Environment	Browser	Server
Dependency Management	npm	npm or Yarn
Development Ecosystem	Large and active community	Large and active community

Q) Explain the significance of NPM.

Node.js provides a package manager called **NPM (Node Package Manager)** which is a collection of all open-source JavaScript libraries

- available in this world. It is the world's largest software registry maintained by the Node.js team. It helps in installing any library or module into your machine.

This can be used to install, update, or uninstall any package through NPM.

1. **Installing:** `npm install <package_name>[@<version>]`

This will create a folder `node_modules` in the current directory and put all the packages related files inside it. Here `@version` is optional if you don't specify the version, the latest version of the module will be downloaded.

There are two modes of installation through NPM

- i. **Global installation:** If we want to globally install any package or tool add `-g` to the command. On installing any package globally, that package gets added to the `PATH` so that we can run it from any location on the computer.

Syntax: `npm install -g <package_name>`

Eg. `npm install -g express`

- ii. **Local installation:** If we do not add `-g` to your command for installation, the modules get installed locally, within a `node_modules` folder under the root directory. This is the default mode, as well.

Syntax: `npm install`

`<package_name>` Eg. `npm install express`

Best Practice: Start all projects with `npm init`. This will create a new `package.json` for you which allows you to add a bunch of metadata to help others working on the project have the same setup as you.

2. **Update:** We can also update the packages downloaded from the registry to keep the code more secure and stable. Any update for a global package can be done using the following command.

Syntax: `npm update -g`

`<package_name>` Eg. `npm update express`

3. **Uninstall:** uninstall the packages :

We can uninstall the package or module, which we downloaded using the following command.

Syntax: `npm uninstall`

`<package_name>[@<version>]` Eg. `npm uninstall express@2.1.0`

4. **Publishing a module:** It is also possible to publish the custom modules that we created to NPM so that we can make our modules to be available for others to download.

- Steps to publish a custom module to NPM:
  - i. Create a public repository like Github to contain the code for the module.
  - ii. Create an account at <https://npmjs.org/signup>.
  - iii. Use the npm adduser command from a console prompt to add the user you created to the environment.
  - iv. Type in the username, password, and email that you used to create the account in step ii.
  - v. Modify the package.json file to include the new repository information and any keywords that you want made available in the registry search
  - vi. Publish the module using the following command from the application folder in the console: **npm publish**

The package will be published successfully. To use this module from npm, just use the "npm install mypackage" command from the command line and it will get installed.

- vii. To remove a package from the registry make sure that you have added a user with rights to the module to the environment using npm adduser and then execute the following command:

**npm unpublish <module\_name>**

5. **Security:** To perform a quick security check know, we can make use of npm audit which generates a report on the dependencies of your application. This report consists of security threats to your application and can help you fix vulnerabilities by providing npm commands and recommendations for further troubleshooting.

Syntax: npm audit

Q) What is package.json file? How to create it?

A Node project needs a configuration file named "package.json". It is a file that contains basic information about the project like the package name, version as well as more information like dependencies which specifies the additional packages required for the project.

To create a package.json file, open the Node command prompt and type the below command.

n

p

m

i

n

i

t

e

```

{
 "name": "my-package",
 "version": "1.0.0",
 "description": "A simple Node.js package for performing
 basic math operations.",
 "main":
 "index.js",
 "scripts":
 {
 "test": "echo \"Error: no test specified\" && exit 1"
 },
 "keywords": [
 "node",
 "math",
 "packag
 e"
],
 "author": "Your
Name", "license":
 "MIT"
}

```

Q) Explain about censorify module in node.js

The censored words are replaced with \*\*\*\* and that the new censored word gloomy is added to the censorify module instance censor.

**Install the module censorify using: npm install**

```

censorify. var censor = require("censorify");
console.log(censor.getCensoredWords());
console.log(censor.censor("Some very sad, bad and mad
text.)); censor.addCensoredWord("gloomy");
console.log(censor.getCensoredWords());
console.log(censor.censor("A very gloomy day.))

```

Output:

```

D:\PVP\MWA\Lab\NodeDemo> node
Censor.js ['sad', 'bad', 'mad']
Some very ****, **** and
**** text. ['sad', 'bad',
'mad', 'gloomy']
A very **** day.

```

Q) Explain how to create modules in node.js with an example.

A module can contain functions, classes, objects, or any other piece of code that can be shared between different parts of an application.

Node.js has a built-in module system that allows you to create and use modules. A module can be defined in a separate file, and can be loaded into other parts of the application using the require() function.

Eg.

**calc.js:**

```
exports.add = (a, b) => {
 console.log("Add Result:",
 a + b);
};

exports.subtract = (a, b) => {
 console.log("Sub Result:", a -
 b);
};
```

**demo.js:**

```
const myCalculator = require("./calc");
myCalculator.add(1, 2);
myCalculator.subtract(3, 2);
```

Here, the functions are exported using exports object in calc.js and require() function is used in demo.js which loads the module calc.

**Output:**

```
D:\PVP\MWA\Lab\NodeDemo> node
demo.js Add Result: 3
Sub Result: 1
```

Q) Explain how to write data to console in Node.js

One of the most useful modules in Node.js during the development process is the console module. This module provides a lot of functionality when writing debug and information statements to the console. The console module allows you to control output to the console, implement time delta output, and write tracebacks and assertions to the console.

Function	Description
<code>log([data], [...])</code>	Writes data output to the console. The data variable can be a string or an object that can be resolved to a string. Additional parameters can also be sent. For example:  <pre>console.log("There are %d items", 5); &gt;&gt;There are 5 items</pre>
<code>info([data], [...])</code>	Same as <code>console.log</code> .
<code>error([data], [...])</code>	Same as <code>console.log</code> ; however, the output is also sent to <code>stderr</code> .
<code>warn([data], [...])</code>	Same as <code>console.error</code> .
<code>dir(obj)</code>	Writes out a string representation of a JavaScript object to the console. For example:  <pre>console.dir({name:"Brad", role:"Author"}); &gt;&gt; { name: 'Brad', role: 'Author' }</pre>
<code>time(label)</code>	Assigns a current timestamp with ms precision to the string <code>label</code> .
<code>timeEnd(label)</code>	Creates a delta between the current time and the timestamp assigned to <code>label</code> and outputs the results. For example:  <pre>console.time("FileWrite"); f.write(data); //takes about 500ms console.timeEnd("FileWrite"); &gt;&gt; FileWrite: 500ms</pre>
<code>trace(label)</code>	Writes out a stack trace of the current position in code to <code>stderr</code> . For example:  <pre>module.trace("traceMark"); &gt;&gt;Trace: traceMark   at Object.&lt;anonymous&gt; (C:\test.js:24:9)   at Module._compile (module.js:456:26)   at Object.Module._ext.js (module.js:474:10)   at Module.load (module.js:356:32)   at Function.Module._load (module.js:312:12)   at Function.Module.runMain (module.js:497:10)   at startup (node.js:119:16)   at node.js:901:3</pre>
<code>assert(expression, [message])</code>	Writes the message and stack trace to the console if expression evaluates to <code>false</code> .



## Q) Explain Event Handling mechanism in Node.js

Node is used to build the back-end of web applications and provides an event-driven, non-blocking I/O model that makes it highly efficient for handling large amounts of data.

The Node.js event model does things differently from traditional event model. Instead of executing all the work for each request on individual threads, work is added to an event queue and then picked up by a single thread running an event loop. The event loop grabs the top item in the event queue, executes it, and then grabs the next item. When executing code that is no longer live or has blocking I/O, instead of calling the function directly, the function is added to the event queue along with a callback that is executed after the function completes. When all events on the Node.js event queue have been executed, the Node application terminates.

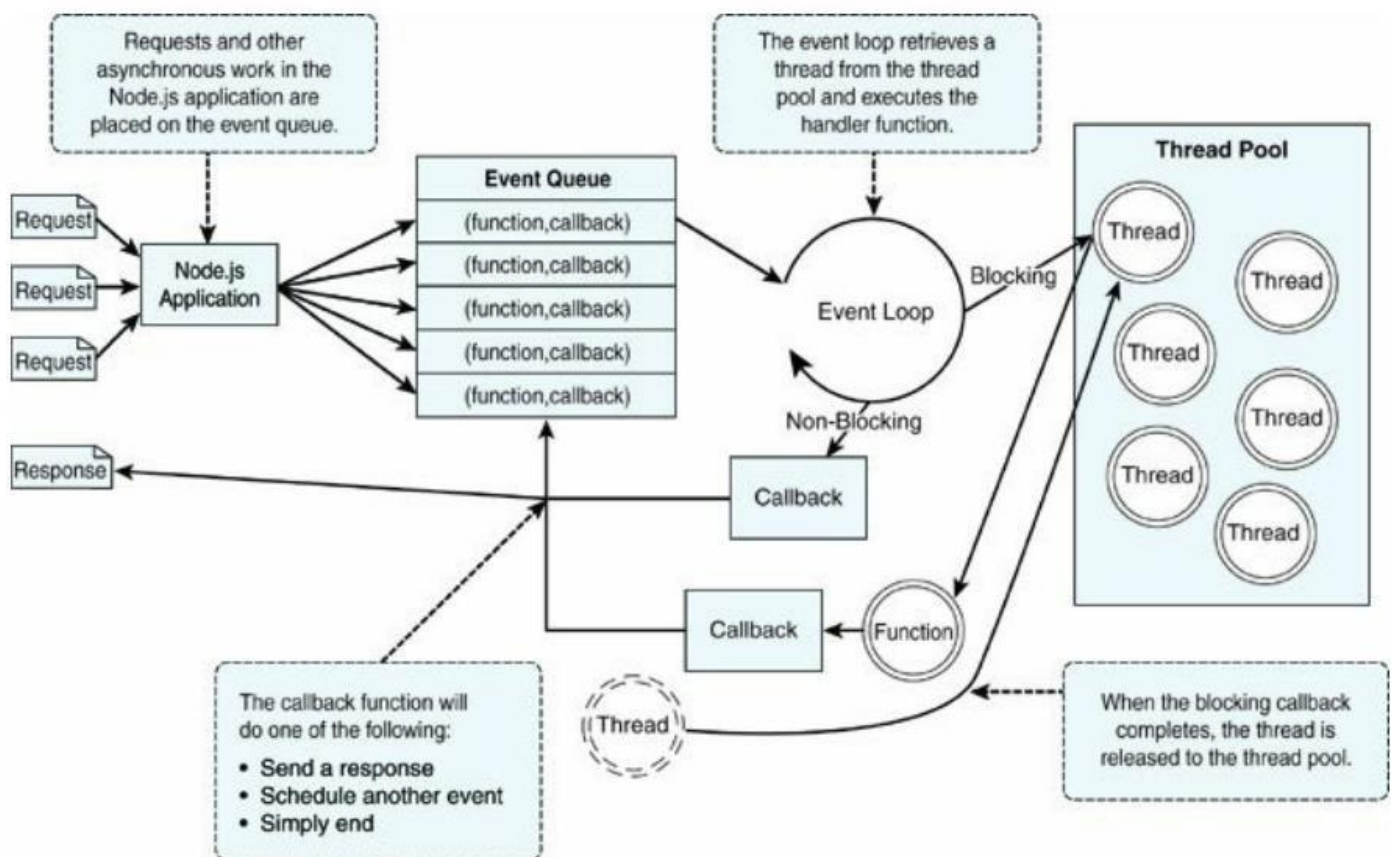


Fig. Event Handling in Node.js

We can then use the event model to schedule work on the event queue. In Node.js applications, work is scheduled on the event queue by passing a callback function using one of these methods:

- Make a call to one of the blocking I/O library calls such as writing to a file or connecting to a database.
- Add a built-in event listener to a built-in event such as an `http.request` or



server.connection.

- Create own event emitters and add custom listeners to them.
- Use the process.nextTick option to schedule work to be picked up on the next cycle of the event loop.
- Use timers to schedule work to be done after a particular amount of time or at periodic intervals.

Eg. Create any custom event as an example.

**Q)** Explain how to schedule/add work to Event Queue using Timers.

There are three types of timers you can implement in Node.js: timeout, interval, and immediate.

**i.** Delaying Work with Timeouts

- Timeout timers are used to delay work for a specific amount of time. When that time expires, the callback function is executed and the timer goes away
- Timeout timers are created using the setTimeout(callback, delayMilliseconds, [args]) method built into Node.js.

When you call setTimeout(), the callback function is executed after delayMilliseconds expires.

For example, the following executes myFunc() after 1 second:

```
setTimeout(myFunc, 1000);
```

The setTimeout() function returns a timer object ID. You can pass this ID to clearTimeout(timeoutId) at any time before the delayMilliseconds expires to cancel the timeout function.

Eg. Implementing a series of timeouts at various intervals

**Timer1.js**

```
function simpleTimeout(consoleTimer){
 console.timeEnd(consoleTimer);
}
console.time("twoSecond");
setTimeout(simpleTimeout, 2000, twoSecond);
console.time("oneSecond");
setTimeout(simpleTimeout, 1000, "oneSecond"); console.time("fiveSecond");
setTimeout(simpleTimeout, 5000, "fiveSecond"); console.time("50MilliSecond");
setTimeout(simpleTimeout, 50, "50MilliSecond");
```

- The **console.time()** method starts a timer you can use to track how long an operation takes. You give each timer a unique name, and may have up to

- 10,000 timers running on a given page.
- When you call `console.timeEnd()` with the same name, the browser will output the time, in milliseconds, that elapsed since the timer was started.

Output:

```
C:\Program Files\nodejs\node.exe .\Timer1.js
50MilliSecond: 50.341064453125 ms 50MilliSecond:
50.751ms
oneSecond: 1015.48388671875 ms
oneSecond: 1.016s
twoSecond: 2014.297119140625 ms twoSecond:
2.014s
fiveSecond: 5008.2470703125 ms
fiveSecond: 5.009s
```

## ii. Performing Periodic Work with Intervals

- Interval timers are used to perform work on a regular delayed interval. When the delay time expires, the callback function is executed and is then rescheduled for the delay interval again.
- Interval timers are created using the `setInterval(callback, delayMilliseconds, [args])` method built into Node.js.
- When you call `setInterval()`, the callback function is executed every interval after

`delayMilliseconds` has expired. For example, the following executes `myFunc()` every second:  
`setInterval(myFunc, 1000);`

Eg. Timer2.js

```
var x=0, y=0, z=0;
function displayValues(){
 console.log("X=%d; Y=%d; Z=%d", x, y, z);
}

function updateX(){
 x += 1;
}
function updateY(){
 y += 1;
}
function updateZ(){
 z += 1;
 displayValues();
}

setInterval(updateX, 500);
setInterval(updateY, 1000);
setInterval(updateZ, 2000);
```

Output:

```

C:\Program Files\nodejs\node.exe .\Timer2.js X=3;
Y=1; Z=1
X=7; Y=3; Z=2
X=11; Y=5; Z=3
X=15; Y=7; Z=4
X=19; Y=9; Z=5
X=23; Y=11; Z=6
X=27; Y=13;
Z=7

```

### iii. Performing Immediate Work with an Immediate Timer

- Immediate timers are used to perform work on a function as soon as the I/O event callbacks are executed, but before any timeout or interval events are executed. This allows you to schedule work to be done after the current events in the event queue are completed.
- Immediate timers are created using the `setImmediate(callback,[args])` method built into Node.js. When you call `setImmediate()`, the callback function is placed on the event queue and popped off once for each iteration through the eventqueue loop after I/O events have a chance to be called

### iv. Using nextTick to Schedule Work

- A useful method of scheduling work on the event queue is the `process.nextTick(callback)` function. This function schedules work to be run on the next cycle of the event loop. Unlike the `setImmediate()` method, `nextTick()` executes before the I/O events are fired.
- This can result in starvation of the I/O events, so Node.js limits the number of `nextTick()` events that can be executed each cycle through the event queue by the value of `process.maxTickDepth`, which defaults to 1000.

Eg. Timer3.js

```

var fs = require("fs");
fs.stat("nexttick.js", function() {
 console.log("nexttick.js Exists");
});

setImmediate(function() {
 console.log("Immediate Timer 1 Executed");
});

setImmediate(function() { console.log("Immediate
 Timer 2 Executed");
});

process.nextTick(function() {
 console.log("Next Tick 1 Executed");
});

process.nextTick(function() {

```

```
console.log("Next Tick 2 Executed");
});
```

Output:

```
C:\Program Files\nodejs\node.exe .\Timer3.js Next
Tick 1 Executed
Next Tick 2 Executed
Immediate Timer 1 Executed
Immediate Timer 2 Executed
nexttick.js Exists
```

**v. Dereferencing Timers from the Event Loop**

Often we do not want timer event callbacks to continue to be scheduled when they are the only events left in the event queue.

The `unref()` function available in the object returned by `setInterval` and `setTimeout` allows us to notify the event loop to not continue when these are the only events on the queue.

Eg.

```
myInterval = setInterval(myFunc);
myInterval.unref();
```

If for some reason later do not want the program to terminate if the interval function is the only event left on the queue, you can use the `ref()` function to rereference it: `myInterval.ref()`;

Q) Explain how to create a custom event in Node.js

In Node.js, events are a core concept that allows applications to respond to different types of actions or changes that occur within the application. An event is essentially a signal that something has happened, such as a user clicking a button, a file being read or written, or a network connection being established.

Event listeners are functions that are registered to listen for and respond to specific events. When an event occurs, all registered event listeners for that event are executed in the order they were registered. Event listeners can be added or removed dynamically, and multiple event listeners can be registered for the same event.

Events are emitted using an `EventEmitter` object. This object is included in the `events` module. The `emit(eventName, [args])` function triggers the `eventName` event and includes any arguments provided.

The following code snippet shows how to implement a simple event emitter: `var`

```
events = require('events');
var emitter = new events.EventEmitter();
emitter.emit("simpleEvent");
```

Adding Event Listeners to Objects

- **`addListener(eventName, callback)`**: Attaches the callback function to the object's listeners. Every time the `eventName` event is triggered, the callback function is placed in the event queue to be executed.
- **`.on(eventName, callback)`**: Same as `.addListener()`.

- **.once(eventName, callback):** Only the first time the eventName event is triggered, the callback function is placed in the event queue to be executed.
- **Removing Listeners from Objects:**
- **.listeners(eventName):** Returns an array of listener functions attached to the eventName event.
- **.setMaxListeners(n):** Triggers a warning if more than n listeners are added to an EventEmitter object. The default is 10.
- **.removeListener(eventName, callback):** Removes the callback function from the eventName event of the EventEmitter object.

Eg. 1:

```
var events = require('events');
var EventEmitter = new events.EventEmitter();

//Create an event handler: var
welcome = function () {
 console.log('Welcome to pvpsit');
}

var bye = function () { console.log('Good
 bye to pvpsit');
}

//Assign the eventhandler to an event:
eventEmitter.on('greet', welcome);
eventEmitter.on('greet', bye);

//Fire the 'greet' event: eventEmitter.emit('greet');

var listener_count = eventEmitter.listenerCount('greet'); console.log(listener_count + "
Listner(s) listening to greet event");
// Remove the binding of listner bye function
eventEmitter.removeListener('greet', bye);
console.log("listener bye removed..");
```

### Output:

```
node Event_Demo.js
Welcome to pvpsit
Good bye to pvpsit
2 Listner(s) listening to greet event listener
bye removed..
```

Eg. 2. Event1.js

```

var events = require('events');
function Account() {
 this.balance = 0;
 events.EventEmitter.call(this);
 this.deposit = function(amount){
 this.balance += amount;
 this.emit('balanceChanged');
 };

 this.withdraw = function(amount){ this.balance
 -= amount; this.emit('balanceChanged');
 };
}

Account.prototype.__proto__ = events.EventEmitter.prototype;
function displayBalance(){
 console.log("Account balance: $%d", this.balance);
}

function checkOverdraw()
{ if (this.balance < 0){
 console.log("Account overdrawn!!!");
}
}

function checkGoal(acc, goal){
 if (acc.balance > goal){
 console.log("Goal
 Achieved!!!");
 }
}

var account = new Account();
account.on("balanceChanged", displayBalance);
account.on("balanceChanged", checkOverdraw);
account.on("balanceChanged", function(){
 checkGoal(this, 1000);
});
account.deposit(220); account.deposit(320);
account.deposit(600);
account.withdraw(1200);

```

Output:

```

C:\Program Files\nodejs\node.exe .\Event1.js Account
balance: $220
Account balance: $540
Account balance: $1140
Goal Achieved!!!
Account balance: $-60
Account overdrawn!!!

```

Q) What is a callback? Explain different types of callbacks with suitable examples.

**Callback:** A callback is a function or piece of code that is passed as an argument to another function, with the intention of being called at some point during the execution of that function or method. A callback function can be defined with or without parameters.

Eg.

```
function sum(a, b, callback) {
 let result = a + b;
 callback();
 console.log(result)
}
```

```
function logResult()
{
 console.log('The sum is:');
}
```

```
sum(2, 3, logResult);
```

Output:

```
node callback0.js
```

```
The sum is:
```

```
5
```

Callback with parameters:

Eg. 1:

```
function add(a, b, callback) {
 let result = a + b;
 callback(result);
}
function logResult(sum) {
 console.log('The sum is %d',sum);
}
add(2, 3, logResult);
```

Output:

```
node callback1.0.js
```

```
The sum is 5
```

Eg. 2:

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

```
function CarShow() {
 events.EventEmitter.call(this);
 this.seeCar = function(make){
 this.emit('sawCar', make);
 };
}
```

```
CarShow.prototype.__proto__ = events.EventEmitter.prototype;
var show = new CarShow();
function logCar(make){
```

```

 console.log("Saw a " + make);
 }
 function logColorCar(make, color){
 console.log("Saw a %s %s", color, make);
 }
 show.on("sawCar", logCar);
 show.on("sawCar", function(make){
 var colors = ['red', 'blue', 'black'];
 var color = colors[Math.floor(Math.random()*3)];
 logColorCar(make, color);
 });
 show.seeCar("Ferrari"); show.seeCar("Porsche");
 show.seeCar("Bugatti");
 show.seeCar("Lamborghini"); show.seeCar("Aston
 Martin");

```

Output:

```

node callback1.js
Saw a Ferrari
Saw a black Ferrari
Saw a Porsche
Saw a red Porsche
Saw a Bugatti
Saw a red Bugatti
Saw a Lamborghini Saw
a red Lamborghini Saw a
Aston Martin Saw a red
Aston Martin

```

**Closure callback:** In Node.js, a closure callback is a function that has access to variables in its parent function scope, even after the parent function has returned. This is achieved through closure, which allows a function to "remember" its lexical scope.

Eg. 1:

```

function counter() {
 let count = 0;

 const incrementCount = function() {
 count++;
 console.log(`Count is now ${count}`);
 };

 return incrementCount;
}

const callback = counter();
callback(); // Output: 'Count is now 1'
callback(); // Output: 'Count is now 2'

```

Output:

```

node callback2.0.js
Count is now 1 Count
is now 2

```



Eg.2:

```
function logCar(logMsg, callback){
 process.nextTick(function() {
 callback(logMsg);
 });
}

var cars = ["Ferrari", "Porsche", "Bugatti"]; for
(var idx in cars){
 var message = "Saw a " + cars[idx];
 logCar(message, function(){ console.log("Normal Callback: " + message);
});
}
for (var idx in cars){
 var message = "Saw a " + cars[idx];
 (function(msg){
 logCar(msg, function(){ console.log("Closure
 Callback: " + msg);
 });
})(message);
}
```

Output:

```
node callback2.js
Normal Callback: Saw a Bugatti
Normal Callback: Saw a Bugatti
Normal Callback: Saw a Bugatti
Closure Callback: Saw a Ferrari
Closure Callback: Saw a Porsche
Closure Callback: Saw a Bugatti
```

The second loop also iterates over the cars array, but it uses a closure to pass the log message to the callback function.

**Chained Callback:** A chained callback is a series of callback functions that are executed one after another in a sequence. The output of one callback function is passed as input to the next callback function in the chain. `function add(a, b, callback) {`

```
 let sum = a + b;
 callback(sum);
}
```

```
function square(num, callback) { let
 result = num * num;
 callback(result);
}
```

```
function logResult(result) { console.log(`The
 final result is ${result}`);
}
```

```
add(2, 3, function(sum) {
```

```

 square(sum, function(result) {
 logResult(result);
 });
});

```

Output:

node callback3.0.js The  
final result is 25

Eg. 2:

```

function logCar(car, callback){
 console.log("Saw a %s", car);
 if(cars.length){
 process.nextTick(function(){
 callback();
 });
 }
}

function logCars(cars){
 var car = cars.pop();
 logCar(car, function(){
 logCars(cars);
 });
}

var cars = ["Ferrari", "Porsche", "Bugatti",
 "Lamborghini", "Aston Martin"]; logCars(cars);

```

Output:

node callback3.js  
Saw a Aston Martin  
Saw a Lamborghini  
Saw a Bugatti  
Saw a Porsche  
Saw a Ferrari

Q) Explain Buffer class in Node.js with an example.

Node provides Buffer class which provides instances to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap. Buffer class is a global class that can be accessed in an application without importing the buffer module.

```

var buf1 = new Buffer(100);
var buf2 = new Buffer(100);

var buf3 = new Buffer(26); for
(var i = 0 ; i < 26 ; i++) {
 buf3[i] = i + 97;
}
console.log("buffer 3:" + buf3.toString('ascii'));

len = buf1.write("Welcome to pvpsit");
console.log(" buf1 Octets written : " + len);
console.log("buffer 1:" + buf1.toString('utf-8'));

```

```

len = buf2.write("Welcome to it");
console.log("buf2 Octets written : "+ len); console.log("buffer
2:"+ buf2.toString('utf-8'));

var buf6 = Buffer.concat([buf1,buf2]);
console.log("buffer after concatinating buf1 and buf2: " + buf6.toString());

var buf4 = new Buffer(26) buf3.copy(buf4);
console.log("buffer4 content: " + buf4.toString());

var buf5 = buf4.slice(0,9);
console.log("buf4.slice(0,9): " + buf5.toString()+" and length is "+buf5.length);

var result = buf5.compare(buf4);
if(result < 0) {
 console.log(buf4 +" comes after " + buf5);
}else if(result == 0){
 console.log(buf4 +" is same as " + buf5);
}else {
 console.log(buf4 +" comes before " + buf5);
}

var json = buf5.toJSON(buf5);
console.log(json);

```

O/P:

PS D:\PVP\MWA\Lab\Files> node Event\_Demo.js

Welcome to pvpsit

Good bye to pvpsit

2 Listner(s) listening to greet event listener

bye removed..

PS D:\PVP\MWA\Lab\Files>

\* History restored

buffer 3:abcdefghijklmnopqrstuvwxy buf1

Octets written : 17

buffer 1:Welcome to pvpsit

buf2 Octets written : 13 buffer

2:Welcome to it

buffer after concatinating buf1 and buf2: Welcome to pvpsitWelcome to it buffer4

content: abcdefghijklmnopqrstuvwxy

buf4.slice(0,9): abcdefghi and length is 9

abcdefghijklmnopqrstuvwxy comes after abcdefghi

```

{
 type: 'Buffer',
 data: [
 97, 98, 99, 100,
 101, 102, 103, 104,
 105
]
}

```

**Writing to Buffers Syntax** Following is the syntax of the method to write into a Node Buffer: `buf.write(string[, offset][, length][, encoding])`

Parameters Here is the description of the parameters used: `string` - This is the string data to be written to buffer.

- `offset` - This is the index of the buffer to start writing at. Default value is 0.
- `length` - This is the number of bytes to write. Defaults to `buffer.length`.
- `encoding` - Encoding to use. 'utf8' is the default encoding.

Return Value This method returns the number of octets written. If there is not enough space in the buffer to fit the entire string, it will write a part of the string.

