UNIT I INTRODUCTION TO CSS and JAVASCRIPT

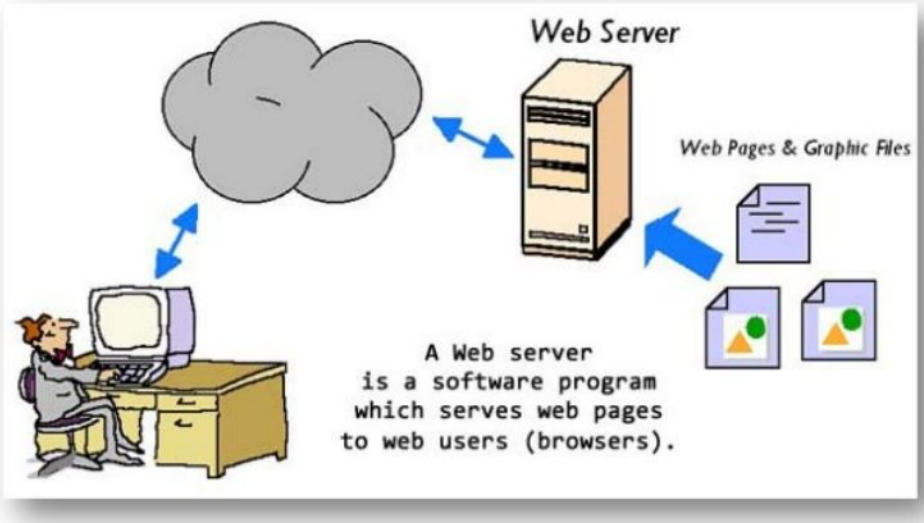
Introduction to Web: Server - Client - Communication Protocol (HTTP) – Structure of HTML Documents – Basic Markup tags – Working with Text and Images with CSS– CSS Selectors – CSS Flexbox - JavaScript: Data Types and Variables - Functions - Events – AJAX: GET and POST.

1.1 Introduction to Web: Server - Client

A web server is a software program that serves web pages to web users (browsers).

A web server delivers requested web pages to users who enter the URL in a web browser. Every computer on the internet that contains a web site must have a web server program.

The computer in which a web server program runs is also usually called a "web server". So, the term "web server" is used to represent both the server program and the computer in which the server program runs.



Characteristics of web servers

A web server computer is just like any other computer.
The basic characteristics of web servers are:

- It is always connected to the internet so that clients can access the web pages hosted by the web server.
- It always has an application called "web server" running.

In short, a "web server" is a computer that is connected to the internet/intranet and has software called "web server". The web server program will always be running in the computer. When a user tries to access a website hosted by the web server, it is actually the web server program that delivers the web page that the client asks for.

All web sites in the internet are hosted in web servers sitting in various parts of the world.

Is a Web Server hardware or software?

Mostly, Web server refers to the software program, that serves the clients request. But sometimes, the computer in which the web server program is installed is also called a "web server".

Web Server, Behind the Scenes

When I type in an URL such as http://www.ASP.NET and click on some link, I dropped into this page.

But what happens behind the scenes to bring you to this page and make you read this line of text.

So now, let's see what is actually happening behind the scenes.

The first you might do is, you type the http://www.asp.net/ in the address bar of your browser and press your return key.

We could break this URL into the following two parts:

1. The protocol we will use to connect to the server (http)
2. The server name (ASP.NET)

And the following process happens:

- The browser breaks up the URL into these parts and then it tries to communicate with the server looking up for the server name.
- The server is identified through a unique IP address but the alias for the IP address is maintained in the DNS Server or the Naming server.
- The browser looks up these naming servers, identifies the IP address of the server requested and gets the site and gets the HTML tags for the web page.
- Finally it displays the HTML Content in the browser.

Where is my web server?

When you try to access a web site, you don't really need to know where the web server is located. The web server may be located in another city or country, but all you need to do is, type the URL of the web site you want to access in a web browser. The web browser will send this information to the internet and find the web server. Once the web server is located, it will request the specific web page from the web server program running in the server. The Web server program will process your request and send the resulting web page to your browser. It is the responsibility of your browser to format and display the web page to you.

How many web servers are needed for a web site?

Typically, there is only one web server required for a web site. But large web sites like Yahoo, Google, MSN and so on will have millions of visitors every minute. One computer cannot process such huge numbers of requests. So, they will have hundreds of servers deployed in various parts of the world so that can provide a faster response.

How many web sites can be hosted in one server?

A web server can host hundreds of web sites. Most of the small web sites in the internet are hosted on shared web servers. There are several web hosting companies who offer shared web hosting. If you buy a shared web hosting from a web hosting company, they will host your web site in their web server along with several other web sites for a fee.