**Reading from Buffers Syntax** Following is the syntax of the method to read data from a Node Buffer: `buf.toString([encoding][, start][, end])`

Parameters Here is the description of the parameters used:

- `encoding` - Encoding to use. 'utf8' is the default encoding.
- `start` - Beginning index to start reading, defaults to 0.
- `end` - End index to end reading, defaults to complete buffer.

Return Value This method decodes and returns a string from buffer data encoded using the specified character set encoding.

**Convert Buffer to JSON Syntax** Following is the syntax of the method to convert a Node Buffer into JSON object:

`buf.toJSON()`

Return Value This method returns a JSON-representation of the Buffer instance.

**Concatenate Buffers Syntax** Following is the syntax of the method to concatenate Node buffers to a single Node Buffer:

`Buffer.concat(list[, totalLength])`

Parameters Here is the description of the parameters used:

- `list` - Array List of Buffer objects to be concatenated.
- `totalLength` - This is the total length of the buffers when concatenated.

Return Value This method returns a Buffer instance.

**Compare Buffers Syntax** Following is the syntax of the method to compare two Node buffers: `buf.compare(otherBuffer);`

Parameters Here is the description of the parameters used:

- `otherBuffer` - This is the other buffer which will be compared with `buf`.

Return Value Returns a number indicating whether it comes before or after or is the same as the `otherBuffer` in sort order.

**Copy Buffer Syntax** Following is the syntax of the method to copy a node buffer:

`buf.copy(targetBuffer[, targetStart][, sourceStart][, sourceEnd])` Parameters Here is the description of the parameters used:

- `targetBuffer` - Buffer object where buffer will be copied.
- `targetStart` - Number, Optional, Default: 0
- `sourceStart` - Number, Optional, Default: 0
- `sourceEnd` - Number, Optional, Default: `buffer.length`

Return Value No return value.

**Slice Buffer Syntax** Following is the syntax of the method to get a sub- buffer of a node buffer: `buf.slice([start][, end])`

Parameters Here is the description of the parameters used:

- `start` - Number, Optional, Default: 0

- end - Number, Optional, Default: buffer.length

Return Value Returns a new buffer which references the same memory as the old one

**Buffer Length Syntax** Following is the syntax of the method to get a size of a node buffer in bytes: buf.length;

Return Value Returns the size of a buffer in bytes.

Q) What are streams? Explain different types of streams with suitable examples.

Streams are objects that let you read data from a source or write data to a destination in continuous fashion. In Node.js, there are four types of streams:

- Readable - Stream which is used for read operation.
- Writable - Stream which is used for write operation.
- Duplex - Stream which can be used for both read and write operation.

Each type of Stream is an EventEmitter instance and throws several events at different instance of times. For example, some of the commonly used events are:

- data - This event is fired when there is data is available to read.
- end - This event is fired when there is no more data to read.
- error - This event is fired when there is any error receiving or writing data.
- finish - This event is fired when all the data has been flushed to underlying system.

Eg. Read Stream

```
var fs = require('fs');
var options = { encoding: 'utf8', flag: 'r' };
var fileReadStream = fs.createReadStream("grains.txt", options);
fileReadStream.on('data', function(chunk) {
 console.log('Grains: %s', chunk);
 console.log('Read %d bytes of data.', chunk.length);
});
fileReadStream.on("close", function(){
 console.log("File Closed.");
});
```

Eg. Write Stream

```
var fs = require('fs');
var grains = ['wheat', 'rice', 'oats'];

var options = { encoding: 'utf8', flag: 'w' };
var fileWriteStream = fs.createWriteStream("grains.txt", options);

fileWriteStream.on("close", function(){
 console.log("File Closed.");
});

while (grains.length){
 var data = grains.pop() + " ";
 fileWriteStream.write(data); console.log("Wrote: %s", data);
}
```

```

 }
 fileWriteStream.end();

 var options = { encoding: 'utf8', flag: 'r' };
 var fileReadStream = fs.createReadStream("grains.txt", options);

 fileReadStream.on('data', function(chunk) {
 console.log('Grains: %s', chunk);
 console.log('Read %d bytes of data.', chunk.length);
 });

```

Eg. Duplex Stream:

```

var stream = require('stream'); var
util = require('util');
util.inherits(Duplexer, stream.Duplex);
function Duplexer(opt) {
 stream.Duplex.call(this, opt);
 this.data = [];
}
Duplexer.prototype._read = function readItem(size) { var
chunk = this.data.shift();
 if(chunk == "stop"){
 this.push(null);
 } else {
 if(chunk){
 this.push(chunk);
 } else {
 setTimeout(readItem.bind(this), 500, size);
 }
 }
};
Duplexer.prototype._write = function(data, encoding, callback) {
 this.data.push(data);
 callback();
};
var d = new Duplexer(); d.on('data',
function(chunk){
 console.log('read: ', chunk.toString());
});
d.on('end', function(){ console.log('Message
Complete');
});
d.write("I think, ");
d.write("therefore "); d.write("I
am."); d.write("Rene
Descartes"); d.write("stop");

```

**Piping** is a mechanism where we provide the output of one stream as the input to another stream. It is normally used to get data from one stream and to pass the output of that stream to another stream. There is no limit on piping operations. Eg. To copy data from one file to another:

```

var fs = require("fs");
// Create a readable stream
var readerStream = fs.createReadStream('count.txt');
// Create a writable stream
var writerStream = fs.createWriteStream('c.txt');
// Pipe the read and write operations
// read input.txt and write data to output.txt
readerStream.pipe(writerStream);
console.log("Program Ended");

```

**Chaining** is a mechanism to connect the output of one stream to another stream and create a chain of multiple stream operations. It is normally used with piping operations.

Eg. Zlib for compressing a file

```

var fs = require("fs");
var zlib = require('zlib');
// Compress the file input.txt to input.txt.gz
fs.createReadStream('c.txt')
 .pipe(zlib.createGzip())
 .pipe(fs.createWriteStream('c.txt.gz'));

```

```

console.log("File Compressed.");

```

```

fs.createReadStream('c.txt.gz')
 .pipe(zlib.createGunzip())
 .pipe(fs.createWriteStream('c.txt'));

```

```

console.log("File Decompressed.");

```

Q) Write a program in Node.js to count no. of lines, words and characters in a given file.

```

const fs = require('fs');

// Function to count lines, words, and characters in a file
const countLinesWordsChars = (file) => {
 let lines = 0;
 let words = 0;
 let chars = 0;

 // Create a readable stream from the file
 const stream = fs.createReadStream(file, { encoding: 'utf8' });

 // Listen for 'data' event, which is emitted whenever data is read from the stream
 stream.on('data', (data) => {
 // Count lines by counting the number of newline characters
 lines += data.split("\n").length;

 // Count words by splitting the data by whitespace characters and filtering out empty strings
 words += data.split(/\s+/).filter(Boolean).length;
 });
}

```

```

-
 // Count characters by adding the length of the data
 chars += data.length;
 });

 // Listen for 'end' event, which is emitted when the end of the stream is reached
 stream.on('end', () => { console.log(`Number
 of lines: ${lines}`); console.log(`Number of
 words: ${words}`);
 console.log(`Number of characters: ${chars}`);
 });

 // Listen for 'error' event, which is emitted when an error occurs stream.on('error',
 (err) => {
 console.error(`Error reading file: ${err}`);
 });
};

// Call the function with the file name as argument
countLinesWordsChars('count.txt');

 Q) Write a program in Node.js to count no.of vowels, consonants, digits
 and special characters in a given file.
const fs = require('fs');

function countChars(filename) {
 const vowels = 'aeiouAEIOU';
 let vowelCount = 0;
 let consonantCount = 0;
 let digitCount = 0;
 let specialCharCount = 0;

 const stream = fs.createReadStream(filename, { encoding: 'utf8' });

 stream.on('data', (data) => { for
 (const char of data) {
 if (char.match(/[a-zA-Z]/)) { if
 (vowels.includes(char)) {
 vowelCount++;
 } else {
 consonantCount++;
 }
 } else if (char.match(/\d/)) {
 digitCount++;
 } else if (char.match(/\S/)) {
 specialCharCount++;
 }
 }
 });

 stream.on('end', () => { console.log(`Vowels:
 ${vowelCount}`);
 console.log(`Consonants: ${consonantCount}`);
 console.log(`Digits: ${digitCount}`);
 });

```



```

 console.log(`Special Characters: ${specialCharCount}`);
 });

 stream.on('error', (err) => { console.error(`Error
 reading file: ${err}`);
 });
}

//reading file from user const
file = process.argv[2]; if (file)
{
 countChars(file);
} else {
 console.error('Please provide a file name as an argument.');
```

### Q) Explain about Node File I/O with suitable examples.

The Node File System (fs) module can be imported using the following syntax:

var fs = require("fs") Synchronous  
vs Asynchronous

Every method in the fs module has synchronous as well as asynchronous forms. Asynchronous methods take the last parameter as the completion function callback and the first parameter of the callback function as error.

It is better to use an asynchronous method instead of a synchronous method, as the former never blocks a program during its execution, whereas the second one does.

Eg. Asynchronous read and write:

```

var fs = require('fs');
var fruitBowl = ['apple', 'orange', 'banana', 'grapes'];
function writeFruit(fd){
 if (fruitBowl.length){
 var fruit = fruitBowl.pop() + " ";
 fs.write(fd, fruit, null, null, function(err, bytes){ if
 (err){
 console.log("File Write Failed.");
 } else {
 console.log("Wrote: %s %dbytes", fruit, bytes);
 }
 });
 } else {
 fs.close(fd);
 }
}
fs.open('fruit.txt', 'w', function(err, fd){
 writeFruit(fd);
});
```

```

function readFruit(fd, fruits){ var
 buf = new Buffer(5); buf.fill();
 fs.read(fd, buf, 0, 5, null, function(err, bytes, data)
```

```

-
 { if (bytes > 0) {
 console.log("read %dbytes", bytes);
 fruits += data;
 readFruit(fd, fruits);
 } else {
 fs.close(fd);
 console.log ("Fruits: %s", fruits);
 }
 });
}

fs.open('fruit.txt', 'r', function(err, fd){
 readFruit(fd, "");
});

```

Eg. Synchronous Read and Write

```

var fs = require('fs');
var veggieTray = ['carrots', 'celery', 'olives']; fd
= fs.openSync('veggie.txt', 'w');
while (veggieTray.length){ veggie
= veggieTray.pop() + " ";
var bytes = fs.writeSync(fd, veggie, null, null); console.log("Wrote
%s %dbytes", veggie, bytes);
}
fs.closeSync(fd);

fd = fs.openSync('veggie.txt', 'r');
var veggies = "";
do {
 var buf = new Buffer(5);
 buf.fill();
 var bytes = fs.readSync(fd, buf, null, 5);
 console.log("read %dbytes", bytes);
 veggies += buf.toString();
} while (bytes > 0);
fs.closeSync(fd);
console.log("Veg g (to get output shown) ies: " + veggies);

```

**Table 6.1 Flags that define how files are opened**

Mode	Description
r	Open file for reading. An exception occurs if the file does not exist.
r+	Open file for reading and writing. An exception occurs if the file does not exist.
rs	Open file for reading in synchronous mode. This is not the same as forcing <code>fs.openSync()</code> . When used, the OS bypasses the local file system cache. Useful on NFS mounts because it lets you skip the potentially stale local cache. You should only use this flag if necessary because it can have a negative impact on performance.
rs+	Same as rs except the file is 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.
w+	Open file for reading and writing. The file is created if it does not exist or truncated if it exists.
wx+	Same as w+ but fails if path exists.
a	Open file for appending. The file is created if it does not exist.
ax	Same as a but fails if the path exists.
a+	Open file for reading and appending. The file is created if it does not exist.
ax+	Same as a+ but fails if the path exists.

Q) Write a program in Node.js to access file system

#### A. accessing file statistics

```
var fs = require('fs');
fs.stat('file_stats.js', function (err, stats) { if
(!err){
 console.log('stats: ' + JSON.stringify(stats, null, ' '));
 console.log(stats.isFile() ? "Is a File" : "Is not a File");
 console.log(stats.isDirectory() ? "Is a Folder" : "Is not a Folder");
 console.log(stats.isSocket() ? "Is a Socket" : "Is not a Socket");
 console.log(stats.isDirectory() ? "Is a Directory" : "Is not a Directory");
 stats.isBlockDevice();
 stats.isCharacterDevice();
 stats.isSymbolicLink(); //only lstat
 stats.isFIFO();
}
});
```

**Table 6.2 Attributes and methods of Stats objects for file system entries**

Attribute/Method	Description
<code>isFile()</code>	Returns <code>true</code> if the entry is a file
<code>isDirectory()</code>	Returns <code>true</code> if the entry is a directory
<code>isSocket()</code>	Returns <code>true</code> if the entry is a socket
<code>dev</code>	Specifies the device ID on which the file is located
<code>mode</code>	Specifies the access mode of the file
<code>size</code>	Specifies the number of bytes in the file
<code>blksize</code>	Specifies the block size used to store the file in bytes
<code>blocks</code>	Specifies the number of blocks the file is taking on disk
<code>atime</code>	Specifies the time the file was last accessed
<code>mtime</code>	Specifies the time the file was last modified
<code>ctime</code>	Specifies the time the file was created

b. To list files/directories in a given directory:

```
var fs = require('fs');
var Path = require('path');
function WalkDirs(dirPath){
 console.log(dirPath);
 fs.readdir(dirPath, function(err, entries){ for
 (var idx in entries){
 var fullPath = Path.join(dirPath, entries[idx]);
 (function(fullPath){
 fs.stat(fullPath, function (err, stats){ if
 (stats.isFile()){
 console.log(fullPath);
 } else if (stats.isDirectory()){
 WalkDirs(fullPath);
 }
 }
));
 })(fullPath);
}
});
```

```
WalkDirs("../Files");
```

C. To create a directory:

```
let fs = require('fs') fs.mkdir("../Files/folderA", function(err){
 console.log(err ? "Directory not created" : "Directory created.");
});
```

D. Listing Files:

```
fs.readdir(path, callback)
fs.readdirSync(path)
```

e. Deleting Files:

```
fs.unlink(path, callback)
```

```

fs.unlinkSync(path)
eg.
fs.unlink("new.txt", function(err){
console.log(err ? "File Delete Failed" : "File Deleted");
});

```

#### **F. Truncating Files:**

To truncate a file, use one of the following fs calls and pass in the number of bytes you want the file to contain when the truncation completes: fs.truncate(path, len, callback)

```
fs.truncateSync(path, len)
```

The truncateSync(path) returns true or false based on whether the file is successfully truncated. The asynchronous truncate() call passes an error value to the callback function if an error is encountered when truncating the file.

Eg.

```

fs.truncate("new.txt", function(err){
console.log(err ? "File Truncate Failed" : "File Truncated");
});

```

#### **G. Making and Removing Directories:**

```
fs.mkdir(path, [mode], callback)
```

```
fs.mkdirSync(path, [mode])
```

The mkdirSync(path) returns true or false based on whether the directory is successfully created. The asynchronous mkdir() call passes an error value to the callback function if an error is encountered when creating the directory. Eg.

```

let fs = require('fs')
fs.mkdir("../Files/folderA", function(err){
 console.log(err ? "Directory not created" : "Directory created.");
});

```

#### **Output:**

```

node CreateDir.js
Directory created.

```

#### **H. Delete Directories:**

```
fs.rmdir(path, callback)
```

```
fs.rmdirSync(path)
```

eg.

```

let fs = require('fs')
fs.rmdir("../Files/folderA", function(err){
 console.log(err ? "Directory not deleted": "Directory deleted.");
});

```

#### **I. Renaming Files and Directories:**

```
fs.rename(oldPath, newPath, callback)
```

```
fs.renameSync(oldPath, newPath)
```

The oldPath specifies the existing file or directory path, and the newPath specifies the new name. The renameSync(path) returns true or false based on whether the file or directory is successfully renamed. The asynchronous rename() call passes an error value to the callback function if an error is encountered when renaming the file or directory.

Eg.

```
fs.rename("old.txt", "new.txt", function(err){ console.log(err ? "Rename Failed" : "File Renamed"); });
```

```
fs.rename("testDir", "renamedDir", function(err){ console.log(err ? "Rename Failed" : "Folder Renamed"); });
```

#### J. Watching for File Changes:

the fs module provides a useful tool to watch a file and execute a callback function when the file changes. This can be useful if you want to trigger events to occur when a file is modified, but do not want to continually poll from your application directly. This does incur some overhead in the underlying OS, so you should use watches sparingly.

```
fs.watchFile(path, [options], callback)
```

When a file change occurs, the callback function is executed and passes a current and previous Stats object.

Eg.

```
fs.watchFile("log.txt", {persistent:true, interval:5000}, function (curr, prev) {
 console.log("log.txt modified at: " + curr.mtime);
 console.log("Previous modification was: " + prev.mtime);
});
```

Q) Explain Events and Methods available on HTTP ClientRequest and ServerResponse objects.

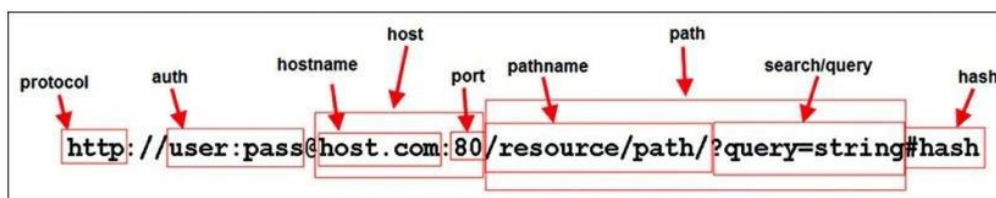


Figure 7.1 Basic components that can be included in a URL



**Table 7.1 Properties of the URL object**

Property	Description
href	This is the full URL string that was originally parsed.
protocol	The request protocol lowercased.
host	The full host portion of the URL including port information lowercased.
auth	The authentication information portion of a URL.
hostname	The hostname portion of the host lowercased.
port	The port number portion of the host.
pathname	The path portion of the URL including the initial slash if present.
search	The query string portion of the URL including the leading question mark.
path	The full path including the pathname and search.
query	This is either the parameter portion of the query string or a parsed object containing the query string parameters and values if the <code>parseQueryString</code> is set to <code>true</code> .
hash	The hash portion of the URL including the pound sign (#).

**The `http.ClientRequest` Object:**

The `ClientRequest` object is created internally when you call `http.request()` when building the HTTP client.

To implement a `ClientRequest` object, you use a call to `http.request()` using the following syntax: `http.request(options, callback)`

**Table 7.2 Options that can be specified when creating a `ClientRequest`**

Property	Description
<code>host</code>	The domain name or IP address of the server to issue the request. Defaults to <code>localhost</code> .
<code>hostname</code>	Same as <code>host</code> but preferred over <code>host</code> to support <code>url.parse()</code> .
<code>port</code>	Port of remote server. Defaults to 80.
<code>localAddress</code>	Local interface to bind for network connections.
<code>socketPath</code>	Unix Domain Socket (use one of <code>host:port</code> or <code>socketPath</code> ).
<code>method</code>	A string specifying the HTTP request method. For example, <code>GET</code> , <code>POST</code> , <code>CONNECT</code> , <code>OPTIONS</code> , etc. Defaults to <code>GET</code> .
<code>path</code>	A string specifying the requested resource path. Defaults to <code>/. </code> . It should also include the query string if any. For example:  <code>/book.html?chapter=12</code>
<code>headers</code>	An object containing request headers. For example:

```
{ 'content-length': '750', 'content-type': 'text/plain'
```

<code>auth</code>	Basic authentication in the form of <code>user:password</code> used to compute an <code>Authorization</code> header.
<code>agent</code>	Defines the Agent behavior. When an Agent is used, request defaults to <code>Connection:keep-alive</code> . Possible values are:  <code>undefined</code> (default): Uses global Agent.  <code>Agent object</code> : Uses specific Agent object.  <code>false</code> : Disables Agent behavior.



**Table 7.3 Events available on ClientRequest objects**

Property	Description
<code>response</code>	Emitted when a response to this request is received from the server. The callback handler receives back an <code>IncomingMessage</code> object as the only parameter.
<code>socket</code>	Emitted after a socket is assigned to this request.
<code>connect</code>	Emitted every time a server responds to a request that was initiated with a <code>CONNECT</code> method. If this event is not handled by the client, then the connection will be closed.
<code>upgrade</code>	Emitted when the server responds to a request that includes an <code>Upgrade</code> request in the headers.
<code>continue</code>	Emitted when the server sends a <code>100 Continue</code> HTTP response instructing the client to send the request body.

**Table 7.4 Methods available on ClientRequest objects**

Method	Description
<code>write(chunk, [encoding])</code>	Writes a <code>chunk</code> , <code>Buffer</code> or <code>String</code> object, of body data into the request. This allows you to stream data into the <code>Writable</code> stream of the <code>ClientRequest</code> object. If you stream the body data, you should include the <code>{ 'Transfer-Encoding', 'chunked' }</code> header option when you create the request. The <code>encoding</code> parameter defaults to <code>utf8</code> .
<code>end([data], [encoding])</code>	Writes the optional data out to the request body and then flushes the <code>Writable</code> stream and terminates the request.
<code>abort()</code>	Aborts the current request.
<code>setTimeout(timeout, [callback])</code>	Sets the socket timeout for the request.
<code>setNoDelay([noDelay])</code>	Disables the Nagle algorithm, which buffers data before sending it. The <code>noDelay</code> argument is a <code>Boolean</code> that is <code>true</code> for immediate writes and <code>false</code> for buffered writes.
<code>setSocketKeepAlive([enable], [initialDelay])</code>	Enables and disables the keep-alive functionality on the client request. The <code>enable</code> parameter defaults to <code>false</code> , which is disabled. The <code>initialDelay</code> parameter specifies the delay between the last data packet and the first keep-alive request.

The `http.ServerResponse` Object :

The `ServerResponse` object is created by the HTTP server internally when a request event is received. It is passed to the request event handler as the second argument. You use the `ServerRequest` object to formulate and send a response to the client. The `ServerResponse` implements a `Writable` stream, so it provides all the functionality of a `Writable` stream object. For example, you can use the `write()` method to write to it as well as pipe a `Readable` stream into it to write data back to the client.

**Table 7.5 Events available on `ServerResponse` objects**

Property	Description
<code>close</code>	Emitted when the connection to the client is closed prior to sending the <code>response.end()</code> to finish and flush the response.
<code>headersSent</code>	A Boolean that is <code>true</code> if headers have been sent; otherwise, <code>false</code> . This is read only.
<code>sendDate</code>	A Boolean that, when set to <code>true</code> , the <code>Date</code> header is automatically generated and sent as part of the response.
<code>statusCode</code>	Allows you to specify the response status code without having to explicitly write the headers. For example:  <pre>response.statusCode = 500;</pre>

**Table 7.6 Methods available on `ServerResponse` objects**

Method	Description
<code>writeContinue()</code>	Sends an HTTP/1.1 100 Continue message to the client requesting that the body data be sent.
<code>writeHead(statusCode, [reasonPhrase], [headers])</code>	Writes a response header to the request. The <code>statusCode</code> parameter is the three-digit HTTP response status code, for example, 200, 401, 500. The optional <code>reasonPhrase</code> is a string denoting the reason for the <code>statusCode</code> . The <code>headers</code> are the response headers object, for example:  <pre>response.writeHead(200, 'Success', {   'Content-Length': body.length,</pre>
	<pre>  'Content-Type': 'text/plain' });</pre>
<code>setTimeout(msecs, callback)</code>	Sets the socket timeout for the client connection in milliseconds along with a <code>callback</code> function to be executed if the timeout occurs.
<code>setHeader(name, value)</code>	Sets the value of a specific header where <code>name</code> is the HTTP header name and <code>value</code> is the header value.
<code>getHeader(name)</code>	Gets the value of an HTTP header that has been set in the response.
<code>removeHeader(name)</code>	Removes an HTTP header that has been set in the response.
<code>write(chunk, [encoding])</code>	Writes a chunk, <code>Buffer</code> or <code>String</code> object, of data out to the response <code>Writable</code> stream. This only writes data to the body portion of the response. The default encoding is <code>utf8</code> . This returns <code>true</code> if the data is written successfully or <code>false</code> if the data is written to user memory. If it returns <code>false</code> , then a <code>drain</code> event is emitted by the <code>Writable</code> stream when the buffer is free again.
<code>addTrailers(headers)</code>	Adds HTTP trailing headers to the end of the response.
<code>end([data], [encoding])</code>	Writes the optional data out to the response body and then flushes the <code>Writable</code> stream and finalizes the response.

- Q) Implement HTTP Services in Node.js to read user name from user and greet the user as the response.

**index.html:**

```
<!DOCTYPE html>
<html>
<head>
 <title>Greeting Form</title>
</head>
<body>
 <form action="/greet" method="POST">
 <label for="name">Enter your name:</label>
 <input type="text" id="name" name="name">
 <button type="submit">Submit</button>
 </form>
</body>
</html>
```

**index.js**

```
const http = require('http'); const
fs = require('fs'); const path =
require('path');

const server = http.createServer((req, res) =>
{ if (req.method === 'GET' && req.url === '/') {
 // Read the HTML file
 fs.readFile(path.join(__dirname, 'index.html'), 'utf8', (err, data) =>
 { if(err) {
 res.statusCode = 500; res.end('Internal
 Server Error');
 } else {
 res.setHeader('Content-Type', 'text/html');
 res.end(data);
 }
});
} else if (req.method === 'POST' && req.url === '/greet')
{ let data = "";

 // Collect the data from the request req.on('data',
 chunk => {
 data += chunk;
 });

 // Process the collected data
 req.on('end', () => {
 const name = new URLSearchParams(data).get('name');
 const greeting = `Hello, ${name}!`;

 res.setHeader('Content-Type', 'text/plain');

 res.statusCode = 200; res.end(greeting);
 });
} else {
```

```
res.statusCode = 404;
res.end('Not Found');
}
});
```

```
// Start the server on port 3000 server.listen(3000, ()
=> {
 console.log('Server listening on port 3000');
});
```

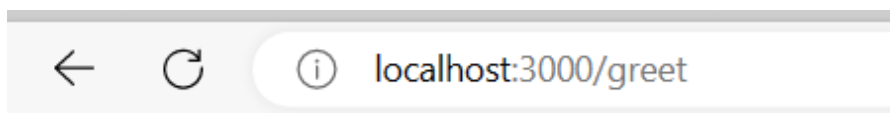
O/P:

node index.js

Server listening on port 3000



A screenshot of a web browser window. The address bar shows 'localhost:3000'. The page content includes the text 'Enter your name:' followed by a text input field containing 'chp' and a 'Submit' button.



A screenshot of a web browser window. The address bar shows 'localhost:3000/greet'. The page content displays 'Hello, chp!'.

Hello, chp!

## UNIT - V

### MongoDB:

Understanding NoSQL and MongoDB, Getting Started with MongoDB, Getting Started with MongoDB and Node.js, Manipulating MongoDB Documents from Node.js, Accessing MongoDB from Node.js, Using Mongoose for Structured Schema and Validation, Advanced MongoDB Concepts.

### Introduction:-

#### Data:-

Data is information such as facts and numbers used to analyze something or make decisions. Computer data is information in a form that can be processed by a computer.

#### Database:-

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS).

#### NoSQL:-

A database is a collection of structured data or information which is stored in a computer system and can be accessed easily. A database is usually managed by a Database Management System (DBMS).

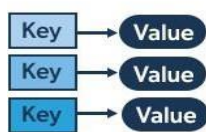
NoSQL is a non-relational database that is used to store the data in the nontabular form. NoSQL stands for Not only SQL. The main types are documents, key-value, wide-column, and graphs.

#### Types of NoSQL Database:

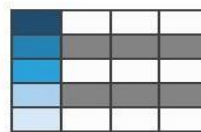
- Document-based databases
- Key-value stores
- Column-oriented databases
- Graph-based databases

## NoSQL

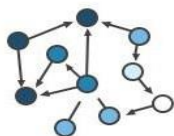
### Key-Value



### Column-Family



### Graph



### Document



### 1. Document-Based Database:

The document-based database is a nonrelational database. Instead of storing the data in rows and columns (tables), it uses the documents to store the data in the database. A document database stores data in JSON, BSON, or XML documents.

Documents can be stored and retrieved in a form that is much closer to the data objects used in applications which means less translation is required to use these data in the applications. In the

Document database, the particular elements can be accessed by using the index value that is assigned for faster querying.

Collections are the group of documents that store documents that have similar contents. Not all the documents are in any collection as they require a similar schema because document databases have a flexible schema.

Key features of documents database:

- Flexible schema: Documents in the database has a flexible schema. It means the documents in the database need not be the same schema.
- Faster creation and maintenance: the creation of documents is easy and minimal maintenance is required once we create the document.
- No foreign keys: There is no dynamic relationship between two documents so documents can be independent of one another. So, there is no requirement for a foreign key in a document database.
- Open formats: To build a document we use XML, JSON, and others.

## **2. Key-Value Stores:**

A key-value store is a nonrelational database. The simplest form of a NoSQL database is a key-value store. Every data element in the database is stored in key-value pairs. The data can be retrieved by using a unique key allotted to each element in the database. The values can be simple data types like strings and numbers or complex objects.

A key-value store is like a relational database with only two columns which is the key and the value.

Key features of the key-value store:

- Simplicity.
- Scalability.
- Speed.

## **3.Column Oriented Databases:**

A column-oriented database is a non-relational database that stores the data in columns instead of rows. That means when we want to run analytics on a small number of columns, you can read those columns directly without consuming memory with the unwanted data.

Columnar databases are designed to read data more efficiently and retrieve the data with greater speed. A columnar database is used to store a large amount of data. Key features of columnar oriented database:

- Scalability.
- Compression.
- Very responsive.

## **4. Graph-Based databases:**

Graph-based databases focus on the relationship between the elements. It stores the data in the form of nodes in the database. The connections between the nodes are called links or relationships.

Key features of graph database:

- In a graph-based database, it is easy to identify the relationship between the data by using the links.
- The Query's output is real-time results.



- The speed depends upon the number of relationships among the database elements.
- Updating data is also easy, as adding a new node or edge to a graph database is a straightforward task that does not require significant schema changes.

## What is MongoDB?

MongoDB is a **document-oriented** NoSQL database system that provides high scalability, flexibility, and performance. Unlike standard relational databases, MongoDB stores data in a JSON document structure form. This makes it easy to operate with dynamic and unstructured data and MongoDB is an open-source and cross-platform database System.

## Why Use MongoDB?

Document Oriented Storage – Data is stored in the form of JSON documents.

- **Index on any attribute:** Indexing in MongoDB allows for faster data retrieval by creating a searchable structure on selected attributes, optimizing query performance.
- **Replication and high availability:** MongoDB's replica sets ensure data redundancy by maintaining multiple copies of the data, providing fault tolerance and continuous availability even in case of server failures.
- **Auto-Sharding:** Auto-sharding in MongoDB automatically distributes data across multiple servers, enabling horizontal scaling and efficient handling of large datasets.
- **Big Data and Real-time Application:** When dealing with massive datasets or applications requiring real-time data updates, MongoDB's flexibility and scalability prove advantageous.
- **Rich queries:** MongoDB supports complex queries with a variety of operators, allowing you to retrieve, filter, and manipulate data in a flexible and powerful manner.
- **Fast in-place updates:** MongoDB efficiently updates documents directly in their place, minimizing data movement and reducing write overhead.
- **Professional support by MongoDB:** MongoDB offers expert technical support and resources to help users with any issues or challenges they may encounter during their database operations.

Internet of Things (IoT) Applications: **Storing and analyzing sensor data with its diverse formats often aligns well with MongoDB's document structure.**

## Where do we use MongoDB?

MongoDB is preferred over RDBMS in the following scenarios:

- **Big Data:** If you have huge amount of data to be stored in tables, think of MongoDB before RDBMS databases. MongoDB has built-in solution for partitioning and sharding your database.
- **Unstable Schema:** Adding a new column in RDBMS is hard whereas MongoDB is schema-less. Adding a new field does not effect old documents and will be very easy.
- **Distributed data** Since multiple copies of data are stored across different servers, recovery of data is instant and safe even if there is a hardware failure.

### Language Support by MongoDB:

MongoDB currently provides official driver support for all popular programming languages like C, C++, Rust, C#, Java, Node.js, Perl, PHP, Python, Ruby, Scala, Go, and Erlang.



## Features of MongoDB –

- **Schema-less Database:** It is the great feature provided by the MongoDB. A Schema-less database means one collection can hold different types of documents in it. Or in other words, in the MongoDB database, a single collection can hold multiple documents and these documents may consist of the different numbers of fields, content, and size. It is not necessary that the one document is similar to another document like in the relational databases. Due to this cool feature, MongoDB provides great flexibility to databases.
- **Document Oriented:** In MongoDB, all the data stored in the documents instead of tables like in RDBMS. In these documents, the data is stored in fields(key-value pair) instead of rows and columns which make the data much more flexible in comparison to RDBMS. And each document contains its unique object id.
- **Indexing:** In MongoDB database, every field in the documents is indexed with primary and secondary indices this makes easier and takes less time to get or search data from the pool of the data. If the data is not indexed, then database search each document with the specified query which takes lots of time and not so efficient.
- **Scalability:** MongoDB provides horizontal scalability with the help of sharding. Sharding means to distribute data on multiple servers, here a large amount of data is partitioned into data chunks using the shard key, and these data chunks are evenly distributed across shards that reside across many physical servers. It will also add new machines to a running database.
- **Replication:** MongoDB provides high availability and redundancy with the help of replication, it creates multiple copies of the data and sends these copies to a different server so that if one server fails, then the data is retrieved from another server.
- **Aggregation:** It allows to perform operations on the grouped data and get a single result or computed result. It is similar to the SQL GROUPBY clause. It provides three different aggregations i.e, aggregation pipeline, map-reduce function, and single-purpose aggregation methods
- **High Performance:** The performance of MongoDB is very high and data persistence as compared to another database due to its features like scalability, indexing, replication, etc.

## Advantages of MongoDB :

- It is a schema-less NoSQL database. You need not to design the schema of the database when you are working with MongoDB.
- It does not support join operation.
- It provides great flexibility to the fields in the documents.
- It contains heterogeneous data.
- It provides high performance, availability, scalability.
- It supports Geospatial efficiently.
- It is a document oriented database and the data is stored in BSON documents.
- It also supports multiple document ACID transition(string from MongoDB 4.0).
- It does not require any SQL injection.
- It is easily integrated with Big Data Hadoop

## Disadvantages of MongoDB :

- It uses high memory for data storage.
- You are not allowed to store more than 16MB data in the documents.
- The nesting of data in BSON is also limited you are not allowed to nest data more than 100 levels.

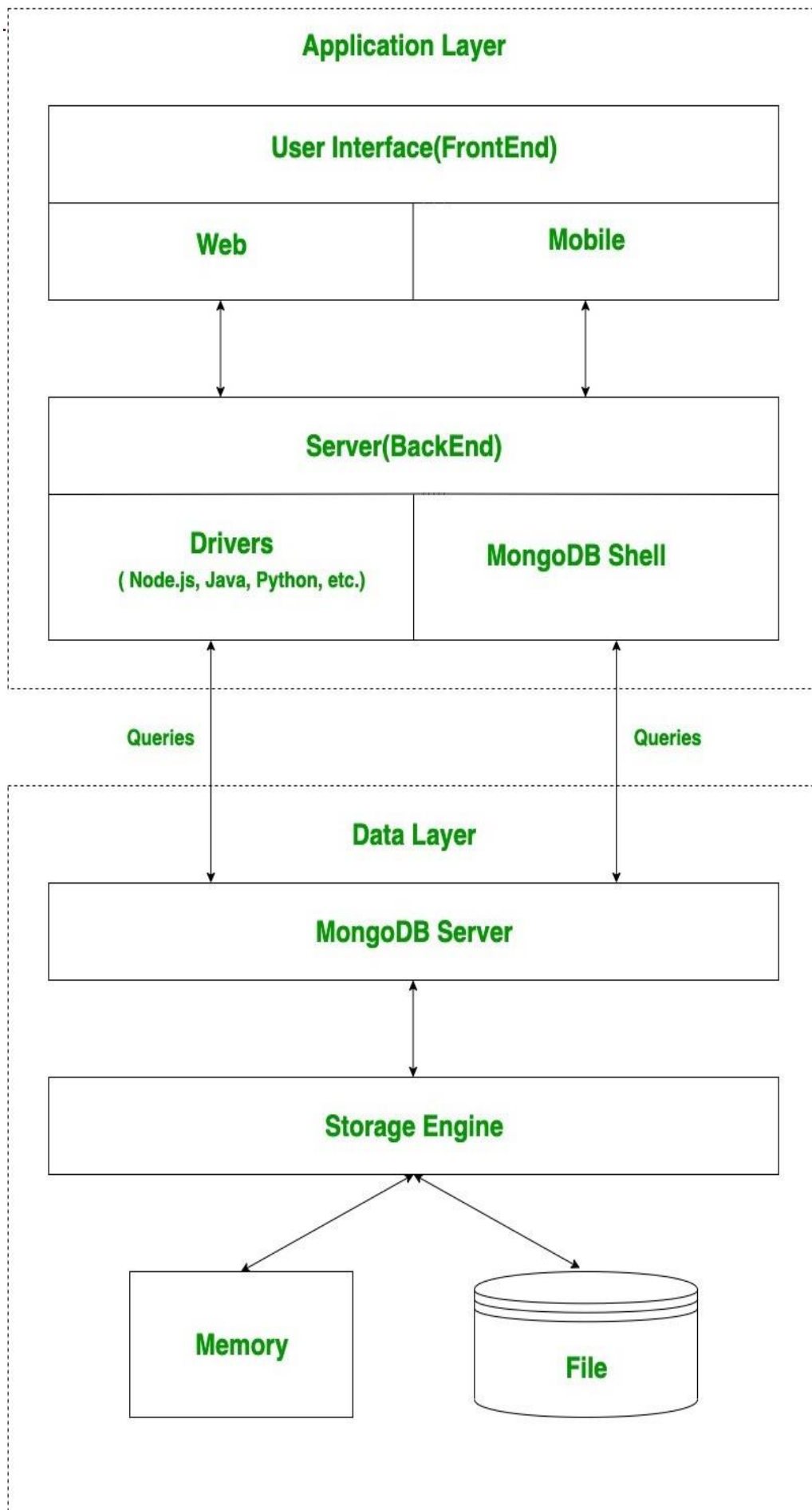
## How MongoDB works ?

MongoDB is an open-source document-oriented database. It is used to store a larger amount of data and also allows you to work with that data. MongoDB is not based on the table-like relational database structure but provides an altogether different mechanism for storage and retrieval of data, that's why known as NoSQL database. Here, the term 'NoSQL' means 'non- relational'. The format of storage is called BSON ( similar to JSON format).

Now, let's see how actually this MongoDB works? But before proceeding to its working, first, let's discuss some important parts of MongoDB –

- **Drivers:** Drivers are present on your server that are used to communicate with MongoDB. The drivers support by the MongoDB are C, C++, C#, and .Net, Go, Java, Node.js, Perl, PHP, Python, Motor, Ruby, Scala, Swift, Mongoid.
- **MongoDB Shell:** MongoDB Shell or mongo shell is an interactive JavaScript interface for MongoDB. It is used for queries, data updates, and it also performs administrative operations.
- **Storage Engine:** It is an important part of MongoDB which is generally used to manage how data is stored in the memory and on the disk. MongoDB can have multiple search engines. You are allowed to use your own search engine and if you don't want to use your own search engine you can use the default search engine, known as *WiredTiger Storage Engine* which is an excellent storage engine, it efficiently works with your data like reading, writing, etc.

**Working of MongoDB** –The following image shows how the MongoDB works:



## MongoDB work in two layers –

- **Application Layer** and
- **Data layer**

**Application Layer** is also known as the **Final Abstraction Layer**, it has two-parts, first is a **Frontend (User Interface)** and the second is **Backend (server)**. The frontend is the place where the user uses MongoDB with the help of a Web or Mobile. This web and mobile include web pages, mobile applications, android default applications, IOS applications, etc. The backend contains a server which is used to perform server-side logic and also contain drivers or mongo shell to interact with MongoDB server with the help of queries.

These queries are sent to the MongoDB server present in the **Data Layer**. Now, the MongoDB server receives the queries and passes the received queries to the storage engine. MongoDB server itself does not directly read or write the data to the files or disk or memory. After passing the received queries to the storage engine, the storage engine is responsible to read or write the data in the files or memory basically it manages the data.

**MongoDB**, the most popular NoSQL database, is an open-source document-oriented database. The term ‘NoSQL’ means ‘non-relational’. It means that MongoDB isn’t based on the table- like relational database structure but provides an altogether different mechanism for storage and retrieval of data. This format of storage is called BSON ( similar to JSON format).

A simple MongoDB document Structure:

```
{
 title: 'Geeksforgeeks',
 by: 'Harshit Gupta',
 url: 'https://www.geeksforgeeks.org',
 type: 'NoSQL'
}
```

SQL databases store data in tabular format. This data is stored in a predefined data model which is not very much flexible for today’s real-world highly growing applications. **Modern applications are more networked, social and interactive than ever.** Applications are storing more and more data and are accessing it at higher rates.

Relational Database Management System(RDBMS) is **not the correct choice when it comes to handling big data by the virtue of their design since they are not horizontally scalable.** If the database runs on a single server, then it will reach a scaling limit. NoSQL databases are more scalable and provide superior performance. MongoDB is such a NoSQL database that scales by adding more and more servers and increases productivity with its flexible document model.

Getting Started

After you install MongoDB, you can see all the installed file inside C:\ProgramFiles\MongoDB\ (default

**mongo:** The Command Line Interface to interact with the db.  
**mongod:** This is the database. Sets up the server.  
**mongodump:** It dumps out the Binary of the Database(BSON)  
**mongoexport:** Exports the document to Json, CSV format  
**mongoimport:** To import some data into the DB.  
**mongorestore:** to restore anything that you've exported.  
**mongostat:** Statistics of databases

location). In the C:\Program Files\MongoDB\Server\3.2\bin directory, there are a bunch of executables and a short- description about them would be:

### **Database, Collection and Documents:-**

#### **Database**

- Database is a container for collections.
- Each database gets its own set of files.
- A single MongoDB server can has multiple databases.

#### **Collection**

- Collection is a group of documents.
- Collection is equivalent to RDBMS table.
- A collection consist inside a single database.
- Collections do not enforce a schema.
- A Collection can have different fields within a Documents.

•

#### **Document:-**

---

A document database has information retrieved or stored in the form of a document or other words semi-structured database. Since they are non-relational, so they are often referred to as NoSQL data.

The document database fetches and accumulates data in forms of key-value pairs but here, the values are called as Documents. A document can be stated as a complex data structure.

Document here can be a form of text, arrays, strings, JSON, XML, or any such format. The use of nested documents is also very common. It is very effective as most of the data created is usually in the form of JSON and is unstructured.

C1	C2	C3

Relational Data Model



Document Store Model

Consider the below example that shows a sample database stored in both Relational and Document Database

## RELATIONAL

ID	first_name	last_name	cell	city	year_of_birth	location_x	location_y
1	'Mary'	'Jones'	'516-555-2048'	'Long Island'	1986	'-73.9876'	'40.7574'

ID	user_id	profession
10	1	'Developer'
11	1	'Engineer'

ID	user_id	name	version
20	1	'MyApp'	1.0.4
21	1	'DocFinder'	2.5.7

ID	user_id	make	year
30	1	'Bentley'	1973
31	1	'Rolls Royce'	1965

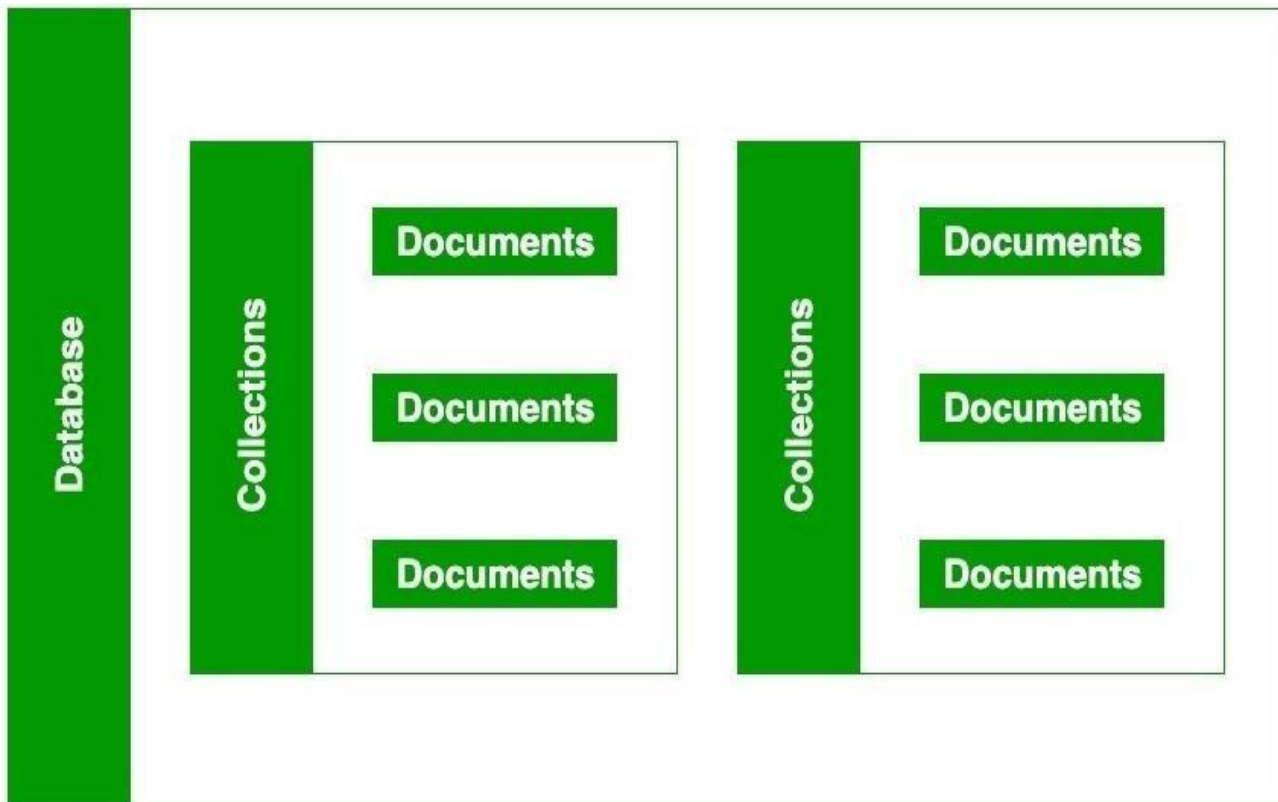
## DOCUMENT

```
first_name: "Mary",
last_name: "Jones",
cell: "516-555-2048",
city: "Long Island",
year_of_birth: 1986,
location: {
 type: "Point",
 coordinates: [-73.9876, 40.7574]
},
profession: ["Developer", "Engineer"],
apps: [
 { name: "MyApp",
 version: 1.0.4 },
 { name: "DocFinder",
 version: 2.5.7 }
],
cars: [
 { make: "Bentley",
 year: 1973 },
 { make: "Rolls Royce",
 year: 1965 }
]
```

### How it works ?

Now, we will see how actually thing happens behind the scene. As we know that MongoDB is a database server and the data is stored in these databases. Or in other words, MongoDB environment gives you a server that you can start and then create multiple databases on it using MongoDB.

Because of its NoSQL database, the data is stored in the collections and documents. Hence the database, collection, and documents are related to each other as shown below:

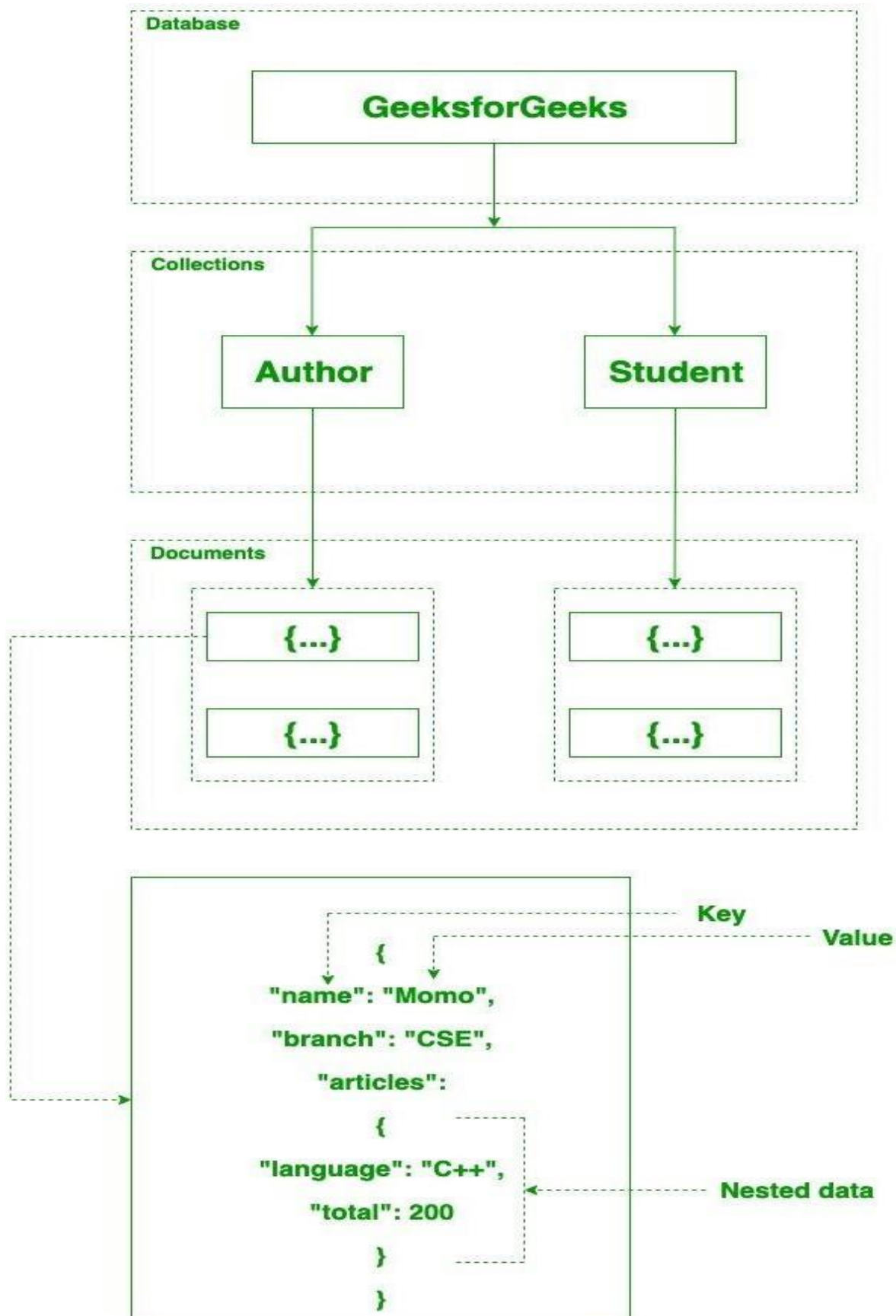


- The MongoDB database contains collections just like the MYSQL database contains tables. You are allowed to create multiple databases and multiple collections.
- Now inside of the collection we have documents. These documents contain the data we want to store in the MongoDB database and a single collection can contain multiple documents and you are schema-less means it is not necessary that one document is similar to another.
- The documents are created using the fields. Fields are key-value pairs in the documents, it is just like columns in the relation database. The value of the fields can be of any BSON data types like double, string, boolean, etc.
- The data stored in the MongoDB is in the format of BSON documents. Here, BSON stands for Binary representation of JSON documents. Or in other words, in the backend, the MongoDB server converts the JSON data into a binary form that is known as BSON and this BSON is stored and queried more efficiently.
- In MongoDB documents, you are allowed to store nested data. This nesting of data allows you to create complex relations between data and store them in the same document which makes the working and fetching of data extremely efficient as compared to SQL. In SQL, you need to write complex joins to get the data from table 1 and table 2. The maximum size of the BSON document is 16MB.

**NOTE:** In MongoDB server, you are allowed to run multiple databases.

For example, we have a database named GeeksforGeeks. Inside this database, we have two collections and in these collections we have two documents. And in these documents we store our data in the form of fields. As shown in the below image:





## **\_ How mongoDB is different from RDBMS ?**

Some major differences in between MongoDB and the RDBMS are as follows:

<b>MongoDB</b>	<b>RDBMS</b>
<b>It is a non-relational and document- oriented database.</b>	<b>It is a relational database.</b>
<b>It is suitable for hierarchical data storage.</b>	<b>It is not suitable for hierarchical data storage.</b>
<b>It has a dynamic schema.</b>	<b>It has a predefined schema.</b>
<b>It centers around the CAP theorem (Consistency, Availability, and Partition tolerance).</b>	<b>It centers around ACID properties (Atomicity, Consistency, Isolation, and Durability).</b>
<b>In terms of performance, it is much faster than RDBMS.</b>	<b>In terms of performance, it is slower than MongoDB.</b>

## Install MongoDB

- **There are 3 ways to install and use MongoDB**

1. Community Server(free and open source. after download use local system)
2. Visual Studio Extension
3. MongoDB Atlas(cloud hosted DB offered by MONGO DB company)

### 1. Let's install MongoDB on our machines(Windows)

- Visit official website: <http://mongodb.com>
- Download the latest stable version from Community Server
- The Community server will also install the following apps
  - a. Community Server
  - b. Compass-GUI Tool for MongoDB

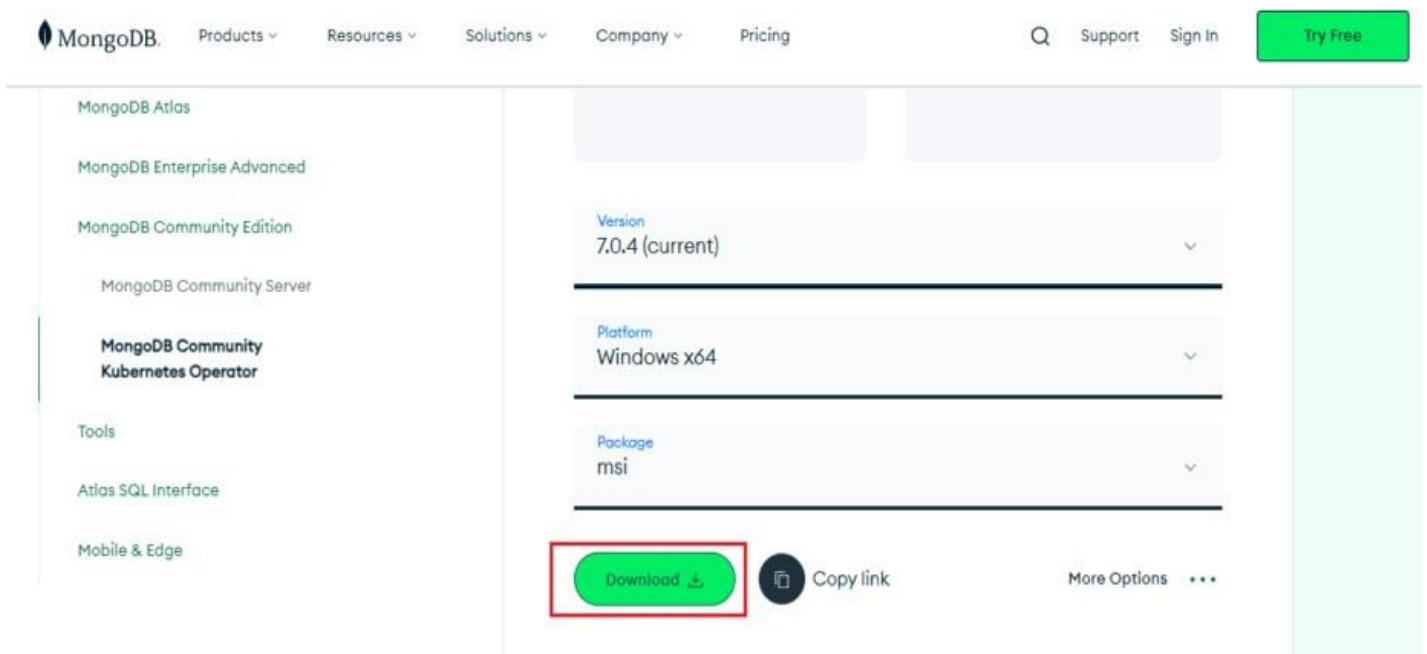
### Install MongoDB on Windows using MSI

#### Requirements to Install MongoDB on Windows

- **MongoDB 4.4** and later only **support 64-bit versions** of Windows.
- MongoDB 7.0 Community Edition supports the following **64-bit versions** of Windows on x86\_64 architecture:
  - **Windows Server 2022**
  - **Windows Server 2019**
  - **Windows 11**

To install MongoDB on windows, first, download the MongoDB server and then install the MongoDB shell. The Steps below explain the installation process in detail and provide the required resources for the smooth **download and install MongoDB**.