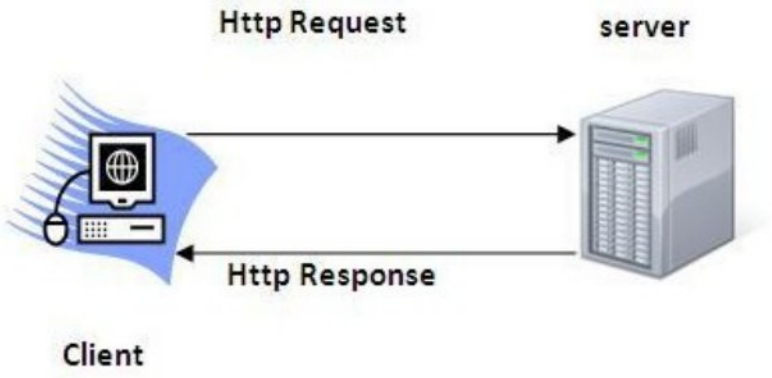
Examples of web server applications:

1. IIS
2. Apache

1.2 Communication Protocol (HTTP)

The Hypertext Transfer Protocol (HTTP) is application-level protocol for collaborative, distributed, hypermedia information systems. It is the data communication protocol used to establish communication between client and server.

HTTP is TCP/IP based communication protocol, which is used to deliver the data like image files, query results, HTML files etc on the World Wide Web (WWW) with the default port is TCP 80. It provides the standardized way for computers to communicate with each other.



The Basic Characteristics of HTTP (Hyper Text Transfer Protocol):

- It is the protocol that allows web servers and browsers to exchange data over the web.
- It is a request response protocol.
- It uses the reliable TCP connections by default on TCP port 80.
- It is stateless means each request is considered as the new request. In other words, server doesn't recognize the user by default.

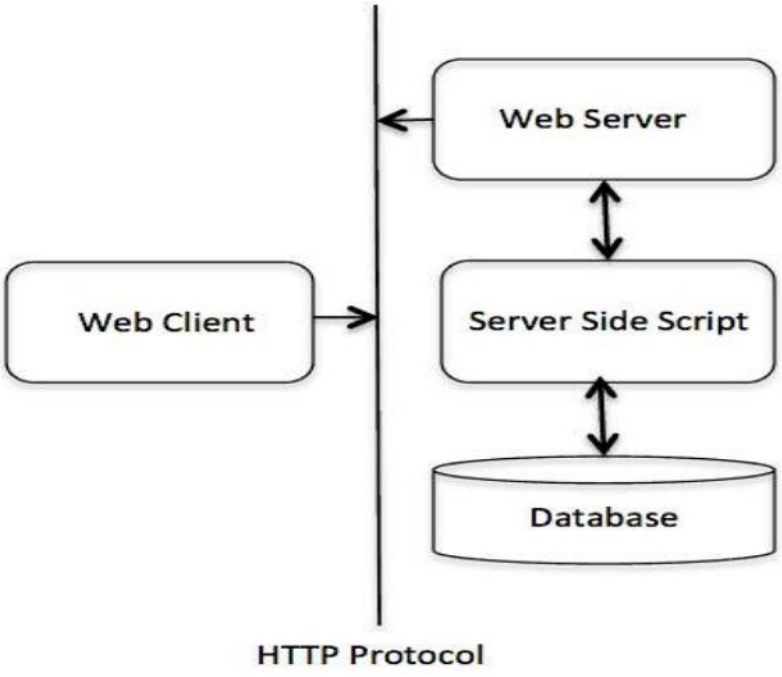
The Basic Features of HTTP (Hyper Text Transfer Protocol):

There are three fundamental features that make the HTTP a simple and powerful protocol used for communication:

- **HTTP is media independent:** It specifies that any type of media content can be sent by HTTP as long as both the server and the client can handle the data content.
- **HTTP is connectionless:** It is a connectionless approach in which HTTP client i.e., a browser initiates the HTTP request and after the request is sent the client disconnects from server and waits for the response.
- **HTTP is stateless:** The client and server are aware of each other during a current request only. Afterwards, both of them forget each other. Due to the stateless nature of protocol, neither the client nor the server can retain the information about different request across the web pages.

The Basic Architecture of HTTP (Hyper Text Transfer Protocol):

The below diagram represents the basic architecture of web application and depicts where HTTP stands:



HTTP is request/response protocol which is based on client/server based architecture. In this protocol, web browser, search engines, etc. behave as HTTP clients and the Web server like Servlet behaves as a server

1.3 Structure of HTML Documents

Here you will learn about document structure of an HTML document. The figure given below shows the general structure of an HTML document.

```
<html>
   <head>
      <title>Title</title>
   </head>
   <body>
     <h1>Main Heading</h1>
     <p>Paragraph</p>
     <h2>Sub Heading</h2>
     <p>Paragraph</p>
   </body>
</html>
```

Texts between the BODY tag (<body> and </body>) will be displayed in or by the browser.