**Step 1:** Go to the [MongoDB Download Center](#) to download the MongoDB Community Server.



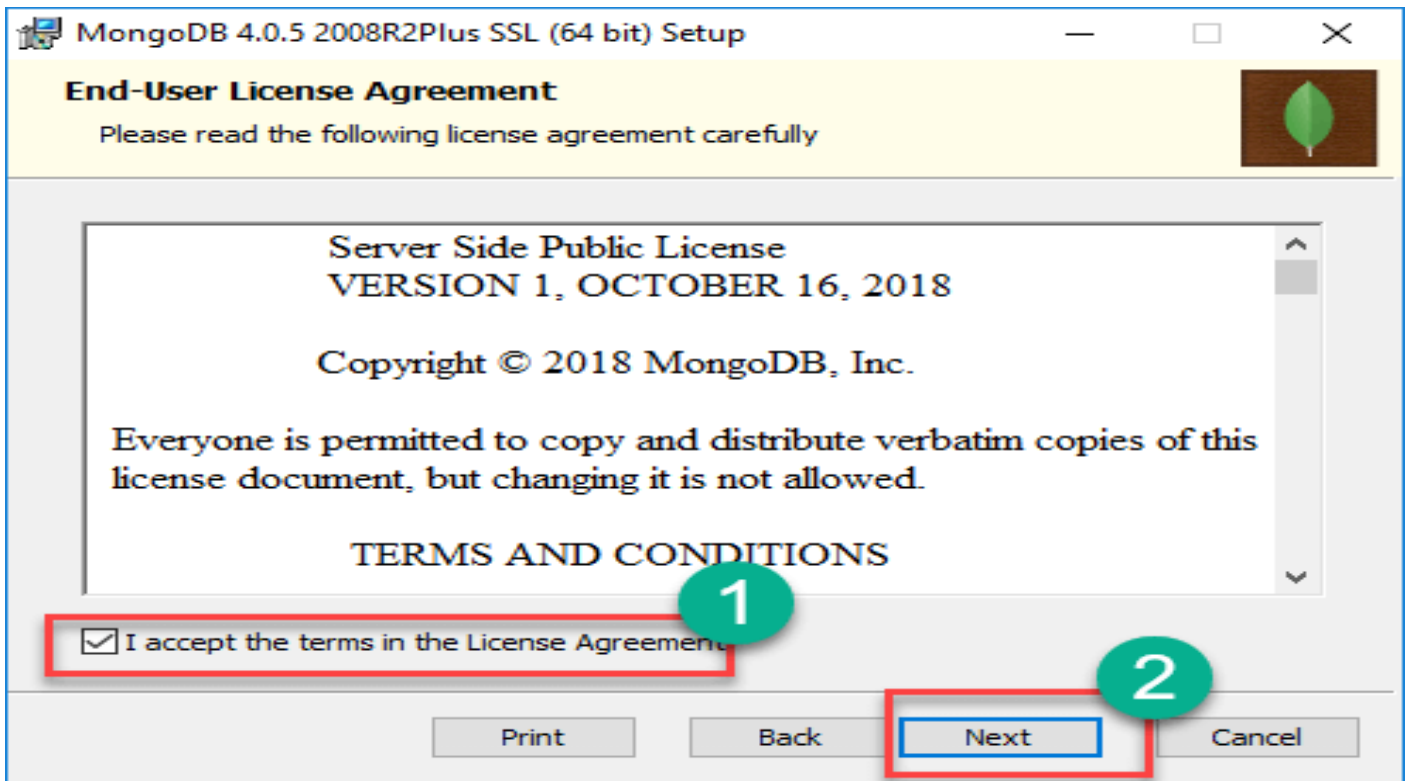
Here, You can select any version, Windows, and package according to your requirement. For Windows, we need to choose:

- Version: 7.0.4
- OS: Windows x64
- Package: msi

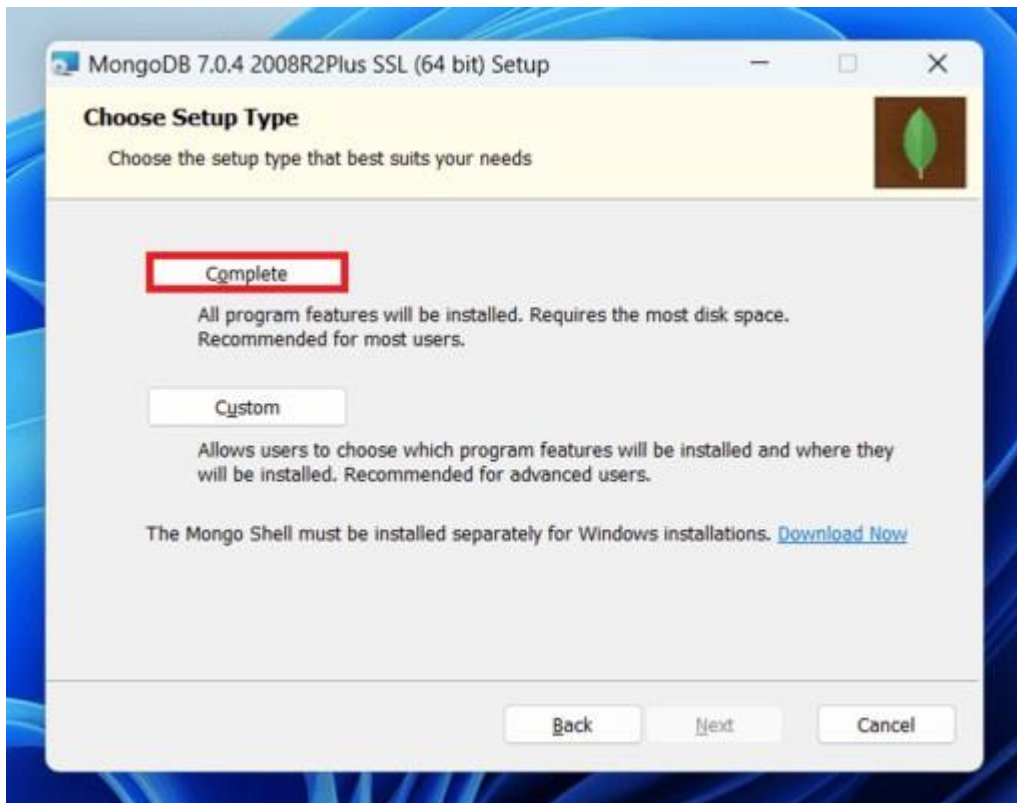
**Step 2:** When the download is complete open the msi file and click the *next button* in the startup screen:



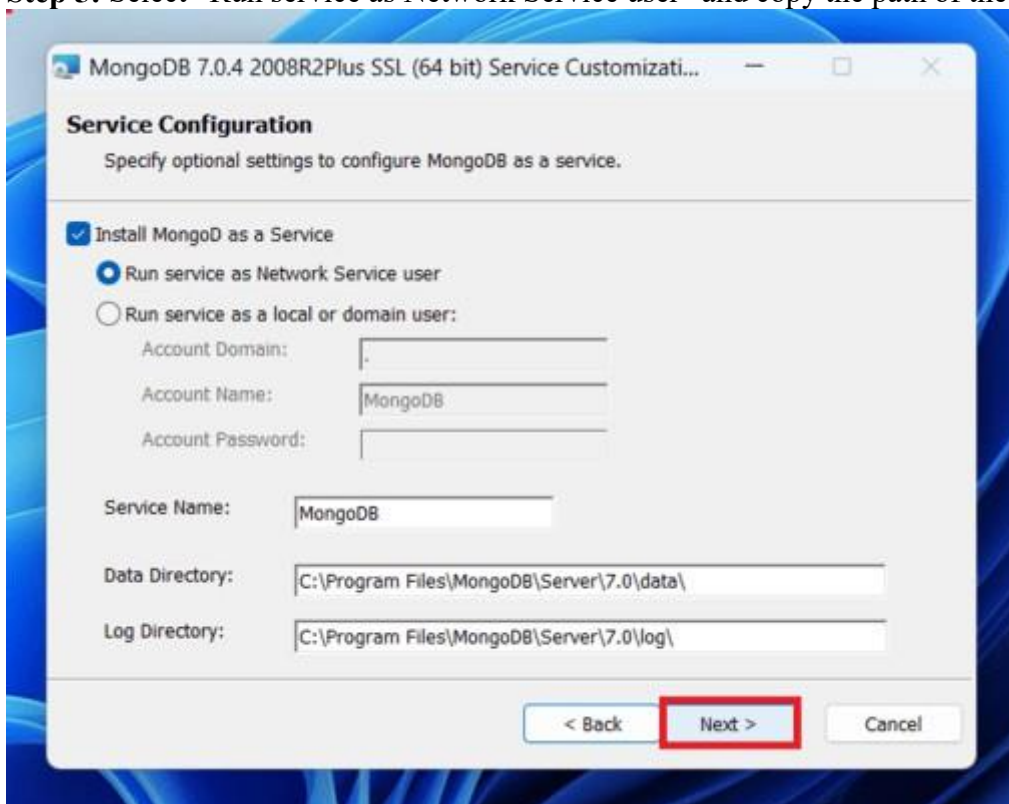
**Step 3:** Now accept the End-User License Agreement and click the next button:



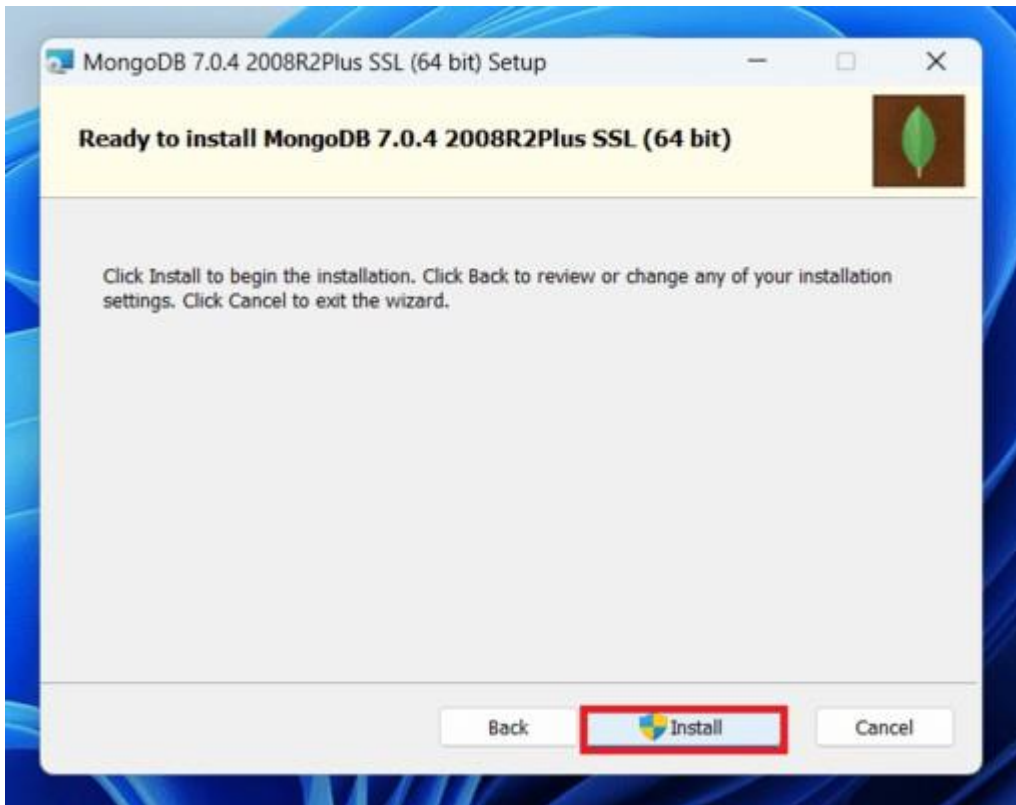
**Step 4:** Now select the *complete option* to install all the program features. Here, if you can want to install only selected program features and want to select the location of the installation, then use the *Custom option*:



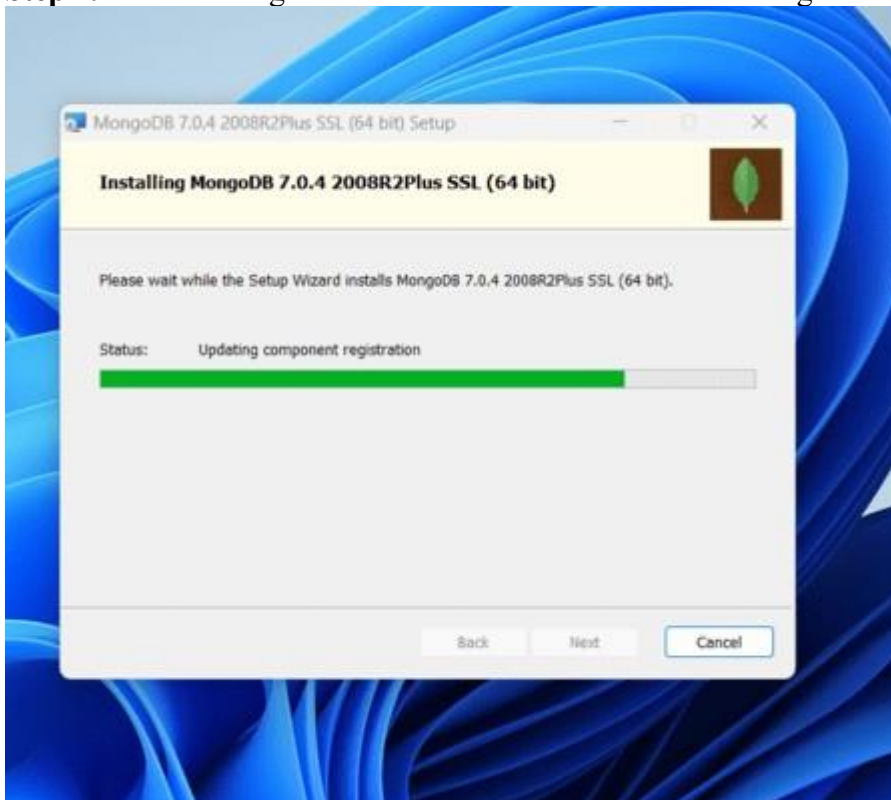
**Step 5:** Select “Run service as Network Service user” and copy the path of the data directory. Click Next:



**Step 6:** Click the *Install button* to start the MongoDB installation process:



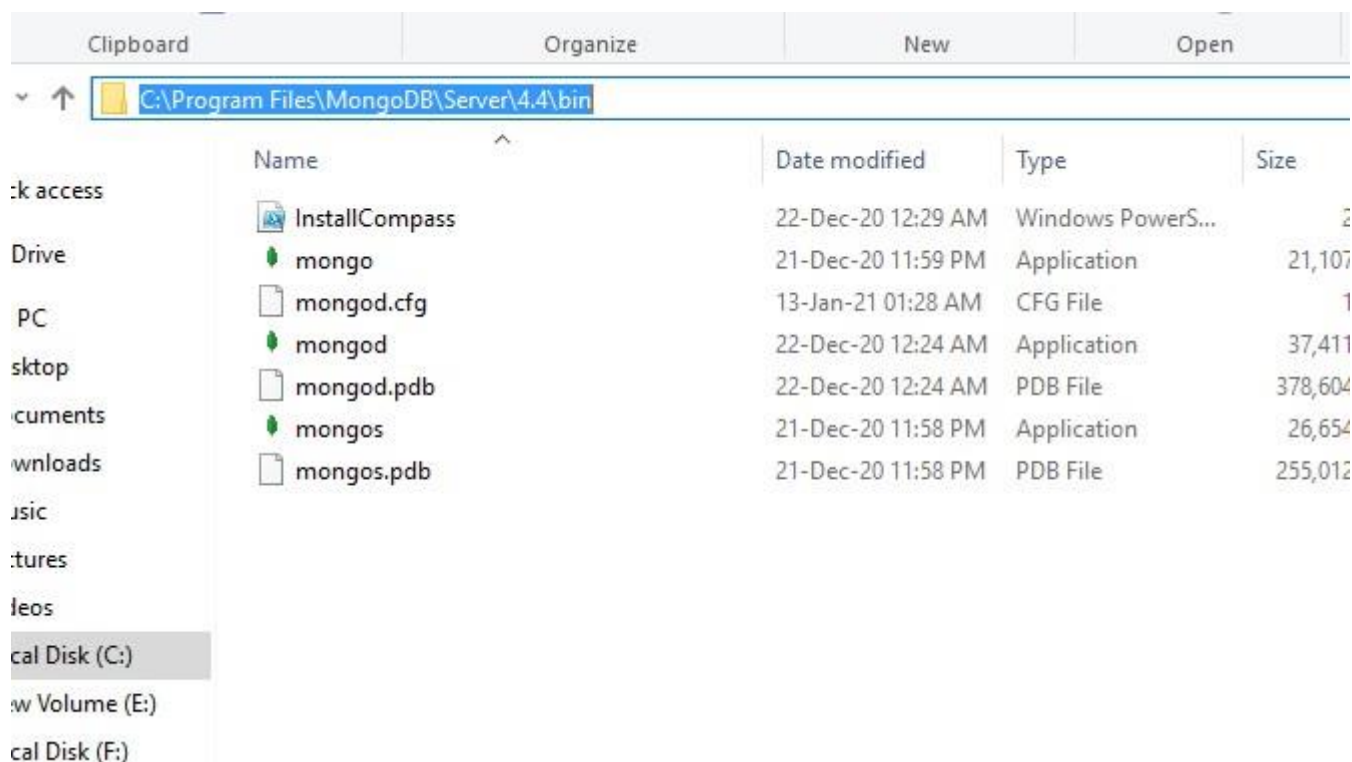
**Step 7:** After clicking on the install button installation of MongoDB begins:



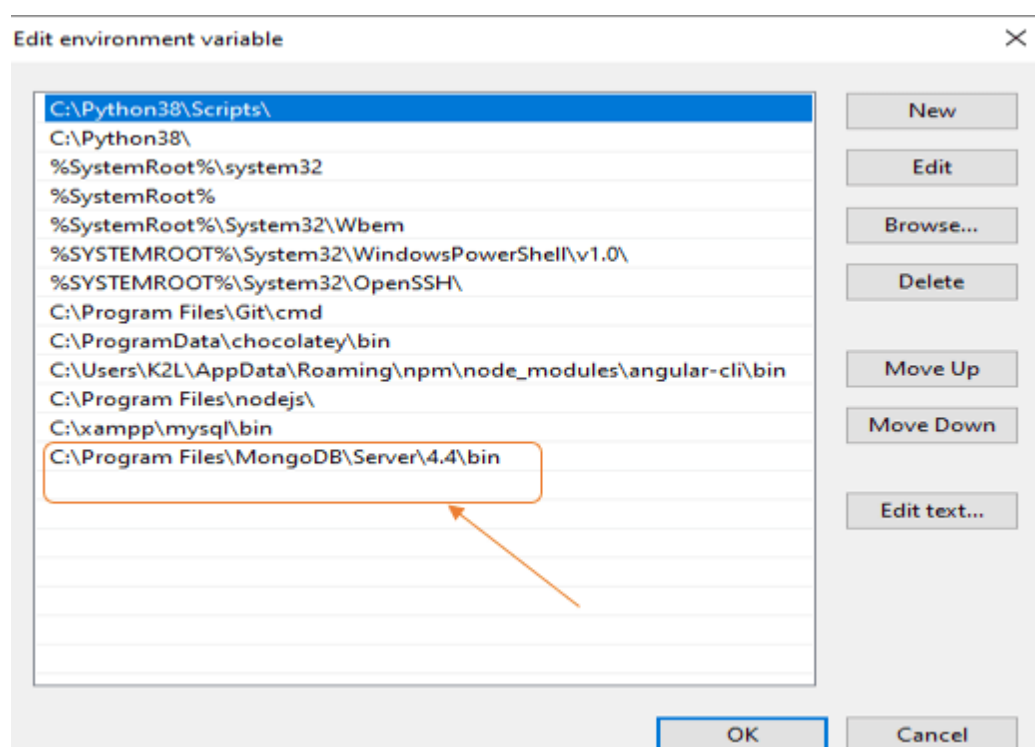
**Step 8:** Now click the ***Finish button*** to complete the MongoDB installation process:

**Step 9:** Now we go to the location where MongoDB installed in step 5 in your system and copy the bin path:





**Step 10:** Now, to create an environment variable open system properties << Environment Variable << System variable << path << Edit Environment variable and paste the copied link to your environment system and click Ok:



**Step 11:** After setting the environment variable, we will run the MongoDB server, i.e. mongod. So, open the command prompt and run the following command:

```
mongod
```

When you run this command you will get an error i.e. *C:/data/db/ not found*.

**Step 12:** Now, Open C drive and create a folder named “data” inside this folder create another folder named “db”. After creating these folders. Again open the command prompt and run the following command:

```
mongod
```

Now, this time the MongoDB server(i.e., mongod) will run successfully.

```
C:\Users\Nikhil Chhipa>mongod
{"t":{"$date":"2021-01-31T00:56:54.081+05:30"},"s":"I", "c":"CONTROL", "id":23285, "ctx"
ify --sslDisabledProtocols 'none'"}
{"t":{"$date":"2021-01-31T00:56:54.087+05:30"},"s":"W", "c":"ASIO", "id":22601, "ctx"
}
{"t":{"$date":"2021-01-31T00:56:54.088+05:30"},"s":"I", "c":"NETWORK", "id":4648602, "ctx"
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I", "c":"STORAGE", "id":4615611, "ctx"
bPath":"C:/data/db/","architecture":"64-bit","host":"DESKTOP-L9MUQ7N"}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I", "c":"CONTROL", "id":23398, "ctx"
rgetMinOS":"Windows 7/Windows Server 2008 R2"}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I", "c":"CONTROL", "id":23403, "ctx"
gitVersion":"913d6b62acfb344dde1b116f4161360acd8fd13","modules":[],"allocator":"tcmalloc",
}}}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I", "c":"CONTROL", "id":51765, "ctx"
ndows 10","version":"10.0 (build 14393)"}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I", "c":"CONTROL", "id":21951, "ctx"
{"t":{"$date":"2021-01-31T00:56:54.157+05:30"},"s":"I", "c":"STORAGE", "id":22270, "ctx"
:{"dbpath":"C:/data/db/","storageEngine":"wiredTiger"}}
{"t":{"$date":"2021-01-31T00:56:54.158+05:30"},"s":"I", "c":"STORAGE", "id":22315, "ctx"
ize=1491M,session_max=33000,eviction=(threads_min=4,threads_max=4),config_base=false,statist
le_manager=(close_idle_time=100000,close_scan_interval=10,close_handle_minimum=250),statisti
ess],"}
{"t":{"$date":"2021-01-31T00:56:54.395+05:30"},"s":"I", "c":"STORAGE", "id":22430, "ctx"
95788][3708:140713908197088], txn-recover: [WT_VERB_RECOVERY_PROGRESS] Recovering log 20 thr
{"t":{"$date":"2021-01-31T00:56:54.631+05:30"},"s":"I", "c":"STORAGE", "id":22430, "ctx"
```

## Run mongo Shell

**Step 13:** Now we are going to connect our server (mongod) with the mongo shell. So, keep that mongod window and open a new command prompt window and write **mongo**. Now, our mongo shell will successfully connect to the mongod.

**Important Point:** Please do not close the mongod window if you close this window your server will stop working and it will not able to connect with the mongo shell.

```
C:\Users\Nikhil Chhipa>mongo
MongoDB shell version v4.4.3
connecting to: mongod://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongod
Implicit session: session { "id" : UUID("96cca5da-dc9f-4a40-aabb-732ee37600c0") }
MongoDB server version: 4.4.3

The server generated these startup warnings when booting:
 2021-01-28T20:56:52.570+05:30: Access control is not enabled for the database. Read and write access
configuration is unrestricted

 Enable MongoDB's free cloud-based monitoring service, which will then receive and display
metrics about your deployment (disk utilization, CPU, operation statistics, etc).

 The monitoring data will be available on a MongoDB website with a unique URL accessible to you
and anyone you share the URL with. MongoDB may use this information to make product
improvements and to suggest MongoDB products and deployment options to you.

 To enable free monitoring, run the following command: db.enableFreeMonitoring()
 To permanently disable this reminder, run the following command: db.disableFreeMonitoring()

>
```



Now, you are ready to write queries in the mongo Shell.

### Run MongoDB

Now you can make a new database, collections, and documents in your shell. Below is an example of how to make a new database:

The `use Database_name` command makes a new database in the system if it does not exist, if the database exists it uses that database:

```
use gfg
```

Now your database is ready of name gfg.

The `db.Collection_name` command makes a new collection in the gfg database and the `insertOne()` method inserts the document in the student collection: `db.student.insertOne({Akshay:500})`

```
> use gfg
switched to db gfg
> db.student.insertOne({Akshay:500})
{
 "acknowledged" : true,
 "insertedId" : ObjectId("60083bf8b7388ed4d54157c9")
}
> db.student.find().pretty()
{ "_id" : ObjectId("60083bf8b7388ed4d54157c9"), "Akshay" : 500 }
>
```

## 2. MongoDB – Visual Studio Extension

- Search and install MongoDB Visual Studio Code – Extension

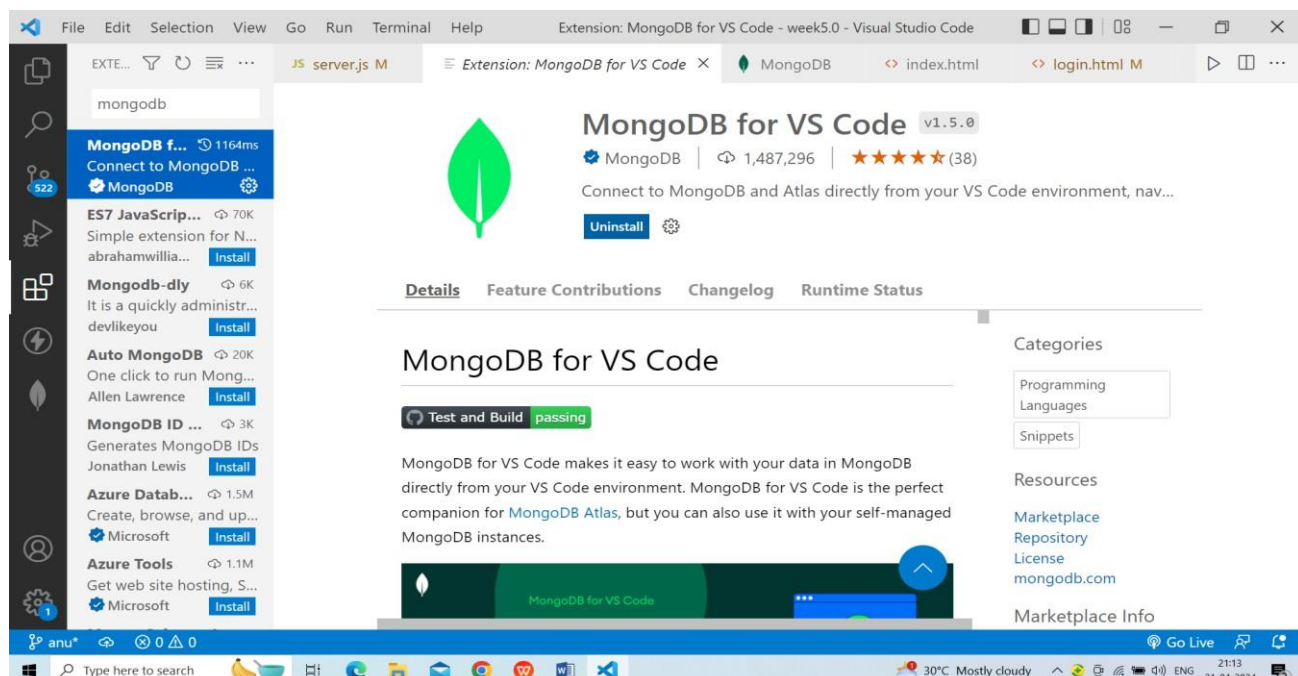


Fig.1. MongoDB – Visual Studio Extension

### MongoDB – Atlas

- Cloud-Hosted and Fully Managed MongoDB
- Pay as you go model

- - Very cost-effective
  - Fully secured and reliable

## Data Modeling

Definition:-

Data modelling refers to the organization of data within a database and the links between related entities. Data in MongoDB has a **flexible schema model**, which means:

- Documents within a single collection are not required to have the same set of fields.
- A field's data type can differ between documents within a collection.

The primary problem in data modeling is balancing application needs, **database engine performance** features, and **data retrieval patterns**. Always consider the application uses of the data (i.e. **queries**, **updates**, and **data processing**) as well as the fundamental design of the data itself when creating data models.

## Advantages Of Data Modelling

Data modelling is essential for a successful application, even though at first it might just seem like one more step. In addition to increasing **overall efficiency** and **improving development cycles**, data modelling helps you better understand the data at hand and identify future business requirements, which can save time and money. In particular, applying suitable data models:

- Improves application performance through better database strategy, design, and implementation.
- Allows faster application development by making object mapping easier.
- Helps with better data learning, standards, and validation.
- Allows organizations to assess long-term solutions and model data while solving not just current projects but also future application requirements, including maintenance.

## Different Types of Data Models

The three types of data models that are typically classified as follows:

### 1. Conceptual data model

Conceptual Data Models are **rough sketches** that provide the big picture, detailing where data/information from various business processes will be stored in the database system and the relationships they will be involved with. A conceptual data model typically includes the **entity class**, **attributes**, **constraints**, and the relationship between security and data integrity requirements.

This model describes the types of data that should be in the system and how they relate to one another. This model, which is typically developed with the support of the business stakeholders, it contains the business logic of the application, often involves **domain-driven design** (DDD) principles, and serves as the foundation for one or more of the following models. The primary purpose of the conceptual model is to identify the **information that will be essential to an organization**.

### 2. Logical data model

Logical data models provide more detailed, **subjective information about data set relationships**. At this stage, we can clearly connect what data types and relations are used. Logical data models are generally missed in rapid business contexts, having their utility in **data-driven** initiatives requiring important procedure execution.

The logical data model specifies **how data will be organized**. The relationship between entities is

established at a high level .In this model, and a list of entity properties is also provided. This data model can be viewed as a “**blueprint**” for the data that will be used.

### **3. Physical data model**

The schema/layout for data storage routines within a database is defined by the **physical data model**. A physical data model is a ready-to-implement plan that can be stored in a **relational database**.

The physical data model is a representation of **how data will be stored in a particular database** management system (DBMS). In this approach, main and secondary keys in a relational database are defined, or the decision to include or connect data in a document database such as MongoDB based on entity relationships is made. This is also where you will define the **data types for each of your fields**, which will create the database structure.

## **Data Model Design (or) Types**

For modelling data in MongoDB, two strategies are available. These strategies are different and it is recommended to analyze our scenario for a better flow. The two methods are as follows:

1. Embedded Data Model
2. Normalized Data Model

### **1. Embedded Data Model**

This method, also known as the **de-normalized** data model, allows you to have (embed) all of the **related data in a single document**.

For example, if we obtain student information in three different documents, Personal\_details, Contact, and Address, we can embed all three in a single one, as shown below.

```
{
 _id: ,
 Std_ID: "987STD001"
 Personal_details:{
 First_Name:
 "Rashmika",
 Last_Name: "Sharma",
 Date_Of_Birth: "1999-08-26"
 },
 Contact: {
 e-mail:
 "rashmika_sharma.123@gmail.com",
 phone: "9987645673"
 },
 Address: {
 city: "Karnataka",
 Area: "BTM2ndStage",
 State: "Bengaluru"
 }
}
```

## 2. Normalized Data Model (or) Reference Data Model:

In a normalized data model, object references are used to express the **relationships between documents and data objects**. Because this approach **reduces data duplication**, it is relatively simple to document **many-to-many relationships** without having to repeat content. Normalized data models are the most effective technique to model large **hierarchical data** with cross-collection relationships.

*Student:*

```
{
 _id: <StudentId101>,
 Std_ID: "10025AE336"
}
```

*Personal\_Details:*

```
{
 _id: <StudentId102>,
 stdDocID: " StudentId101",
 First_Name: "Rashmika",
 Last_Name: "Sharma",
 Date_Of_Birth: "1999-08-26"
}
```

*Contact:*

```
{
 _id: <StudentId103>,
 stdDocID: "
 StudentId101",
 e-mail:
 "rashmika_sharma.123@gmail.com",
 phone: "9987645673"
}
```

*Address:*

```
{
 _id: <StudentId104>,
 stdDocID: "
 StudentId101", city:
 "Karnataka",
 Area:
 "BTM2ndStage",
 State: "Bengaluru"
}
```

### Considerations while designing Schema in MongoDB

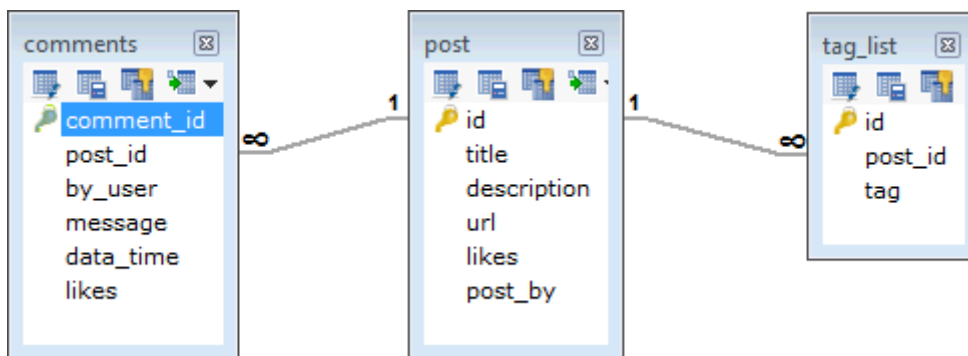
- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.
- Do joins while write, not on read.
- Optimize your schema for most frequent use cases.
- Do complex aggregation in the schema.

## Example

Suppose a client needs a database design for his blog/website and see the differences between RDBMS and MongoDB schema design. Website has the following requirements.

- Every post has the unique title, description and url.
- Every post can have one or more tags.
- Every post has the name of its publisher and total number of likes.
- Every post has comments given by users along with their name, message, data-time and likes.
- On each post, there can be zero or more comments.

In RDBMS schema, design for above requirements will have minimum three tables.



While in MongoDB schema, design will have one collection post and the following structure –

```

{
 _id: POST_ID
 title: TITLE_OF_POST,
 description: POST_DESCRIPTION,
 by: POST_BY,
 url: URL_OF_POST,
 tags: [TAG1, TAG2, TAG3],
 likes: TOTAL_LIKES,
 comments: [
 {
 user:'COMMENT_BY',
 message: TEXT,
 dateCreated: DATE_TIME,
 like: LIKES
 },
 {
 user:'COMMENT_BY',
 message: TEXT,
 dateCreated: DATE_TIME,
 like: LIKES
 }
]
}

```

So while showing the data, in RDBMS you need to join three tables and in MongoDB, data will be shown from one collection only.

### Connect MongoDB:-

Mongodb compass

- Connect with compass app
- Understand basics of compass app
- Get your hands-on examples with compass

Hostname:-Localhost

Port:-27017

Visual Studio:-

- Connect with Visual Studio Code Extension
- Understand basics of visual Studio Code Extension
- Get your hands-on examples with Visual Studio Code Extension

**Connect:-** mongodb://localhost:27017

Shell:**mongos**

**h** (Or)

**Mongod** creating server and **mongo** for shell.

CURD Operation:-

**Creating and drop database:-**

- use `use anu;`//creating
- `show dbs;`//display all db
- `db;`//current db
- `db.dropDatabase();`//deleting

**db Creating and drop collections:-**

**Syntax:-**

- `db.createCollection(name,options)`//creating
- `db.collection.drop()`
- `db.collection.insertOne({key:"value"})`

**Ex:-**

- `db.createCollection("products");`
- `db.products.drop()`

**Inserting Documents into**

**Collections:- Syntax:-**

- `db.collection_name.insert({"name":"aaa"})`//one
- `db.collection_name.insertMany([{"name":"aaa"}, {"name":"bbb"}])`//many
- **Example:-** `db.aaa.insert({"name":"aaa"})`//one
- `db.bbb.insertMany([{"name":"aaa"}, {"name":"bbb"}])`//many

**Update:-**

- `db.bbb.update({"name":"bbb"},{$set: {"name":"ccc","isActive":true}});`

**Read:-**

- `db.bbb.find();`
- `db.bbb.findOne();`
- `db.bbb.find({"name":"ccc"});`
- `db.bbb.findOneAndReplace({"name":"ccc"}, {"name":"eee"});`
- `db.bbb.findOneAndDelete({"name":"eee"});`



- •

### Delete:-

```
db.orders.deleteOne({"name":"aaa"})
```

```
/*
```

```
Db.student.insertOne({name:"anusha"})
```

```
Db.student.find().pretty()
```

```
*/
```

### Query and Projection

#### MongoDB Query

##### MongoDB Query Operators

Similar to **SQL** MongoDB have also some operators to operate on data in the collection. MongoDB query operators check the conditions for the given data and **logically** compare the data with the help of two or more fields in the [document](#).

**Query operators** help to filter data based on specific conditions. **E.g.**, \$eq,\$and,\$exists, etc.

MongoDB provides the function names as *db.collection\_name.find()* to operate query operation on database.

#### Syntax:

```
db.collection_name.find()
```

#### Example:

```
db.article.find()
```

#### Types of Query Operators in MongoDB

The Query operators in MongoDB can be further classified into 8 more types. The 8 types of Query Operators in MongoDB are:

1. Comparison Operators
2. Logical Operators
3. Array Operators
4. Evaluation Operators
5. Element Operators
6. Bitwise Operators
7. Geospatial Operators
8. Comment Operators

#### 1. Comparison Operators

The comparison operators in MongoDB are used to perform value-based comparisons in queries. The

comparison operators in the MongoDB are shown as below:

Comparison Operator	Description	Syntax
<b>\$eq</b>	Matches values that are equal to a specified value.	{ field: { \$eq: value } }
<b>\$ne</b>	Matches all values that are not equal to a specified value.	{ field: { \$ne: value } }
<b>\$lt</b>	Matches values that are less than a specified value.	{ field: { \$lt: value } }
<b>\$gt</b>	Matches values that are greater than a specified value.	{ field: { \$gt: value } }
<b>\$lte</b>	Matches values that are less than or equal to a specified value.	{ field: { \$lte: value } }
<b>\$gte</b>	Matches values that are greater than or equal to a specified value.	{ field: { \$gte: value } }
<b>\$in</b>	Matches any of the values specified in an array.	{ field: { \$in: [<value1>, <value2>, ... ] } }

### Documents:

```
db.books.insertMany([{"p_name":"book","price":50}, {"p_name":"pen","price":100}, {"p_name":"pencilbox","price":500}, {"p_name":"ball","price":200}]);
```

### MongoDB Comparison Operators

#### 1. \$eq

The \$eq specifies the equality condition. It matches documents where the value of a field equals the specified value.

#### Syntax:

```
{ <field> : { $eq: <value> } }
```

#### Example:

```
db.books.find ({ price: { $eq: 200 } })
```

The above example queries the books collection to select all documents where the value of the price filed equals 300.



```
query> db.books.find ({ price: { $eq: 200 } })
[
 {
 _id: ObjectId('662c059ccbf0d16226117b86'),
 p_name: 'ball',
 price: 200
 }
]
```

## 2. \$gt

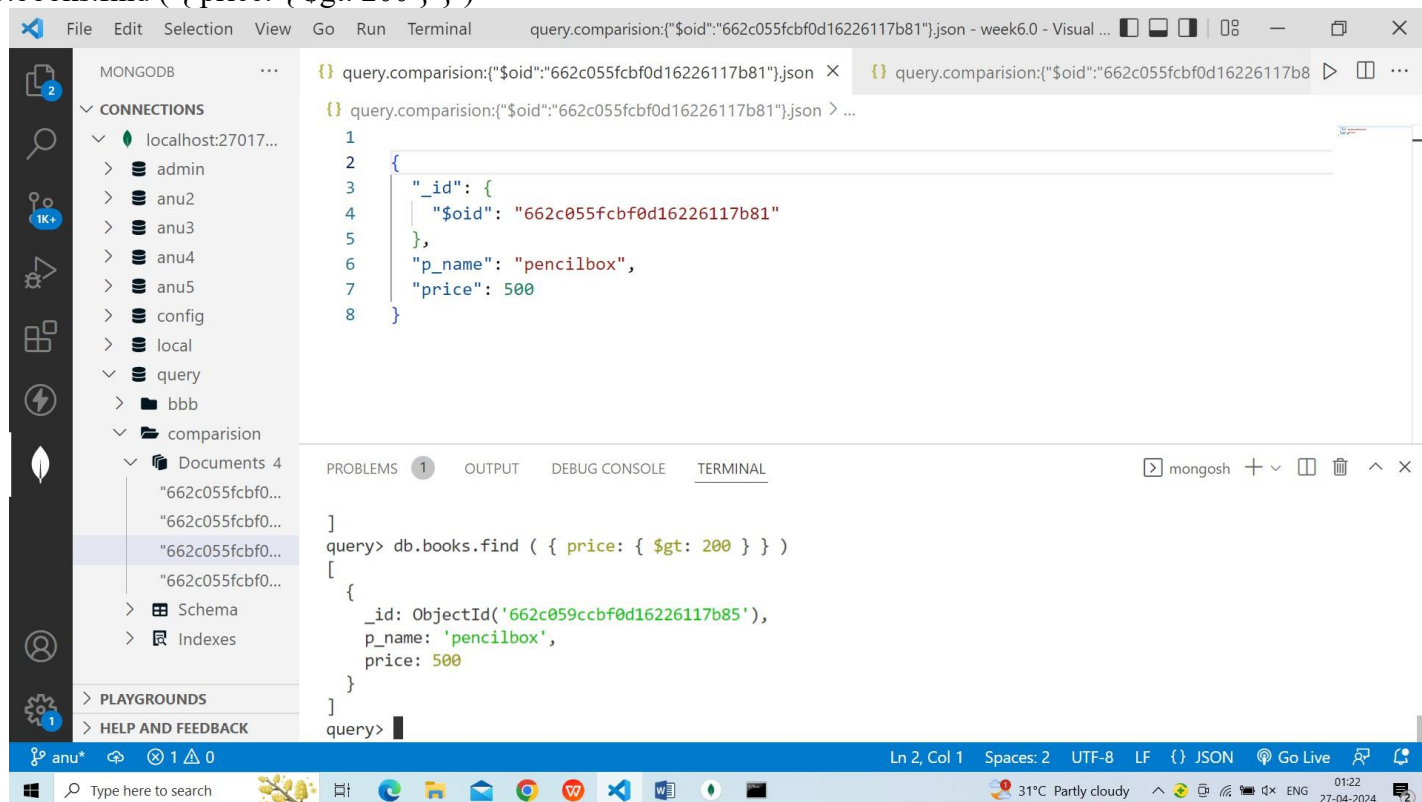
The \$gt chooses a document where the value of the field is greater than the specified value.

**Syntax:**

`{ field: { $gt: value } }`

**Example:**

`db.books.find ( { price: { $gt: 200 } } )`



```
query> db.books.find ({ price: { $gt: 200 } })
[
 {
 _id: ObjectId('662c059ccbf0d16226117b85'),
 p_name: 'pencilbox',
 price: 500
 }
]
```

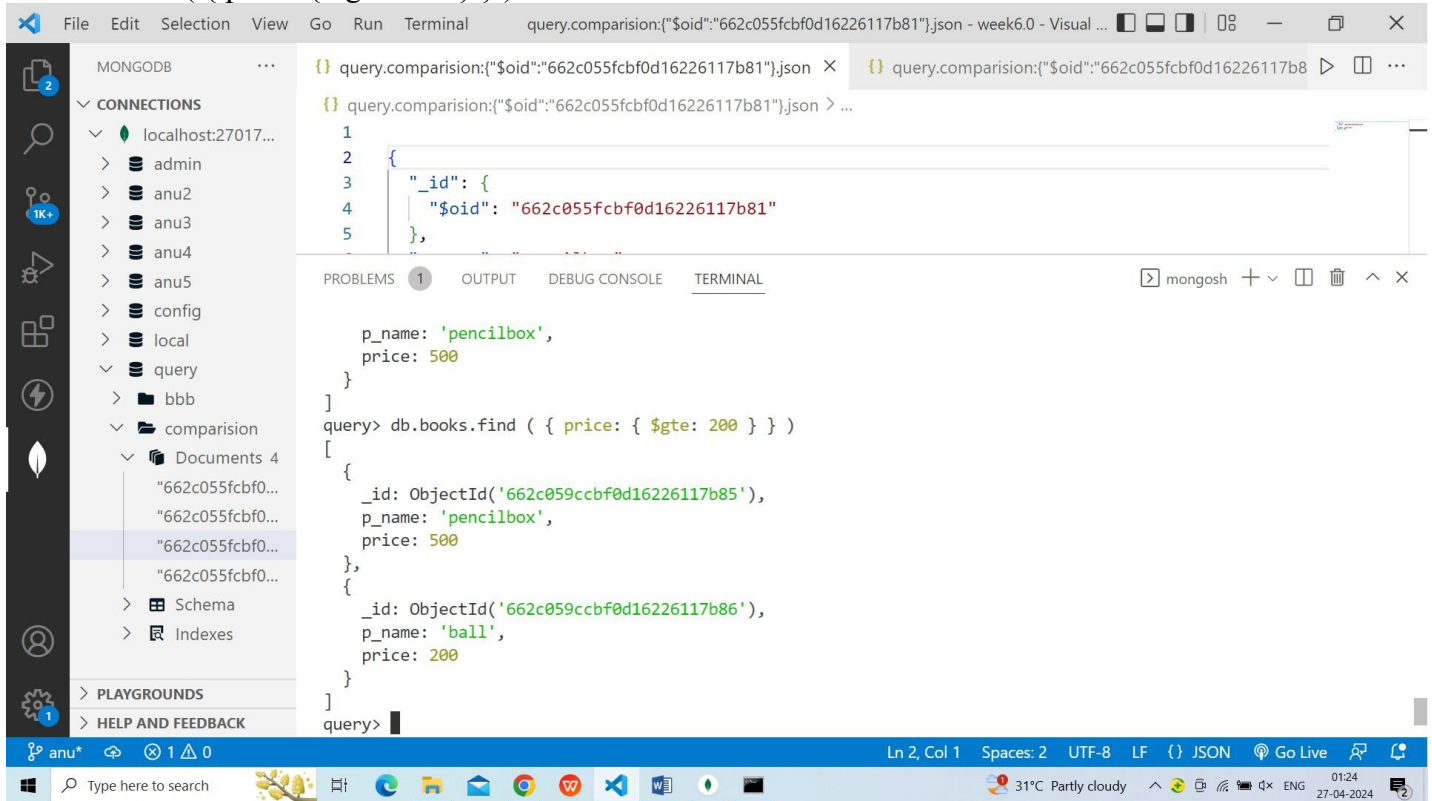
## 3. \$gte

The \$gte choose the documents where the field value is greater than or equal to a specified value.

**Syntax:**

### Example:

```
db.books.find ({ price: { $gte: 250 } })
```



4. \$in

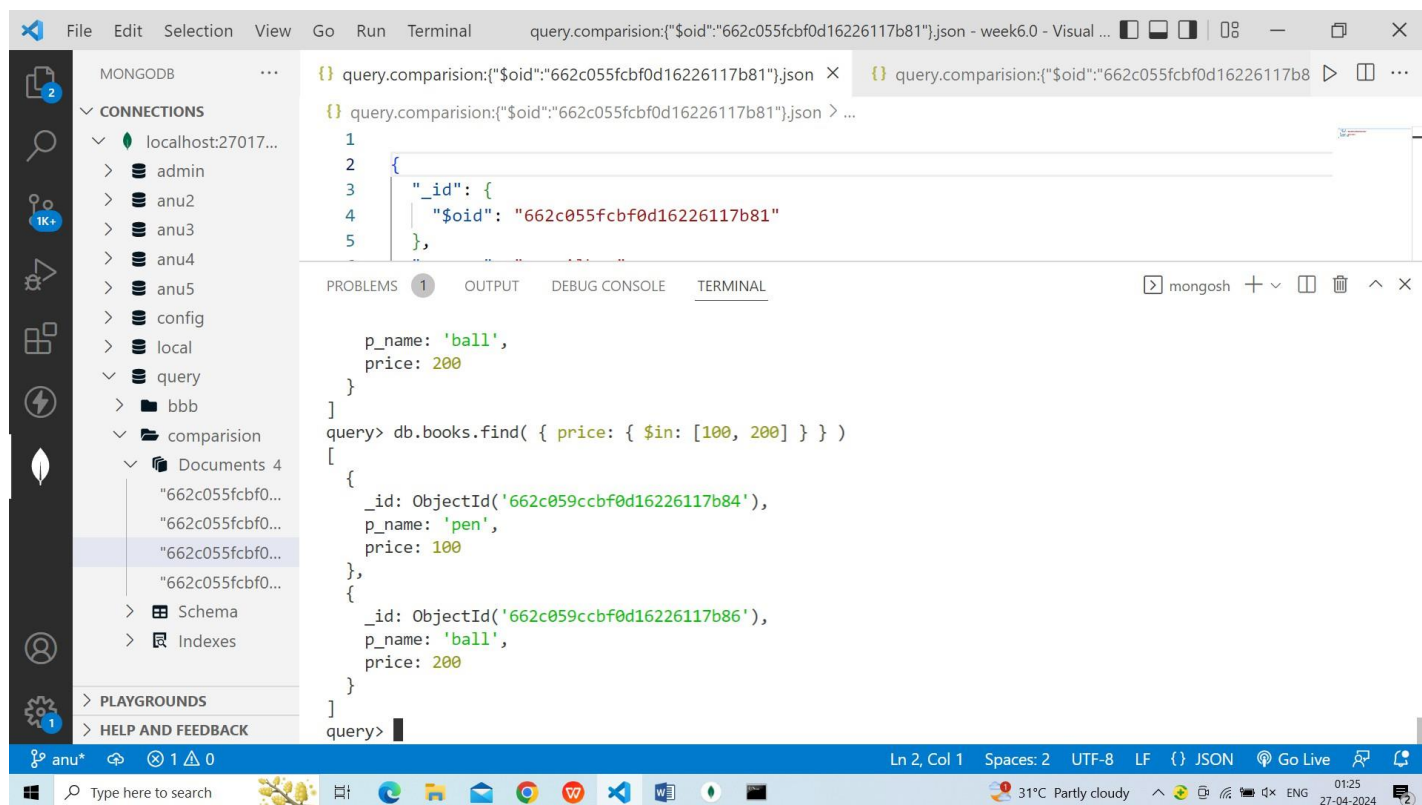
The \$in operator choose the documents where the value of a field equals any value in the specified array.

### Syntax:

```
. { filed: { $in: [<value1>, <value2>,] } }
```

### Example:

```
. db.books.find({ price: { $in: [100, 200] } })
```



## 5. \$lt

The \$lt operator chooses the documents where the value of the field is less than the specified value.

### Syntax:

```
{ field: { $lt: value } }
```

### Example:

```
db.books.find ({ price: { $lt: 20 } })
```

Visual Studio Code interface showing a MongoDB connection and a query execution.

**Left Panel (MongoDB Explorer):**

- CONNECTIONS
  - localhost:27017...
    - admin
    - anu2
    - anu3
    - anu4
    - anu5
    - config
    - local
    - query
    - bbb
    - comparison
      - Documents 4
        - "662c055fcbf0d16226117b81"
        - "662c055fcbf0d16226117b82"
        - "662c055fcbf0d16226117b83"
        - "662c055fcbf0d16226117b84"
      - Schema
      - Indexes

- PLAYGROUNDS
- HELP AND FEEDBACK

**Editor (query.comparison.json):**

```
{
 "$oid": "662c055fcbf0d16226117b81"
}
```

**Terminal (mongosh):**

```
query> db.books.find ({ price: { $lt: 200 } })
[
 {
 _id: ObjectId('662c059ccbf0d16226117b83'),
 p_name: 'book',
 price: 50
 },
 {
 _id: ObjectId('662c059ccbf0d16226117b84'),
 p_name: 'pen',
 price: 100
 }
]
```

**Status Bar:** anu\* | 1 | 0 | Ln 2, Col 1 | Spaces: 2 | UTF-8 | LF | {} | JSON | Go Live | 31°C Partly cloudy | 01:26 | 27-04-2024





```
query.comparison:{"$oid":"662c055fcbf0d16226117b81"}.json - week6.0 - Visual ...
query.comparison:{"$oid":"662c055fcbf0d16226117b81"}.json > ...

p_name: 'ball',
price: 200
}
]
query> db.books.find ({ price: { $ne: 500 } })
[
 {
 _id: ObjectId('662c059ccbf0d16226117b83'),
 p_name: 'book',
 price: 50
 },
 {
 _id: ObjectId('662c059ccbf0d16226117b84'),
 p_name: 'pen',
 price: 100
 },
 {
 _id: ObjectId('662c059ccbf0d16226117b86'),
 p_name: 'ball',
 price: 200
 }
]
query>
```

## 8. \$nin

The \$nin operator chooses the documents where the field value is not in the specified array or does not exist.

### Syntax:

```
{ field : { $nin: [<value1>, <value2>, ...] } }
```

### Example:

```
db.books.find ({ price: { $nin: [50, 150, 200] } })
```

Visual Studio Code interface showing a MongoDB connection and a terminal window.

**Left Panel (MongoDB Explorer):**

- CONNECTIONS
  - localhost:27017...
    - admin
    - anu2
    - anu3
    - anu4
    - anu5
    - config
    - local
    - query
      - bbb
      - comparison
        - Documents 4
          - "662c055fcfb0d16226117b81"
          - "662c055fcfb0d16226117b82"
          - "662c055fcfb0d16226117b83"
          - "662c055fcfb0d16226117b84"
        - Schema
        - Indexes

**Terminal Window (mongosh):**

```
query.comparison:{"$oid":"662c055fcfb0d16226117b81"}.json
query.comparison:{"$oid":"662c055fcfb0d16226117b82"}.json
1
p_name: 'pen',
price: 100
},
{
 _id: ObjectId('662c059ccbf0d16226117b86'),
 p_name: 'ball',
 price: 200
}
]
query> db.books.find ({ price: { $nin: [50, 150, 200] } })
[
 {
 _id: ObjectId('662c059ccbf0d16226117b84'),
 p_name: 'pen',
 price: 100
 },
 {
 _id: ObjectId('662c059ccbf0d16226117b85'),
 p_name: 'pencilbox',
 price: 500
 }
]
query>
```

**Bottom Panel (Taskbar):**

- Windows taskbar with search bar and icons for various applications.
- System tray showing weather (31°C Partly cloudy), time (01:28), and date (27-04-2024).

## MongoDB Logical Operator

### Logical Operators

The logical operators in MongoDB are used to filter data based on expressions that evaluate to true or false. The Logical operators in MongoDB are shown in the table below:

Logical Operator	Description	Syntax
<b>\$and</b>	Returns all the documents that satisfy all the conditions.	<pre>{ \$and: [ { &lt;expression1&gt; }, { &lt;expression2&gt; }, ... , { &lt;expressionN&gt; } ] }</pre>
<b>\$not</b>	Inverts the effect of the query expression and returns documents that do not match the query expression.	<pre>{ field: { \$not: { &lt;operator-expression&gt; } } }</pre>
<b>\$or</b>	Returns the documents from the query that match either one of the conditions in the query.	<pre>{ \$or: [ { &lt;expression1&gt; }, { &lt;expression2&gt; }, ... , { &lt;expressionN&gt; } ] }</pre>
<b>\$nor</b>	Returns the documents that fail to match both conditions.	<pre>{ \$nor: [ { &lt;expression1&gt; }, { &lt;expression2&gt; }, ... , { &lt;expressionN&gt; } ] }</pre>

### \$and

The \$and operator works as a logical AND operation on an array. The array should be of one or more expressions and chooses the documents that satisfy all the expressions in the array.

#### Syntax:

```
{ $and: [{ <exp1> }, { <exp2> },...] }
```

#### Example:

```
db.books.find ({ $and: [{ price: { $ne: 500 } }, { price: { $exists: true } }] })
```

The screenshot shows a MongoDB IDE interface. On the left, a sidebar displays the 'CONNECTIONS' list with a tree view of databases and collections. The 'query.comparison' database is selected, showing a 'Documents' collection with 4 documents. The main editor area shows a MongoDB query in the terminal: `query> db.books.find ( { $and: [ { price: { $ne: 500 } }, { price: { $exists: true } } ] } )`. The results are displayed in the terminal, showing three documents: a 'pencilbox' with price 500, a 'book' with price 50, a 'pen' with price 100, and a 'ball' with price 200. The status bar at the bottom indicates the current line and column, spaces, encoding, and line endings.

## \$not

The \$not operator works as a logical NOT on the specified expression and chooses the documents that are not related to the expression.

### Syntax:

```
{ field: { $not: { <operator-expression> } } }
```

### Example:

```
db.books.find ({ price: { $not: { $gt: 200 } } })
```

```

query.comparison:{"$oid":"662c055fcbf0d16226117b81"},json
query.comparison:{"$oid":"662c055fcbf0d16226117b81"},json > ...

1
query.comparison:{"$oid":"662c055fcbf0d16226117b81"},json > ...

p_name: 'ball',
price: 200
}
]
query> db.books.find ({ price: { $not: { $gt: 200 } } })
[
 {
 _id: ObjectId('662c059ccbf0d16226117b83'),
 p_name: 'book',
 price: 50
 },
 {
 _id: ObjectId('662c059ccbf0d16226117b84'),
 p_name: 'pen',
 price: 100
 },
 {
 _id: ObjectId('662c059ccbf0d16226117b86'),
 p_name: 'ball',
 price: 200
 }
]
query>

```

## \$nor

The \$nor operator works as logical NOR on an array of one or more query expression and chooses the documents that fail all the query expression in the array.

### Syntax:

```
{ $nor: [{ <expression1> } , { <expresion2> } ,] }
```

### Example:

```
db.books.find ({ $nor: [{ price: 200 } , { p_name:"pen" }] })
```



**\$or**

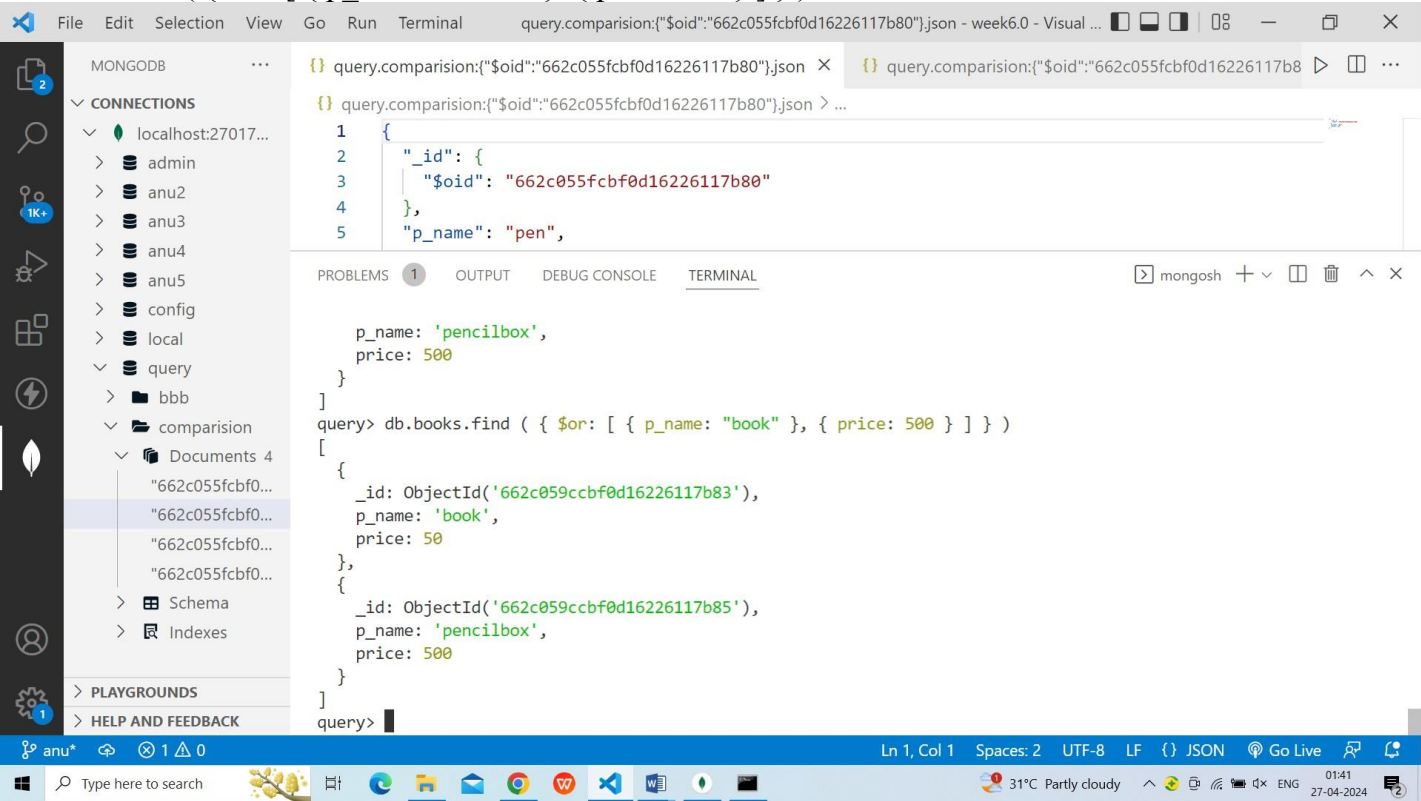
It works as a logical OR operation on an array of two or more expressions and chooses documents that meet the expectation at least one of the expressions.

**Syntax:**

`{ $or: [ { <exp_1> }, { <exp_2> }, ... , { <exp_n> } ] }`

**Example:**

`db.books.find ( { $or: [ { p_name: "book" }, { price: 500 } ] } )`



**Array Operator**

Name	Description
<code>\$all</code>	Matches arrays that contain all elements specified in the query.
<code>\$elemMatch</code>	Selects documents if element in the array field matches all the specified <code>\$elemMatch</code> conditions.
<code>\$size</code>	Selects documents if the array field is a specified size.

**\$all**

It chooses the document where the value of a field is an array that contains all the specified elements.

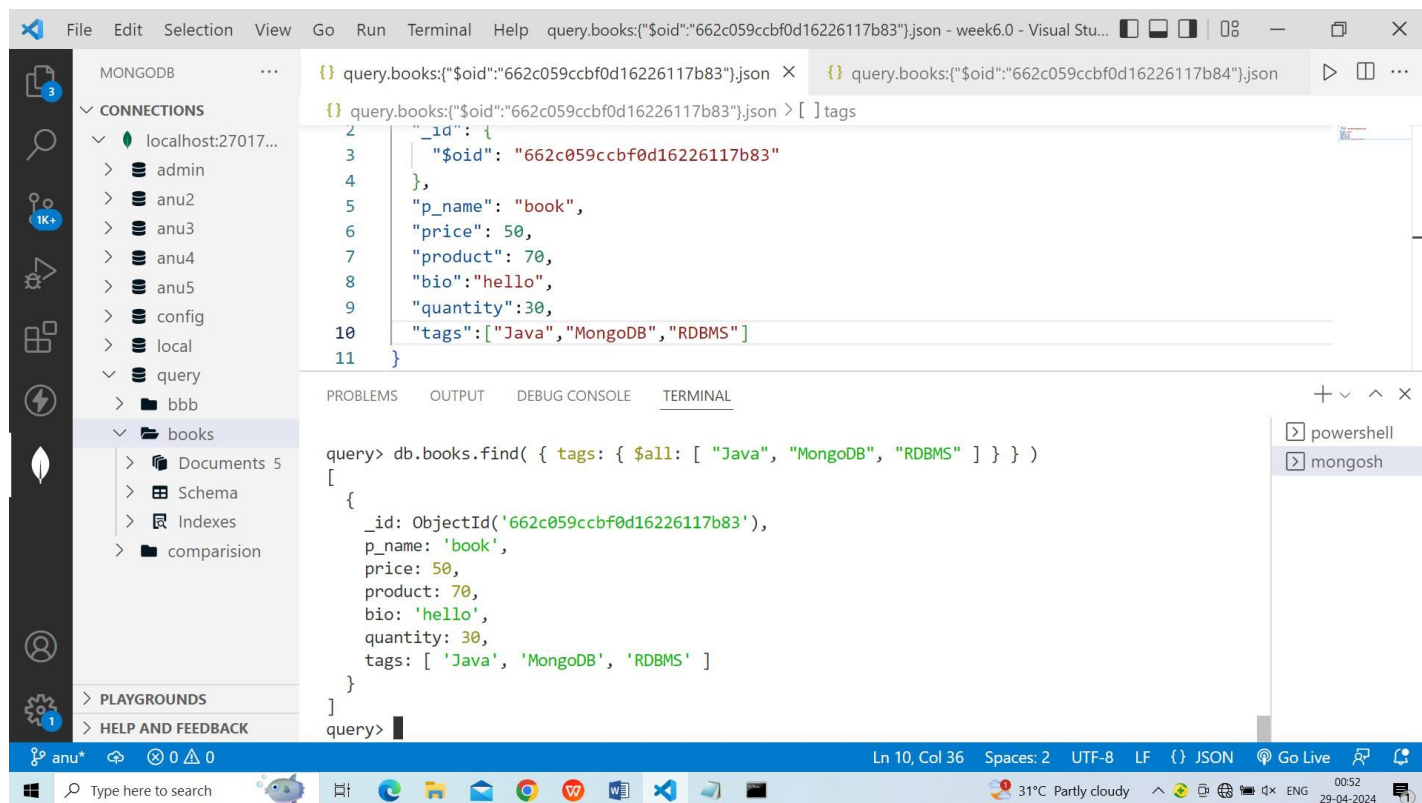


## Syntax:

```
{ <field>: { $all: [<value1> , <value2> ...] } }
```

## Example:

```
db.books.find({ tags: { $all: ["Java", "MongoDB", "RDBMS"] } })
```



The screenshot shows the Visual Studio Code interface with a MongoDB connection to localhost:27017. The left sidebar shows the 'MongoDB' section with a tree view of databases and collections. The 'books' collection is selected. The main editor shows a JSON document for a book with the following fields: `_id`, `$oid`, `p_name`, `price`, `product`, `bio`, `quantity`, and `tags`. The `tags` array contains 'Java', 'MongoDB', and 'RDBMS'. The terminal at the bottom shows the command `query> db.books.find( { tags: { $all: [ 'Java', 'MongoDB', 'RDBMS' ] } } )` and its output, which is a JSON document matching the one in the editor.

## \$elemMatch

The operator relates documents that contain an array field with at least one element that matches with all the given query criteria.

## Syntax:

```
{ <field>: { $elemMatch: { <query1>, <query2>, ... } } }
```

## Example of Using \$elemMatch Operator

Let's first make some updates in our demo collection. Here we will Insert some data into the count\_no database

## Query:

```
db.count_no.insertOne({"name" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green","Blue"]});
```

## Output:



```

Data_base> db.count_no.insertOne({"name":"Harsha","Age":24,"Likes":2,"Colors":["Red","Green","Blue"]});
{
 acknowledged: true,
 insertedId: ObjectId("6589856d543f99be41e7a5ab")
}
Data_base> db.count_no.find({}, {_id:0});
[
 { name: 'Krishna', Age: 22, Likes: 1 },
 { name: 'Shiva', Age: 25, Likes: 2 },
 { name: 'Rama', Age: 23, Likes: 2 },
 { name: 'Hanuman', Age: 23, Likes: 3 },
 {
 name: 'Harsha',
 Age: 24,
 Likes: 2,
 Colors: ['Red', 'Green', 'Blue']
 }
]

```

### Inserting the array of elements .

Now, using the **\$elemMatch** operator in MongoDB let's match the Red colors from a set of Colors.

#### Query:

```
db.count_no.find({"Colors": { $elemMatch: { $eq: "Red" } } }, {_id: 0 });
```

#### Output:

```

Data_base> db.count_no.find({"Colors":{ $elemMatch:{ $eq:"Red" } } }, {_id:0});
[
 {
 name: 'Harsha',
 Age: 24,
 Likes: 2,
 Colors: ['Red', 'Green', 'Blue']
 }
]
Data_base>

```

### Using the elemMatch Operator.

**Explanation:** The **\$elemMatch** operator is used with the field **Colors** which is of type array. In the above query, it returns the documents that have the field Colors and if any of the values in the Colors field has “Red” in it.

#### Example:

```
db.books.find({ price: { $elemMatch: { $gte: 500, $lt: 400 } } })
```

#### \$size

It selects any array with the number of the element specified by the argument.

#### Syntax:

```
db.collection.find({ field: { $size: 2 } });
```

1. `db.count_no.find( {"Colors": { $size: 3 } },{_id: 0 } );`

```
Data_base> db.count_no.find({"Colors":{"$elemMatch":{"$eq:"Red"}}},{_id:0});
[
 {
 name: 'Harsha',
 Age: 24,
 Likes: 2,
 Colors: ['Red', 'Green', 'Blue']
 }
]
Data_base>
```

```
Command Prompt - mongo
{.db.count_no.insertOne({"name" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green","Blue"
 "acknowledged" : true,"e" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green","Blue
 "insertedId" : ObjectId("662ea655b3cd6a4bed84e5e3")es" : 2,"Colors":["Red","Green","Blu
}.db.count_no.insertOne({"name" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green","Bl
> db.count_no.insertOne({"name" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green","B
> db.count_no.insertOne({"name" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green","
> db.count_no.insertOne({"name" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green",
> db.count_no.insertOne({"name" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green"
> db.count_no.insertOne({"name" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green
>
>
> db.count_no.insertOne({"name" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green","Blue"]
});
{
 "acknowledged" : true,
 "insertedId" : ObjectId("662ea65cb3cd6a4bed84e5e4")
}
> db.count_no.find({"Colors": { $elemMatch: { $eq: "Red" } } },{_id: 0 });
{ "name" : "Harsha", "Age" : 24, "Likes" : 2, "Colors" : ["Red", "Green", "Blue"] }
{ "name" : "Harsha", "Age" : 24, "Likes" : 2, "Colors" : ["Red", "Green", "Blue"] }
> db.count_no.find({"Colors": { $size: 3 } },{_id: 0 });
{ "name" : "Harsha", "Age" : 24, "Likes" : 2, "Colors" : ["Red", "Green", "Blue"] }
{ "name" : "Harsha", "Age" : 24, "Likes" : 2, "Colors" : ["Red", "Green", "Blue"] }
>
```

## MongoDB Evaluation Operator

The evaluation operators in the MongoDB are used to return the documents based on the result of the given expression.

Some of the evaluation operators present in the MongoDB are:

Evaluation Operator	Description	Syntax
<b>\$mod operator</b>	The \$mod operator in MongoDB performs a modulo operation on the value of a field and selects documents where the modulo equals a specified value. It only works with numerical fields.	{ field: { \$mod: [ divisor, remainder ] } }
<b>\$expr operator</b>	The \$expr operator in MongoDB allows aggregation expressions to be used as query conditions. It returns documents that satisfy the conditions of the query.	{ \$expr: { <aggregation expression> } }
<b>\$where operator</b>	The \$where operator in MongoDB uses JavaScript expression or function to perform queries. It evaluates the function for every document in the database and returns the documents that match the condition.	{ \$where: <JavaScript expression> }

## Evaluation

### Name Description

---

<a href="#">\$expr</a>	Allows use of aggregation expressions within the query language.
<a href="#">\$jsonSchema</a>	Validate documents against the given JSON Schema.
<a href="#">\$mod</a>	Performs a modulo operation on the value of a field and selects documents with a specified result.
<a href="#">\$regex</a>	Selects documents where values match a specified regular expression.
<a href="#">\$text</a>	Performs text search.
<a href="#">\$where</a>	Matches documents that satisfy a JavaScript expression.

## \$expr

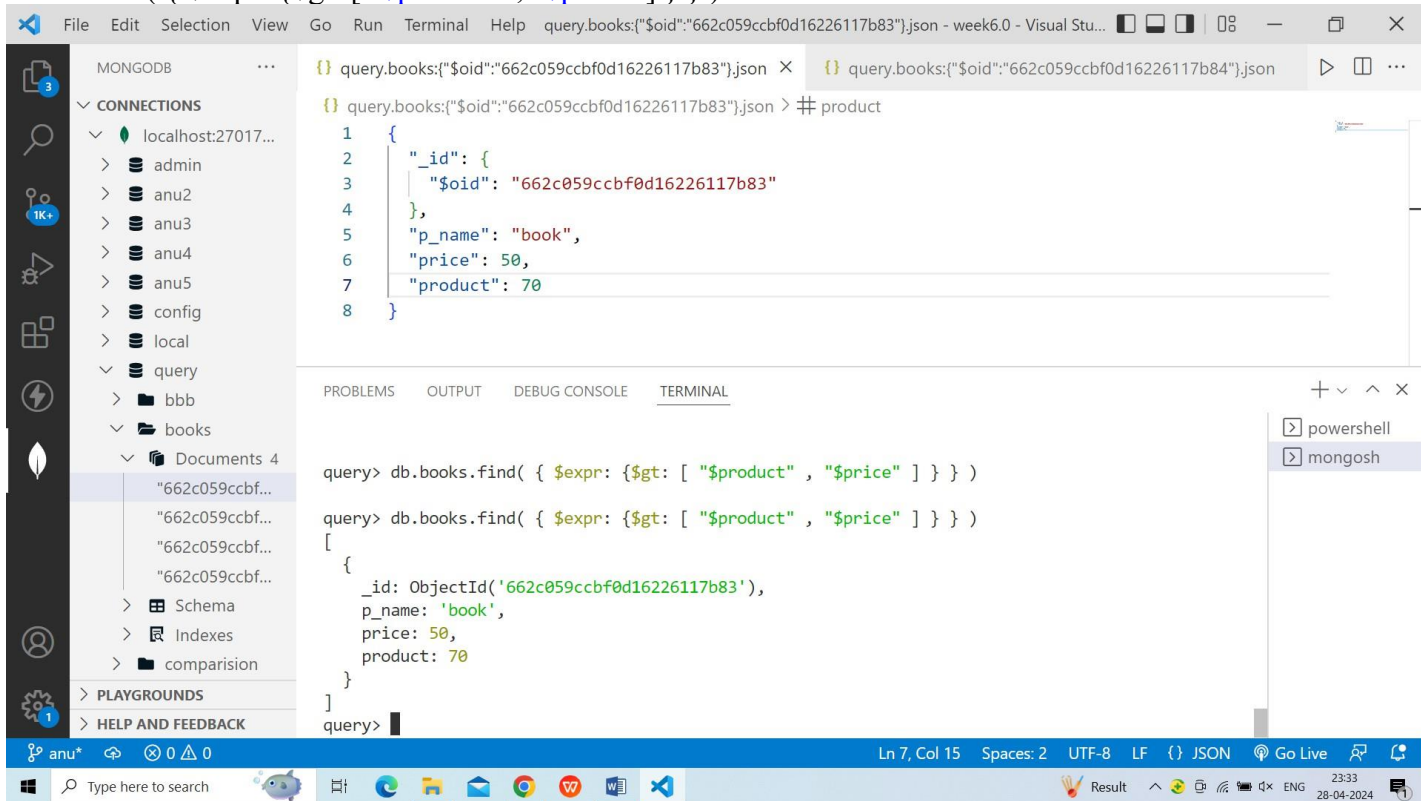
The expr operator allows the use of aggregation expressions within the query language.

### Syntax:

```
{ $expr: { <expression> } }
```

## Example:

```
db.store.find({ $expr: { $gt: ["$product" , "$price"] } })
```



## \$jsonSchema

It matches the documents that satisfy the specified JSON Schema. `db.createCollection("students12",{ validator:{`

```
$jsonSchema: {
 required: ["name", "major", "gpa", "address"],
 properties: {
 name: {
 bsonType: "string",
 description: "must be a string and is required"
 },
 address: {
 bsonType: "object",
 required: ["zipcode"],
```

```

- properties: {
 "street": { bsonType: "string" },
 "zipcode": { bsonType: "string"
 }
 }
 }
 }
}
})

```

/////

Studen

```

t:{
 name:'anusha',
 major:'aaa',
 gpa:'20',
 address:{
 zipcode:'25
 5'
 }
}

```

}Syntax:

```
{ $jsonSchema: <JSON schema object> }
```

```
Select Command Prompt - mongo
... }
... })
uncaught exception: SyntaxError: expected property name, got '{' :
@(shell):3:0
> db.createCollection("students12",{
... validator:{
...
... $jsonSchema: {
... required: ["name", "major", "gpa", "address"],
... properties: {
... name: {
... bsonType: "string",
... description: "must be a string and is required"
... },
... address: {
... bsonType: "object",
... required: ["zipcode"],
... properties: {
... "street": { bsonType: "string" },
... "zipcode": { bsonType: "string" }
... }
... }
... }
... }
... }
... }
{ "ok" : 1 }
> db.books.insertOne({})
... name:anusha,
... major:'aaa'
... gpa:'20'
... address:{
...
... })
uncaught exception: ReferenceError: anusha is not defined :
@(shell):2:1
> db.books.insertOne({
... name:'anusha',
... major:'aaa',
... gpa:'20',
... address:{
... zipcode:'255'
... }
...
... })
{
 "acknowledged" : true,
 "insertedId" : ObjectId("662e9ff0b3cd6a4bed84e5e2")
}
```

**\$mod**

The mod operator selects the document where the value of a field is divided by a divisor has the specified remainder.

**Syntax:**

```
{ field: { $mod: [divisor, remainder] } }
```

**Example:**

1. `db.books.find ( { quantity: { $mod: [ 3, 0] } } )`

The screenshot shows the Visual Studio Code interface with a MongoDB extension. The left sidebar displays the 'MONGODB' section with a tree view of connections and databases. The main editor shows a JSON document for a book with the following fields: `_id`, `$oid`, `p_name`, `price`, `product`, `bio`, and `quantity`. The `quantity` field is highlighted. The bottom terminal window shows the command `query> db.books.find ( { quantity: { $mod: [ 3, 0] } } )` and its output, which is a JSON array containing one document matching the criteria.

```
{
 "_id": {
 "$oid": "662c059ccbf0d16226117b83"
 },
 "p_name": "book",
 "price": 50,
 "product": 70,
 "bio": "hello",
 "quantity": 30
}
```

```
query> db.books.find ({ quantity: { $mod: [3, 0] } })
[
 {
 _id: ObjectId('662c059ccbf0d16226117b83'),
 p_name: 'book',
 price: 50,
 product: 70,
 bio: 'hello',
 quantity: 30
 }
]
```

## \$regex

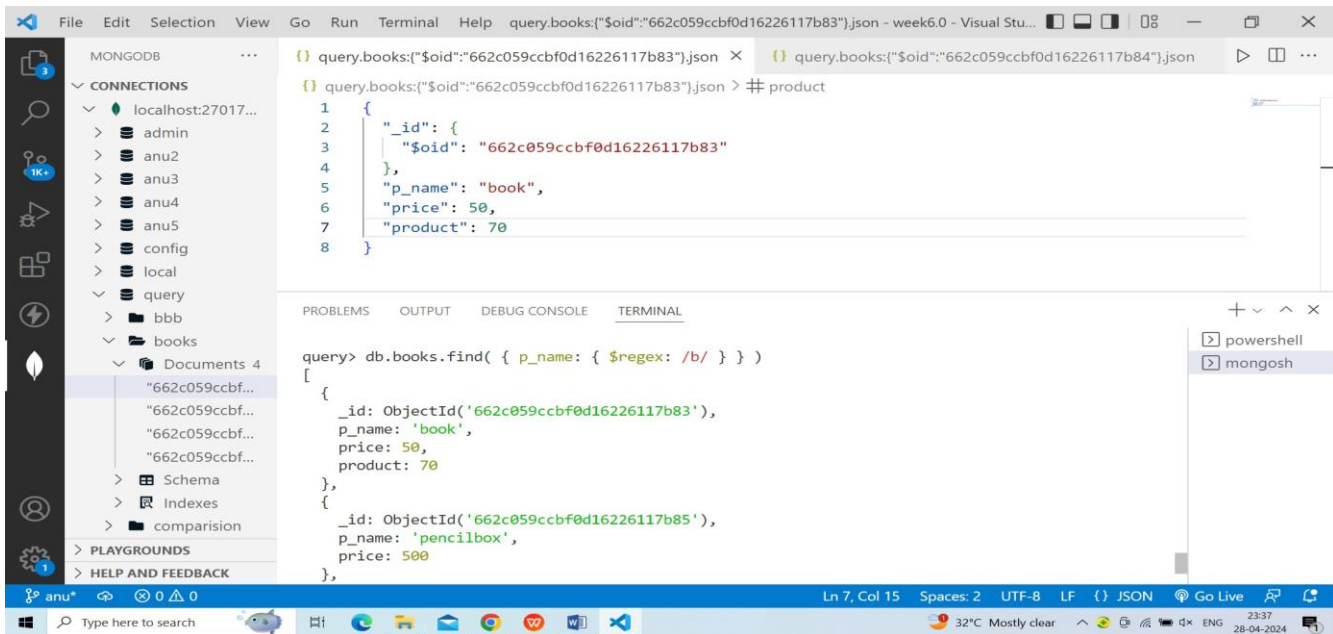
It provides regular expression abilities for pattern matching strings in queries. The MongoDB uses regular expressions that are compatible with Perl.

### Syntax:

1. `{ <field>: /pattern/<options> }`

### Example:

`db.books.find( { p_name: { $regex: /b/ } } )`



## \$text

The \$text operator searches a text on the content of the field, indexed with a text index.

```
db.books.createIndex({bio:"text"})
```

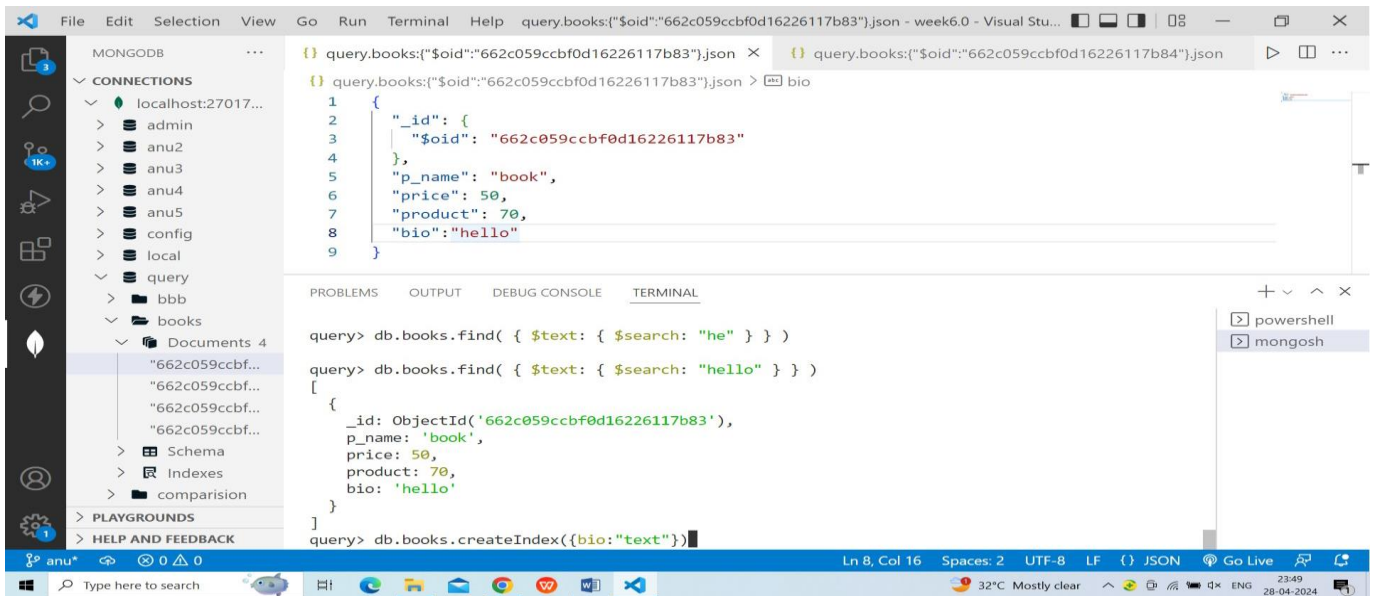
## Syntax:

1. {
2. \$text:
3. {
4. \$search: <string>,
5. \$language: <string>,
6. \$caseSensitive: <boolean>,
7. \$diacriticSensitive: <boolean>
8. }
9. }

## Example:

```
db.books.find({ $text: { $search: "hello" } })
```





```
query.books({'$oid':'662c059ccbf0d16226117b83'})
{
 "_id": {
 "$oid": "662c059ccbf0d16226117b83"
 },
 "p_name": "book",
 "price": 50,
 "product": 70,
 "bio": "hello"
}
```

```
query> db.books.find({ $text: { $search: "he" } })
query> db.books.find({ $text: { $search: "hello" } })
[
 {
 _id: ObjectId('662c059ccbf0d16226117b83'),
 p_name: 'book',
 price: 50,
 product: 70,
 bio: 'hello'
 }
]
```

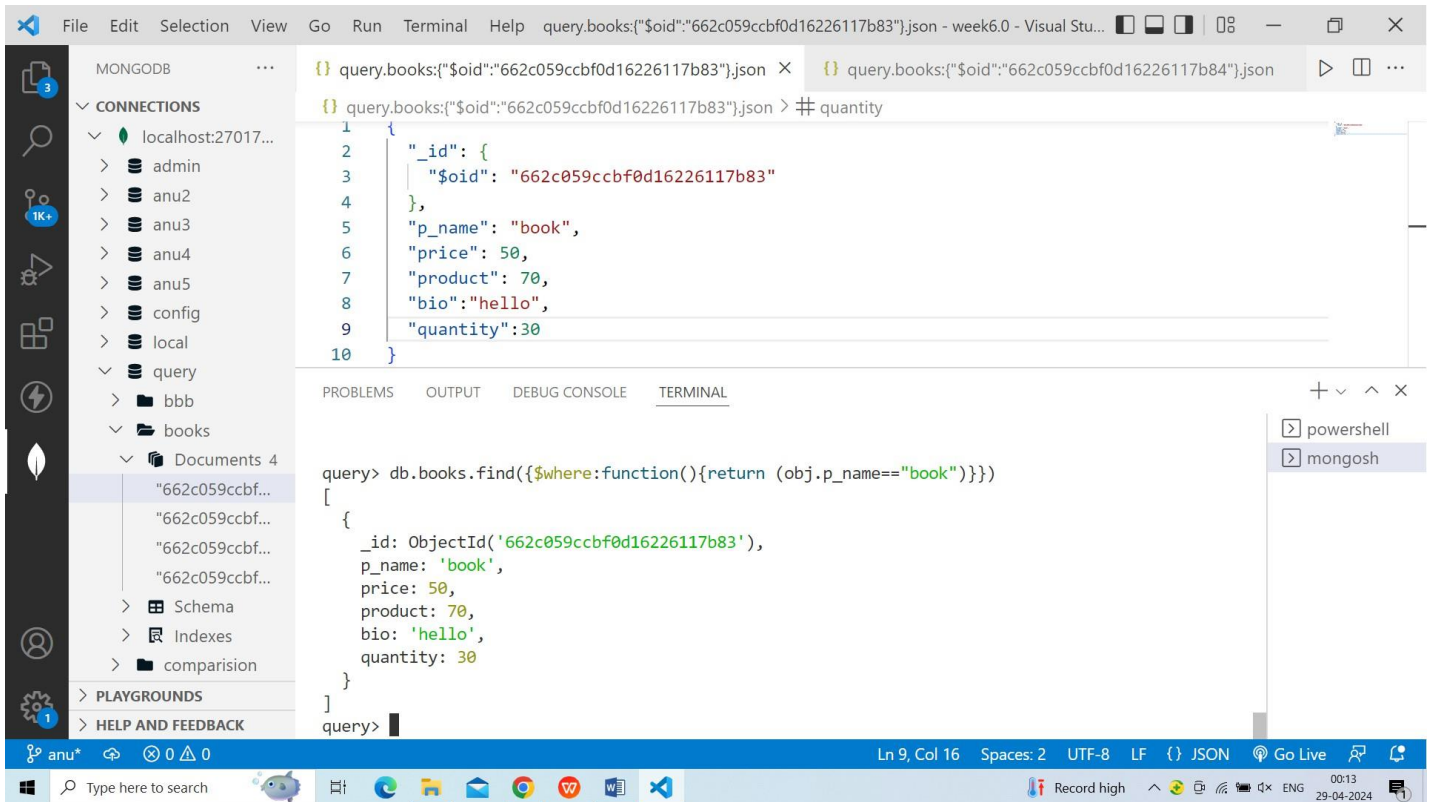
```
query> db.books.createIndex({bio:"text"})
```

## \$where

The "where" operator is used for passing either a string containing a JavaScript expression or a full JavaScript function to the query system.

### Example:

```
db.books.find({$where:function(){return (obj.p_name=="book")}})
```



```
query.books({'$oid':'662c059ccbf0d16226117b83'})
{
 "_id": {
 "$oid": "662c059ccbf0d16226117b83"
 },
 "p_name": "book",
 "price": 50,
 "product": 70,
 "bio": "hello",
 "quantity": 30
}
```

```
query> db.books.find({'$where':function(){return (obj.p_name=="book")}})
[
 {
 _id: ObjectId('662c059ccbf0d16226117b83'),
 p_name: 'book',
 price: 50,
 product: 70,
 bio: 'hello',
 quantity: 30
 }
]
```

## Element Operators

The element operators in the MongoDB return the documents in the collection which returns true if the keys match the fields and datatypes.

There are mainly two Element operators in MongoDB:



Type	Number	Alias	Notes
Double	1	"double"	
String	2	"string"	
Object	3	"object"	
Array	4	"array"	
Binary Data	5	"binData"	
Undefined	6	"undefined"	Deprecated
ObjectId	7	"objectId"	
Boolean	8	"bool"	
Date	9	"date"	
Null	10	"null"	
Regular Expression	11	"regex"	
DBPointer	12	"dbPointer"	Deprecated
JavaScript	13	"javascript"	
Symbol	14	"symbol"	Deprecated
Javascript (with scope)	15	"javascriptWithScope"	
32-bit integer	16	"int"	
Timestamp	17	"timestamp"	
64-bit Integer	18	"long"	
Decimal128	19	"decimal"	New in Version 3.4
Min Key	-1	"minKey"	
Max Key	127	"maxKey"	

## Syntax:

```
{ field: { $type: <BSON type> } }
```

## Example:

```
db.books.find ({ "bookid" : { $type : 2 } });
```

```
query.comparison({"_id":"662c055fcbf0d16226117b80").json - week6.0 - Visual ...
{} query.comparison({"_id":"662c055fcbf0d16226117b80").json > ...
{} query.comparison({"_id":"662c055fcbf0d16226117b80").json > ...
1 {
2 "_id": {
3 "$oid": "662c055fcbf0d16226117b80"
```

```
query> db.books.find ({ "price" : { $type : 16 } });
[
 {
 _id: ObjectId('662c059ccb0d16226117b83'),
 p_name: 'book',
 price: 50
 },
 {
 _id: ObjectId('662c059ccb0d16226117b84'),
 p_name: 'pen',
 price: 100
 },
 {
 _id: ObjectId('662c059ccb0d16226117b85'),
 p_name: 'pencilbox',
 price: 500
 },
 {
 _id: ObjectId('662c059ccb0d16226117b86'),
 p_name: 'ball',
 price: 500
 }
]
```

## Bitwise Operators

The Bitwise operators in the MongoDB return the documents in the MongoDB mainly on the fields that have

numeric values based on their bits similar to other programming languages.

Bitwise Operator	Description	Syntax
\$bitsAllClear	Returns documents where all bits in the specified field are 0.	{ field: { \$bitsAllClear: <bitmask> } }
\$bitsAllSet	Returns documents where all bits in the specified field are 1.	{ field: { \$bitsAllSet: <bitmask> } }
\$bitsAnySet	Returns documents where at least one bit in the specified field is set (1).	{ field: { \$bitsAnySet: <bitmask> } }
\$bitsAnyClear	Returns documents where at least one bit in the specified field is clear (0).	{ field: { \$bitsAnyClear: <bitmask> } }

In the below example, we also specified the positions we wanted.

Example of Using \$bitsAllSet Operator

Let's find the **Person** whose **Age** has bit **1** from position **0 to 4**.

**Query:**

```
db.count_no.find({"Age":{"$bitsAllSet": [0,4] } },{_id:0 });
```

**Output:**

```
Data_base> db.count_no.find({"Age":{"$bitsAllSet":[0,4]}},{_id:0});
[
 { name: 'Shiva', Age: 25, Likes: 2 },
 { name: 'Rama', Age: 23, Likes: 2 },
 { name: 'Hanuman', Age: 23, Likes: 3 }
]
Data_base>
```

*\$bitsAllSet in MongoDB*

**Explanation:** In the above query, we have used **\$bitsAllSet** and it returns documents whose **bits position from 0 to 4 are only ones**. It works only with the **numeric values**. The numeric values will be converted into the bits and the bits numbering takes place from the right.

## Geospatial Operators

The Geospatial operators in the MongoDB are used mainly with the terms that relate to the data which mainly focuses on the directions such as **latitude** or **longitudes**.

The Geospatial operators in the MongoDB are:

Geospatial Operator	Description	Syntax
<b>\$near</b>	Finds geospatial objects near a point. Requires a geospatial index.	<pre>{ \$near: { geometry: &lt;point_geometry&gt;, maxDistance: &lt;distance&gt; (optional) } }</pre>
<b>\$center</b>	(For \$geoWithin with planar geometry) Specifies a circle around a center point	<pre>{ \$geoWithin: { \$center: [&lt;longitude&gt;, &lt;latitude&gt;], radius: &lt;distance&gt; } }</pre>
<b>\$maxDistance</b>	Limits results of \$near and \$nearSphere queries to a maximum distance from the point.	<pre>{ \$near: { geometry: &lt;point_geometry&gt;, maxDistance: &lt;distance&gt; } }</pre>
<b>\$minDistance</b>	Limits results of \$near and \$nearSphere queries to a minimum distance from the point.	<pre>{ \$near: { geometry: &lt;point_geometry&gt;, minDistance: &lt;distance&gt; } }</pre>

## Comment Operators

The \$comment operator in MongoDB is used to **write the comments along with the query in the MongoDB** which is used to easily understand the data.

Comment Operator Example

Let's apply some comments in the queries using the **\$comment** Operator. **Query:**

```
db.collection_name.find({ $comment : comment })
```

**Output:**

```
Data_base> db.count_no.find({$comment:"This is comment"},{_id:0});
[
 { name: 'Krishna', Age: 22, Likes: 1 },
 { name: 'Shiva', Age: 25, Likes: 2 },
 { name: 'Rama', Age: 23, Likes: 2 },
 { name: 'Hanuman', Age: 23, Likes: 3 },
 {
 name: 'Harsha',
 Age: 24,
 Likes: 2,
 Colors: ['Red', 'Green', 'Blue']
 }
]
Data_base>
```

*\$Comment operator in MongoDB*

**Explanation:** In the above query we used the \$comment operator to mention the comment. We have used “**This is a comment**” with \$comment to specify the comment. The comment operator in the MongoDB is used to represent the comment and it increases the understandability of the code.

MongoDB provides a special feature that is known as **Projection**. It allows you to select only the necessary data rather than selecting whole data from the document. For example, a document contains 5 fields, i.e.,

```
{
 name:
 "Roma", age:
 30, branch:
 EEE,
 department: "HR",
 salary: 20000
}
```

But we only want to display the *name* and the *age* of the employee rather than displaying whole details. Now, here we use projection to display the name and age of the employee.

One can use projection with `db.collection.find()` method. In this method, the second parameter is the projection parameter, which is used to specify which fields are returned in the matching documents.

**Syntax:**

```
db.collection.find({}, {field1: value2, field2: value2, ..})
```

- If the value of the field is set to 1 or true, then it means the field will include in the return document.
- If the value of the field is set to 0 or false, then it means the field will not include in the return document.
- You are allowed to use projection operators.
- There is no need to set `_id` field to 1 to return `_id` field, the `find()` method always return `_id` unless you set a `_id` field to 0.

**Examples:**

In the following examples, we are working with:

**Database:** GeeksforGeeks

**Collection:** employee

**Document:** five documents that contain the details of the employees in the form of field-value pairs.





```
[> use GeeksforGeeks
switched to db GeeksforGeeks
[> db.employee.find().pretty()
{
 "_id" : ObjectId("5e49177592e6dfa3fc48dd73"),
 "name" : "Sonu",
 "age" : 26,
 "branch" : "CSE",
 "department" : "HR",
 "salary" : 44000,
 "joiningYear" : 2018
}
{
 "_id" : ObjectId("5e539e0492e6dfa3fc48ddaa"),
 "name" : "Amu",
 "age" : 24,
 "branch" : "ECE",
 "department" : "HR",
 "joiningYear" : 2017,
 "salary" : 25000
}
{
 "_id" : ObjectId("5e539e0492e6dfa3fc48ddab"),
 "name" : "Priya",
 "age" : 24,
 "branch" : "CSE",
 "department" : "Development",
 "joiningYear" : 2017,
 "salary" : 30000
}
{
 "_id" : ObjectId("5e539e0492e6dfa3fc48ddac"),
 "name" : "Mohit",
 "age" : 26,
 "branch" : "CSE",
 "department" : "Development",
 "joiningYear" : 2018,
 "salary" : 30000
}
{
 "_id" : ObjectId("5e539e0492e6dfa3fc48ddad"),
 "name" : "Sumit",
 "age" : 26,
 "branch" : "ECE",
 "department" : "HR",
 "joiningYear" : 2019,
 "salary" : 25000
}
>
```



### Displaying the names of the employees –

```
anki — mongo — 80x55
[> db.employee.find({}, {name: 1}).pretty()
{ "_id" : ObjectId("5e49177592e6dfa3fc48dd73"), "name" : "Sonu" }
{ "_id" : ObjectId("5e539e0492e6dfa3fc48ddaa"), "name" : "Amu" }
{ "_id" : ObjectId("5e539e0492e6dfa3fc48ddab"), "name" : "Priya" }
{ "_id" : ObjectId("5e539e0492e6dfa3fc48ddac"), "name" : "Mohit" }
{ "_id" : ObjectId("5e539e0492e6dfa3fc48ddad"), "name" : "Sumit" }
>]
```

### Displaying the names of the employees without the `_id` field –

```
anki — mongo — 80x55
[> db.employee.find({}, {name: 1, _id: 0}).pretty()
{ "name" : "Sonu" }
{ "name" : "Amu" }
{ "name" : "Priya" }
{ "name" : "Mohit" }
{ "name" : "Sumit" }
>]
```

### Displaying the name and the department of the employees without the `_id` field –

```
anki — mongo — 80x55
[> db.employee.find({}, {name: 1, _id: 0, department: 1}).pretty()
{ "name" : "Sonu", "department" : "HR" }
{ "name" : "Amu", "department" : "HR" }
{ "name" : "Priya", "department" : "Development" }
{ "name" : "Mohit", "department" : "Development" }
{ "name" : "Sumit", "department" : "HR" }
>]
```

## Displaying the names and the department of the employees whose joining year is 2018 –

```
anki — mongo — 80x55
[> db.employee.find({joiningYear: 2018}, {name: 1, department: 1, _id: 0}).pretty()
]
{ "name" : "Sonu", "department" : "HR" }
{ "name" : "Mohit", "department" : "Development" }
>
```

## Projection Operators

Name	Description
<a href="#">\$</a>	Projects the first element in an array that matches the query condition.
<a href="#">\$elemMatch</a>	Projects the first element in an array that matches the specified <a href="#">\$elemMatch</a> condition.
<a href="#">\$meta</a>	Projects the document's score assigned during the <a href="#">\$text</a> operation.
<b>NOTE</b> <a href="#">\$text</a> provides text query capabilities for self-managed (non-Atlas) deployments. For data hosted on MongoDB Atlas, MongoDB offers an improved full-text query solution, <a href="#">Atlas Search</a> .	
<a href="#">\$slice</a>	Limits the number of elements projected from an array. Supports skip and limit slices.

### MongoDB Projection Operator

\$

The \$ operator limits the contents of an array from the query results to contain only the first element matching the query document.

## Syntax:

1. `db.books.find( { <array>: <value> ... },`
2. `{ "<array>.$": 1 } )`
3. `db.books.find( { <array.field>: <value> ...},`
4. `{ "<array>.$": 1 } )`

```
> db.books.find({}, {tags: 1})
{ "_id" : ObjectId("62dc0d1dd9417e55fbc2d41a"), "tags" : ["drama", "horror"] }
{ "_id" : ObjectId("62dc0d28d9417e55fbc2d41b"), "tags" : ["drama"] }
{ "_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "tags" : ["horror", "drama", "suspense"] }
{ "_id" : ObjectId("62dc0d3ad9417e55fbc2d41d"), "tags" : ["suspense"] }
{ "_id" : ObjectId("62dc0d41d9417e55fbc2d41e"), "tags" : ["horror"] }
{ "_id" : ObjectId("62dcfb53efba2194ea59fa04"), "tags" : ["drama", "horror"] }
{ "_id" : ObjectId("62dcfb68efba2194ea59fa05"), "tags" : ["suspense", "horror"] }
{ "_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "tags" : ["suspense", "drama"] }
{ "_id" : ObjectId("62dd8981efba2194ea59fa07"), "tags" : ["cooking"] }
{ "_id" : ObjectId("62e56e939a9b29460d7a2000"), "tags" : [["drama", "horror"]] }
> db.books.find({tags: 'drama'}, {title: 1, tags: 1})
{ "_id" : ObjectId("62dc0d1dd9417e55fbc2d41a"), "title" : "Book 1", "tags" : ["drama", "horror"] }
{ "_id" : ObjectId("62dc0d28d9417e55fbc2d41b"), "title" : "Book 2", "tags" : ["drama"] }
{ "_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "title" : "Book 3", "tags" : ["horror", "drama", "suspense"] }
{ "_id" : ObjectId("62dcfb53efba2194ea59fa04"), "title" : "Book 6", "tags" : ["drama", "horror"] }
{ "_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "title" : "Book 8", "tags" : ["suspense", "drama"] }
> db.books.find({tags: 'drama'}, {title: 1, "tags.$": 1})
{ "_id" : ObjectId("62dc0d1dd9417e55fbc2d41a"), "title" : "Book 1", "tags" : ["drama"] }
{ "_id" : ObjectId("62dc0d28d9417e55fbc2d41b"), "title" : "Book 2", "tags" : ["drama"] }
{ "_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "title" : "Book 3", "tags" : ["drama"] }
{ "_id" : ObjectId("62dcfb53efba2194ea59fa04"), "title" : "Book 6", "tags" : ["drama"] }
{ "_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "title" : "Book 8", "tags" : ["drama"] }
>
{ "_id" : ObjectId("62dcfb68efba2194ea59fa05"), "tags" : ["suspense", "horror"] } : 7, "price" : 15 }, "tags
```

## \$elemMatch

The content of the array field made limited using this operator from the query result to contain only the first element matching the element \$elemMatch condition.

## Syntax:

1. `db.library.find( { bookcode: "63109" },`
2. `{ students: { $elemMatch: { roll: 102 } } } )`

```
{ "_id" : ObjectId("62dc0d41d9417e55fbc2d41e"), "tags" : ["horror"] }
{ "_id" : ObjectId("62dcfb53efba2194ea59fa04"), "tags" : ["drama", "horror"] }
{ "_id" : ObjectId("62dcfb68efba2194ea59fa05"), "tags" : ["suspense", "horror"] }
{ "_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "tags" : ["suspense", "drama"] }
{ "_id" : ObjectId("62dd8981efba2194ea59fa07"), "tags" : ["cooking"] }
{ "_id" : ObjectId("62e56e939a9b29460d7a2000"), "tags" : [["drama", "horror"]] }
> db.books.find({tags: 'drama'}, {title: 1, tags: 1})
{ "_id" : ObjectId("62dc0d1dd9417e55fbc2d41a"), "title" : "Book 1", "tags" : ["drama", "horror"] }
{ "_id" : ObjectId("62dc0d28d9417e55fbc2d41b"), "title" : "Book 2", "tags" : ["drama"] }
{ "_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "title" : "Book 3", "tags" : ["horror", "drama", "suspense"] }
{ "_id" : ObjectId("62dcfb53efba2194ea59fa04"), "title" : "Book 6", "tags" : ["drama", "horror"] }
{ "_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "title" : "Book 8", "tags" : ["suspense", "drama"] }
> db.books.find({tags: 'drama'}, {title: 1, "tags.$": 1})
{ "_id" : ObjectId("62dc0d1dd9417e55fbc2d41a"), "title" : "Book 1", "tags" : ["drama"] }
{ "_id" : ObjectId("62dc0d28d9417e55fbc2d41b"), "title" : "Book 2", "tags" : ["drama"] }
{ "_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "title" : "Book 3", "tags" : ["drama"] }
{ "_id" : ObjectId("62dcfb53efba2194ea59fa04"), "title" : "Book 6", "tags" : ["drama"] }
{ "_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "title" : "Book 8", "tags" : ["drama"] }
> db.books.find({tags: 'drama'}, {title: 1, "tags": {$elemMatch: {$eq: 'horror'}}})
{ "_id" : ObjectId("62dc0d1dd9417e55fbc2d41a"), "title" : "Book 1", "tags" : ["horror"] }
{ "_id" : ObjectId("62dc0d28d9417e55fbc2d41b"), "title" : "Book 2" }
{ "_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "title" : "Book 3", "tags" : ["horror"] }
{ "_id" : ObjectId("62dcfb53efba2194ea59fa04"), "title" : "Book 6", "tags" : ["horror"] }
{ "_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "title" : "Book 8" }
>
{ "_id" : ObjectId("62dcfb68efba2194ea59fa05"), "tags" : ["suspense", "horror"] } : 7, "price" : 15 }, "tags
```

## \$meta

The meta operator returns the result for each matching document where the metadata associated with the query.

### Syntax:

1. { \$meta: <metaDataKeyword> }

### Example:

1. db.books.find(  
2. <query>,  
3. { score: { \$meta: "textScore" } } }

## \$slice

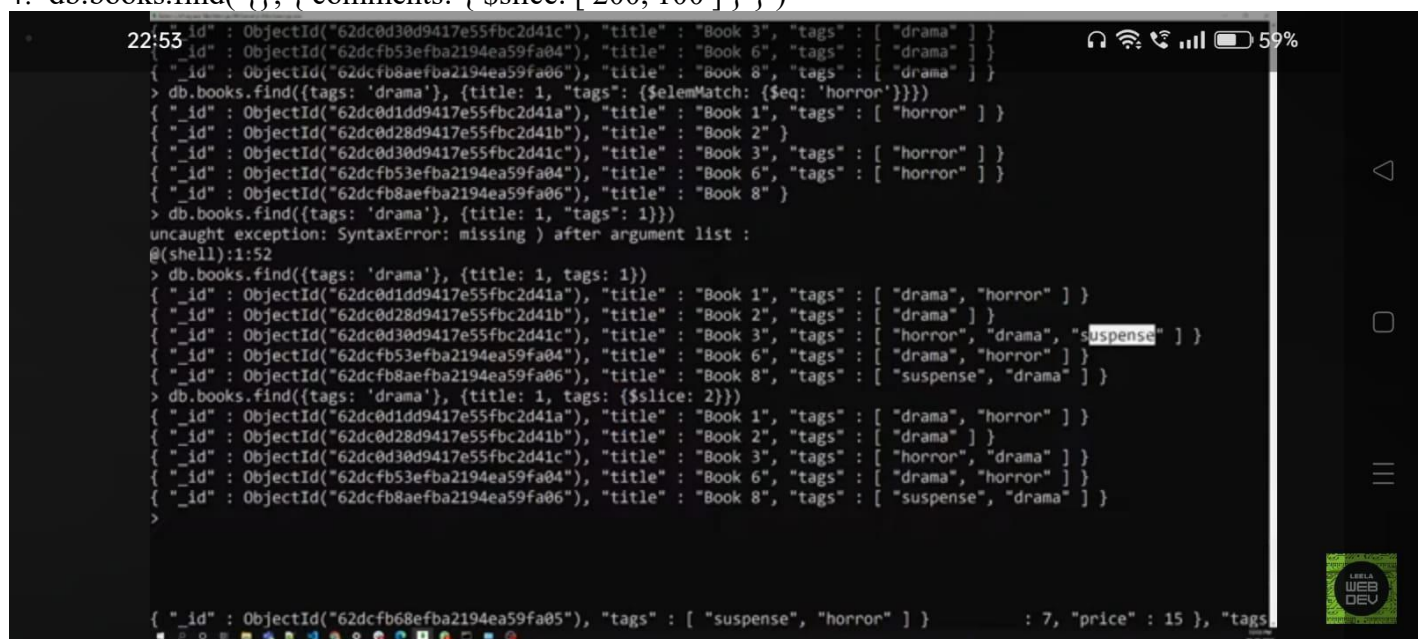
It controls the number of values in an array that a query returns.

### Syntax:

1. db.books.find( { field: value }, { array: { \$slice: count } } );

### Example:

1. db.books.find( {}, { comments: { \$slice: [ 200, 100 ] } } )



```
22:53 ["_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "title" : "Book 3", "tags" : ["drama"] }
["_id" : ObjectId("62dcfb53efba2194ea59fa04"), "title" : "Book 6", "tags" : ["drama"] }
["_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "title" : "Book 8", "tags" : ["drama"] }
> db.books.find({tags: 'drama'}, {title: 1, "tags": {$elemMatch: {$eq: 'horror'}}})
["_id" : ObjectId("62dc0d1dd9417e55fbc2d41a"), "title" : "Book 1", "tags" : ["horror"] }
["_id" : ObjectId("62dc0d28d9417e55fbc2d41b"), "title" : "Book 2"]
["_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "title" : "Book 3", "tags" : ["horror"] }
["_id" : ObjectId("62dcfb53efba2194ea59fa04"), "title" : "Book 6", "tags" : ["horror"] }
["_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "title" : "Book 8"]
> db.books.find({tags: 'drama'}, {title: 1, "tags": 1})
uncaught exception: SyntaxError: missing) after argument list :
@(shell):1:52
> db.books.find({tags: 'drama'}, {title: 1, tags: 1})
["_id" : ObjectId("62dc0d1dd9417e55fbc2d41a"), "title" : "Book 1", "tags" : ["drama", "horror"] }
["_id" : ObjectId("62dc0d28d9417e55fbc2d41b"), "title" : "Book 2", "tags" : ["drama"] }
["_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "title" : "Book 3", "tags" : ["horror", "drama", "suspense"] }
["_id" : ObjectId("62dcfb53efba2194ea59fa04"), "title" : "Book 6", "tags" : ["drama", "horror"] }
["_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "title" : "Book 8", "tags" : ["suspense", "drama"] }
> db.books.find({tags: 'drama'}, {title: 1, tags: {$slice: 2}})
["_id" : ObjectId("62dc0d1dd9417e55fbc2d41a"), "title" : "Book 1", "tags" : ["drama", "horror"] }
["_id" : ObjectId("62dc0d28d9417e55fbc2d41b"), "title" : "Book 2", "tags" : ["drama"] }
["_id" : ObjectId("62dc0d30d9417e55fbc2d41c"), "title" : "Book 3", "tags" : ["horror", "drama"] }
["_id" : ObjectId("62dcfb53efba2194ea59fa04"), "title" : "Book 6", "tags" : ["drama", "horror"] }
["_id" : ObjectId("62dcfb8aefba2194ea59fa06"), "title" : "Book 8", "tags" : ["suspense", "drama"] }
>
["_id" : ObjectId("62dcfb8aefba2194ea59fa05"), "tags" : ["suspense", "horror"] } : 7, "price" : 15 }, "tags"
```

## Aggregation Pipeline

### a. What is Aggregation in MongoDB?

**Aggregation** is a way of processing a large number of documents in a collection by means of passing them through different stages. The stages make up what is known as a pipeline. The stages in a pipeline can filter, sort, group, reshape and modify documents that pass through the pipeline.

One of the most common use cases of Aggregation is to calculate aggregate values for groups of documents. This is similar to the basic aggregation available in SQL with the GROUP BY clause and COUNT, SUM and AVG functions. MongoDB Aggregation goes further though and can also perform relational-like joins, reshape documents, create new and update existing collections, and so on.

There are what are called **single purpose methods** like `estimatedDocumentCount()`, `count()`, and `distinct()` which are appended to a `find()` query making them quick to use but limited in scope.

- Each stage of the pipeline transforms the documents as they pass through it and allowing for operations like **filtering**, **grouping**, **sorting**, **reshaping** and performing calculations on the data.

### b. MongoDB aggregate pipeline syntax

This is an example of how to build an aggregation query:

```
db.collectionName.aggregate(pipeline, options),
```

- where *collectionName* – is the name of a collection,
- *pipeline* – is an array that contains the aggregation stages,
- *options* – optional parameters for the

aggregation This is an example of the

```
pipeline = [

 { $match : { ... } },

 { $group : { ... } },

 { $sort : { ... } }

]
```

aggregation pipeline syntax:

### c. Single-purpose aggregation

- It is used when we need simple access to document like counting the number of documents or for finding all distinct values in a document.
- It simply provides the access to the common aggregation process using the **count()**, **distinct()** and **estimatedDocumentCount()** methods so due to which it lacks the flexibility and capabilities of the pipeline.

#### Example of Single-purpose aggregation

Let's consider a single-purpose aggregation example where we find the total number of users in each city from the users collection.



```
db.users.aggregate([
 { $group: { _id: "$city", totalUsers: { $sum: 1 } } }
])
```

**Output:**

```
[
 { _id: 'Los Angeles', totalUsers: 1 },
 { _id: 'New York', totalUsers: 1 },
 { _id: 'Chicago', totalUsers: 1 }
]
```

In this example, the aggregation pipeline first groups the documents by the city field and then uses the \$sum accumulator to count the number of documents (users) in each city.

The result will be a list of documents, each containing the city (\_id) and the total number of users (totalUsers) in that city.

#### d. How to use MongoDB to Aggregate Data?

To use MongoDB for aggregating data, follow below steps:

1. **Connect to MongoDB:** Ensure you are connected to your MongoDB instance.
2. **Choose the Collection:** Select the collection you want to perform aggregation on, such as students.
3. **Define the Aggregation Pipeline:** Create an array of stages, like \$group to group documents and perform operations (e.g., calculate the average grade).
4. **Run the Aggregation Pipeline:** Use the aggregate method on the collection with your defined pipeline.

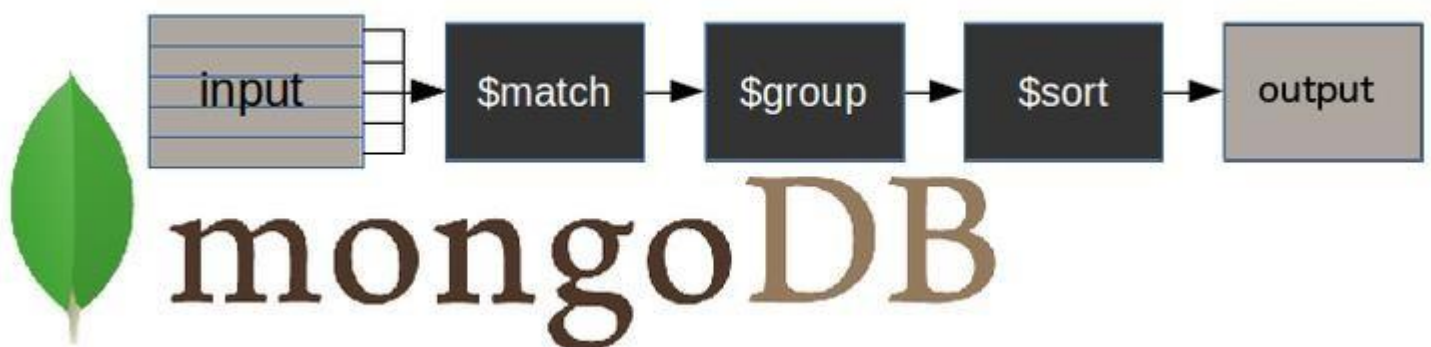
**Example:**

```
db.students.aggregate([
 {
 $group: {
 _id: null,
 averageGrade: { $avg: "$grade" }
 }
 }
])
```

This calculates the average grade of all students in the studentscollection.

#### e. Mongoddb Aggregation Pipeline

##### MongoDB Aggregation Framework



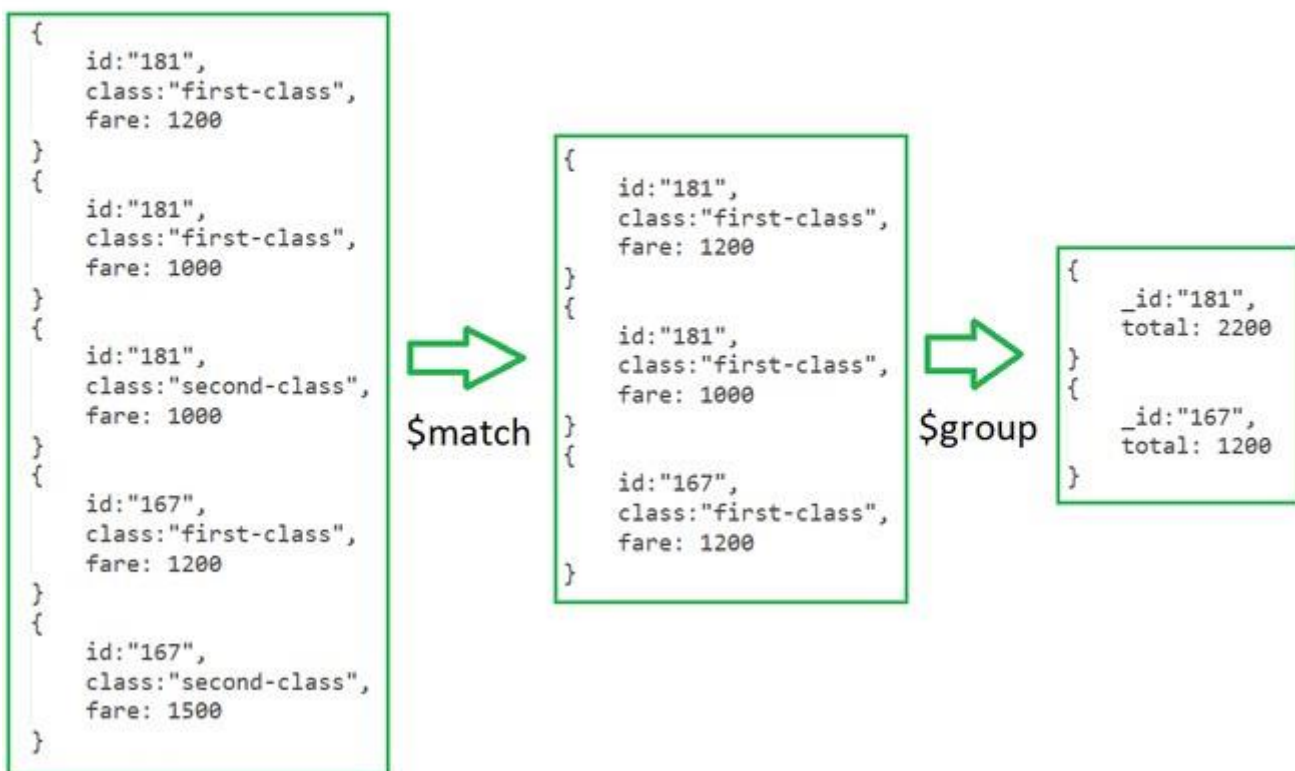
- **Mongoddb Aggregation Pipeline** consist of stages and each stage transforms the document. It is a **multi- stage pipeline** and in each state and the documents are taken as input to produce the resultant set of documents.

- In the next stage (ID available) the resultant documents are taken as input to produce output, this process continues till the last stage.
- The **basic pipeline stages** are defined below:
  1. filters that will operate like queries.
  2. the document transformation that modifies the resultant document.
  3. provide pipeline provides tools for **grouping and sorting documents**.
- Aggregation pipeline can also be used in **sharded collection**.

### Example:

```
db.train.aggregate([
 { $match: { class: "first-class" } },
 { $group: { _id: "id", total: { $sum: "$fare" } } }])
```

} pipeline stages



### Explanation:

In the above example of a collection of “train fares”. **\$match** stage filters the documents by the value in class field i.e. class: “first-class” in the first stage and passes the document to the second stage.

In the Second Stage, the **\$group** stage groups the documents by the id field to calculate the sum of fare for each unique id.

Here, the **aggregate()** function is used to perform aggregation. It can have three operators **stages**, **expression** and **accumulator**. These operators work together to achieve final desired outcome.

```
db.train.aggregate([{ $group : { _id : "$id", total : { $sum : "$fare" } } }])
```

Stage
Expression
Accumulator

#### f. Aggregation Pipeline Method

To understand Aggregation Pipeline Method Let's imagine a collection named **users** with some documents for our examples.

```
{
 "_id": ObjectId("60a3c7e96e06f64fb5ac0700"), "name": "Alice", "age": 30, "email":
 "alice@example.com",
 "city": "New York"
}
{
 "_id":
 ObjectId("60a3c7e96e06f64fb5ac0701"),
 "name": "Bob",
 "age": 35,
 "email": "bob@example.com",
 "city": "Los Angeles"
}
{
 "city": "New York"
}
{
 "_id":
 ObjectId("60a3c7e96e06f64fb5ac0701"),
 "name": "Bob",
 "age": 35,
 "email": "bob@example.com",
 "city": "Los Angeles"
}
{
 "_id":
 ObjectId("60a3c7e96e06f64fb5ac0702"),
 "name": "Charlie",
 "age": 25,
 "email":
 "charlie@example.com",
 "city": "Chicago"
}
```



**\$group:** It [Groups](#) documents by the city field and calculates the average age using the \$avg accumulator.

```
db.users.aggregate([
 { $group: { _id: "$city", averageAge: { $avg: "$age" } } }
])
```

**Output:**

```
[
 { _id: 'New York', averageAge: 30 },
 { _id: 'Chicago', averageAge: 25 },
 { _id: 'Los Angeles', averageAge: 35 }
]
```

**\$project:** Include or exclude fields from the output documents.

```
db.users.aggregate([
 { $project: { name: 1, city: 1, _id: 0 } }
])
```

**Output:**

```
[
 { name: 'Alice', city: 'New York' },
 { name: 'Bob', city: 'Los Angeles' },
 { name: 'Charlie', city: 'Chicago' }
]
```

**\$match:** Filter documents to pass only those that match the specified condition(s).

```
db.users.aggregate([
 { $match: { age: { $gt: 30 } } }
])
```

**Output:**

```
[
 {
 _id:
 ObjectId('60a3c7e96e06f64fb5ac0701'),
 name: 'Bob',
 age: 35,
 email:
 'bob@example.com', city:
 'Los Angeles'
 }
]
```

- **\$sort**: It Order the documents.

```
db.users.aggregate([
 { $sort: { age: 1 } }
])
```

**Output:**

```
[
 {
 _id:
 ObjectId('60a3c7e96e06f64fb5ac0702'),
 name: 'Charlie',
 age: 25,
 email: 'charlie@example.com',
 city: 'Chicago'
 },
 {
 _id:
 ObjectId('60a3c7e96e06f64fb5ac0700'),
 name: 'Alice',
 age: 30,
 email:
 'alice@example.com', city:
 'New York'
 },
 {
 _id:
 ObjectId('60a3c7e96e06f64fb5ac0701'),
 name: 'Bob',
 age: 35,
 email: 'bob@example.com',
 city: 'Los Angeles'
 }
]
```

-

**\$limit:** Limit the number of documents passed to the next stage.

```
db.users.aggregate([
 { $limit: 2 }
])
```

Output:

```
[
 {
 _id:
 ObjectId('60a3c7e96e06f64fb5ac0700'),
 name: 'Alice',
 age: 30,
 email: 'alice@example.com',
 city: 'New York'
 },
 {
 _id:
 ObjectId('60a3c7e96e06f64fb5ac0701'),
 name: 'Bob',
 age: 35,
 email: 'bob@example.com',
 city: 'Los Angeles'
 }
]
```

#### - g. How Fast is MongoDB Aggregation?

- The speed of MongoDB aggregation depends on various factors such as the complexity of the aggregation pipeline, the size of the data set, the hardware specifications of the MongoDB server and the efficiency of the indexes.
- In general, MongoDB's aggregation framework is designed to efficiently process large volumes of data and complex aggregation operations. When used correctly it can provide fast and scalable aggregation capabilities.
- So with any database operation, the performance can vary based on the specific use case and configuration. It is important to optimize our aggregation queries and use indexes where appropriate and ensure that our MongoDB server is properly configured for optimal performance.

---

### How to Insert a Document into a MongoDB Collection using Node.js?

**MongoDB, a popular NoSQL database, offers flexibility** and scalability for handling data. If you're developing a Node.js application and need to interact with MongoDB, one of the fundamental operations you'll perform is inserting a document into a collection.

The steps to insert documents in MongoDB collection are given below

- [NodeJS and MongoDB Connection](#)
- [Create a Collection in MongoDB using Node Js](#)
- [Insert a Single Document](#)
- [Insert Many Document](#)
- [Handling Insertion Results](#)
- [Read Documents from the collection](#)

Steps to Setup the Project

**Step 1:** Create a nodeJS application by using this command

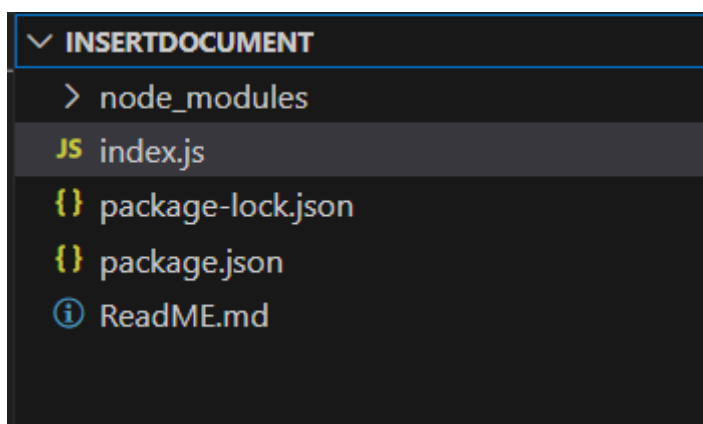
```
npm init
or
npm init -y
```

- **npm init** command asks some setup questions that are important for the project
- **npm init -y** command is used to set all the answers of the setup questions as **yes**.

**Step 2:** Install the necessary packages/libraries in your project using the following commands.

```
npm install mongodb
```

**Project Structure:**



*Project Structure*

The updated dependencies in package.json file will look like:

```
"dependencies": {
 "mongodb": "^6.6.1"
}
```

## NodeJS and MongoDB Connection

Once the MongoDB is installed we can use MongoDB database with the Nodejs Project. Initially we need to specify the database name, connection URL and the instance of MongoClient.

```
const { MongoClient } = require('mongodb');
// or as an ECMAScript module:
// import { MongoClient } from 'mongodb'

// Connection URL
const url =
'mongodb://localhost:27017'; const
client = new MongoClient(url);

const dbName = 'project_name'; //

Database Name
async function main() {

 await client.connect();
 console.log('Connected successfully to server');

 const db = client.db(dbName);
 const collection = db.collection('collection_name');

 // Can Add the CRUD operations
}

main().then(console.log)
 .catch(console.error)
 .finally(() => client.close());
```

- **MongoClient** class provided method, to connect MongoDB and Nodejs.
- **client** is the instance of MongoDB and Node Js connection.
- **client.connect()** is used to connect to MongoDB database, it **awaits** until the connection is established.

## Create a Collection in MongoDB using Node Js

In this operation we create a collection inside a database. Initially we specify the database in which collections are to be created.

```
//Specify Database
const dbName = 'database_name';

const db = client.db(dbName);

//Create Collection
const collection = db.collection('collection_name');
```

- **client** is the instance of the connection which provides the **db()** method to create a new Database.
- **collection()** method is used to set the instance of the collection .

### Insert a Single Document

To insert a document into the collection **insertOne()** method is used.

```
const insertDoc = await
 collection.insertOne({ filed1: value1,
 field2: value2,
 });
```

```
//Insert into collection
console.log('Inserted documents =>', insertDoc);
```

### Insert Many Documents

To insert a document into the collection **insertMany()** method is used.

```
const doc_array = [
 { document1 },
 { document2 },
 { document3 },
];

//Insert into collection
const insertDoc =
 await collection.insertMany(doc_array);
console.log('Inserted documents =>',
insertDoc);
```

### Handling Insertion Results

In a project we have different tasks which needs to be executed in specific order. In the MongoDB and Node Js project we must ensure that connection is set. While performing insertion of documents , we perform asynchronous insertion so that execution is not interrupted. We use try-catch block to handle errors while setting up connection, inserting document or while performing any other operation. If an error occurs during execution ,catch block handles it or provide the details about the error ,which helps to resolve the error.

```

try {
 const dbName =
'database_name'; await
client.connect();
const collection = db.collection('collection_name');

const doc_array = [
 { document1 },
 { document2 },
 { document3 },
];

//Insert into collection
const insertDoc = await collection.insertMany(doc_array);

 console.log('Inserted documents =>', insertDoc);
} catch (error) {
 console.error('Error:',

```

- Initially connection is established .AS the connection is established insertMany() method or insertOne() method is used to insert the document in the collection.
- **insertDoc** stores the result of the insertion which is further logged.

### Read Documents from the collection

We can read the documents inside the collection using the find() method.

```

const doc = await
collection.find({}).toArray();

```

find() method is used to along with empty {} are used to read all the documents in the collection. Which are further converted into the array using the toArray() method.

### Closing the Connection

```

finally{ client.close()
}

```

Once the promise is resolved or rejected , code in finally block is executed. The **close()** method is used to close the connection.

Connection is closed irrespective of the error .It is generally used to cleanup and release the resource.

**Example:** Implementation to show Insertion of documents into a MongoDB collection using Node.js JavaScript

```

const { MongoClient } = require("mongodb");

async function main() {
 const url = "mongodb://127.0.0.1:27017";
 const dbName = "GeeksforGeeks";
 const studentsData = [
 { rollno: 101, Name: "Raj ", favSub: "Math" },
 { rollno: 102, Name: "Yash", favSub: "Science" },
 { rollno: 103, Name: "Jay", favSub: "History" },
];

 let client = null;

```

```

try {
 // Connect to MongoDB
 client = await MongoClient.connect(url);
 console.log("Connected successfully to MongoDB");

 const db = client.db(dbName);
 const collection = db.collection("students");

 // Add students to the database
 await collection.insertMany(studentsData);
 console.log("Three students added successfully");

 // Query all students from the database
 const students = await collection.find().toArray();
 console.log("All students:", students);
} catch (err) { console.error("Error:", err); }
finally {
 // Close the connection
 if (client) {
 client.close();
 ;
 console.log("Connection closed successfully");
 }
}
}

main();

```

**Output:**

```

Connected successfully to MongoDB
Three students added successfully
All students: [
 {
 _id: new ObjectId('665c48ccd4397ee8dc0af355'),
 rollno: 101,
 Name: 'Raj ',
 favSub: 'Math'
 },
 {
 _id: new ObjectId('665c48ccd4397ee8dc0af356'),
 rollno: 102,
 Name: 'Yash',
 favSub: 'Science'
 },
 {
 _id: new ObjectId('665c48ccd4397ee8dc0af357'),
 rollno: 103,
 Name: 'Jay',
 favSub: 'History'
 }
]
Connection closed successfully

```



### **Explanation :**

In the above example, Initially **MongoClient** class is imported which is used to connect MongoDB and Nodejs .**client** is the instance of MongoDB and Node Js connection. which is used to name the database .As database is set ,**collection()** method sets the instance of the collection .Three documents are inserted in the **students** collection using **insertMany()** method .Error during the execution are handled using the try catch block ,finally connection is closed using the **close() method**

## **Using Mongoose for Structured Schema and Validation**

**Mongoose** is a MongoDB object modeling and handling for a node.js environment. **Mongoose Validation** is essentially a customizable middleware that gets defined inside the SchemaType of Mongoose schema. It automatically fires off before a document is saved in the NoSQL DB. Validation can also be run manually using doc.validate(callback) or doc.validateSync() methods.

Types of mongoose validation:

### **In Mongoose there are two types of validation:**

1. Built-in validation
2. Custom validation

To get a better understanding of how built-in validators work, let's look at the following:

#### **1. Built-in validators:**

**A. Required validator:** A schema uses required whenever it is mandatory to fill that field with any value.

Required validator takes an array with 2 items, first a Boolean var and a message to return the validation if it fails.

```
required: [true, "user name is required"]
```

we can also specify the Required validator without custom error message,

```
required: true
```

**B. Unique validator:** Unique is not a validator, but an option.

If the unique option is set, Mongoose will require each document to have a unique value for each path.

Unique option takes an array with 2 items, first a Boolean var and a custom error message.

```
unique: [true, 'email already exists']
```

we can also specify unique option without custom error message,

```
unique: true
```

#### **2. Custom validators**

In addition to the built-in validators, you can define custom validators. In custom validation, a validation function is passed along with the declaration. Defining a custom validator involves creating a specialized

function within the schema's field definition to ensure that the data inserted or updated meets specific criteria beyond the standard validation rules offered by Mongoose.

Steps to create node application And Installing Mongoose:

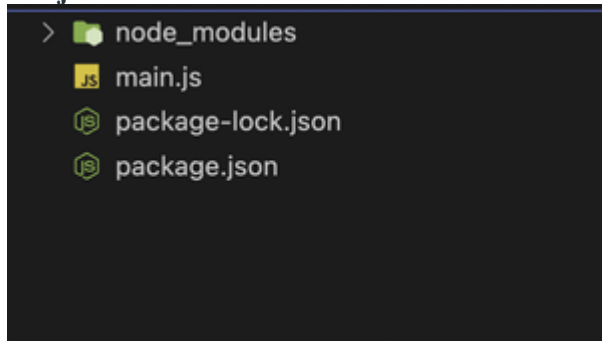
**Step 1:** Create a node application using the following command:

```
mkdir folder_name
cd folder_name
npm init -y
touch main.js
```

**Step 2:** After creating the NodeJS application, Install the required module using the following command:

```
npm install mongoose
```

**Project Structure:** It will look like the following.



The updated **dependencies** in package.json file will look like:

```
"dependencies": {
 "express": "^4.18.2"
}
```

**Example 1:** In this example, we will use a “required” validator to check whether a value is passed to the document or not before saving it to the DB.

// main.js

```
const mongoose = require('mongoose')
```

// Database connection

```
mongoose.connect('mongodb://localhost:27017/query-helpers', {
 dbName: 'event_db',
 useNewUrlParser: true,
 useUnifiedTopology: true
}, err => err ? console.log(err) :
 console.log('Connected to database'));
```

```
const personSchema = new mongoose.Schema({
 name: {
 type: String,
```

```

 required: true
 }
});
const Person = mongoose.model('Person', personSchema);
const person = new Person({});
(async () => {
 try {
 await person.save();
 } catch (err) {
 console.log(err)
 }
})();

```

**Step to Run Application:** Run the application using the following command from the root directory of the project:

```
node main.js
```

#### Output:

```

> node main.js
Error: Person validation failed: name: Path `name` is required.
 at ValidationError.inspect (/Users/dishebh/Downloads/temp/mongo_modules/mongoose/lib/error/validation.js:49:26)
 at formatValue (node:internal/util/inspect:763:19)
 at inspect (node:internal/util/inspect:340:10)
 at formatWithOptionsInternal (node:internal/util/inspect:200:10)
 at formatWithOptions (node:internal/util/inspect:1888:10)
 at console.value (node:internal/console/constructor:323:14)
 at console.log (node:internal/console/constructor:359:61)
 at /Users/dishebh/Downloads/temp/mongo/main.js:25:13
 at processTicksAndRejections (node:internal/process/task_queues:95:5)
) {
 errors: {
 name: ValidatorError: Path `name` is required.

```