Basic Structure of an HTML Document

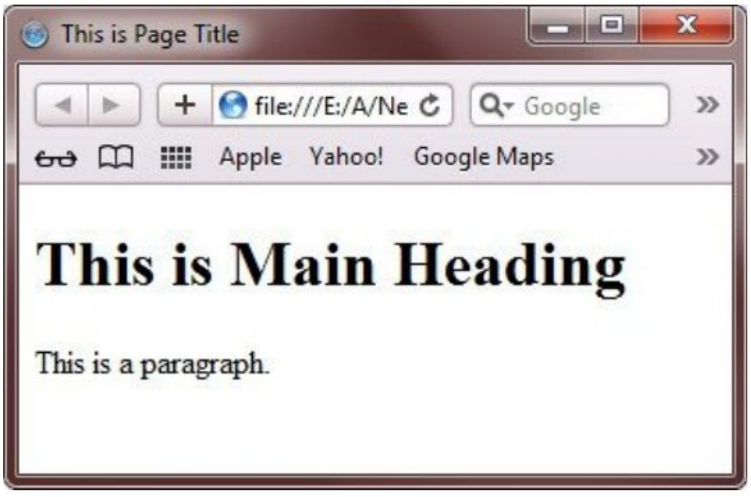
Here is an example shows the basic structure of an HTML document.

```
<!DOCTYPE html>
<html>
<head>
  <title>This is Page Title</title>
</head>
<body>

<h1>This is Main Heading</h1>
<p>This is a paragraph.</p>

</body>
</html>
```

To start HTML coding, open your text editor like Notepad for windows user. Type the above HTML code or just do copy and paste.

After typing/copying, save it as filename.htm or filename.html in you computer. Now open saved HTML document in a web browser to watch output webpage.

You will watch the following given HTML output webpage on your browser.



Here is the explanation of the above HTML document structure example:

- The **DOCTYPE** declaration defines the document type to be HTML
- The text between **<html>** and **</html>** describes an HTML document
- The text between **<head>** and **</head>** provides information about the HTML document
- The text between **<title>** and **</title>** provides a title for the HTML document
- The text between **<body>** and **</body>** describes the visible page content i.e. the content which is visible in the browser.

- The text between **<h1>** and **</h1>** describes the main heading
- The text between ** ** and ** ** describes a paragraph

1.4 Basic Markup tags

HTML tags are like keywords which defines that how web browser will format and display the content. With the help of tags, a web browser can distinguish between an HTML content and a simple content. HTML tags contain three main parts: opening tag, content and closing tag. But some HTML tags are unclosed tags.

When a web browser reads an HTML document, browser reads it from top to bottom and left to right. HTML tags are used to create HTML documents and render their properties. Each HTML tags have different properties.

An HTML file must have some essential tags so that web browser can differentiate between a simple text and HTML text. You can use as many tags you want as per your code requirement.

- All HTML tags must enclosed within < > these brackets.
- Every tag in HTML perform different tasks.
- If you have used an open tag <tag>, then you must use a close tag </tag> (except some tags)

Unclosed HTML Tags

Some HTML tags are not closed, for example br and hr.

**
 Tag**: br stands for break line, it breaks the line of the code.

<hr> Tag: hr stands for Horizontal Rule. This tag is used to put a line across the webpage.

HTML Meta Tags

DOCTYPE, title, link, meta and style

HTML Text Tags

, <h1>, <h2>, <h3>, <h4>, <h5>, <h6>, , , <abbr>, <acronym>, <address>, <bdo>, <blockquote>, <cite>, <q>, <code>, <ins>, <del>, <dfn>, <kbd>, <pre>, <samp>, <var> and

HTML Link Tags

<a> and <base>

HTML Image and Object Tags

<img>, <area>, <map>, <param> and <object>

HTML List Tags

, <ol>, <li>, <dl>, <dt> and <dd>

open tag	Close tag	Description	Example
		This tag allows you to create paragraphs	My name is Fred. I live in Medway
<h1>	</h1>	This is the largest heading	**Heading 1**
<h2>	</h2>	This is second biggest	**Heading 2**

		heading	
<h3>	</h3>	This is the next heading	**Heading 3**
<h4>	</h4>	This is another heading	***Heading 4***
<h5>	</h5>	This is the second smallest heading	Heading 5
<h6>	</h6>	This is the smallest heading	*Heading 6*
<hr >	n/a	This is a horizontal line. You can use width and size attributes	
		This makes text bold	**Bold text**
		This makes text italic	*Italic text*
 	n/a	This tag allows you to insert line breaks	**abc** **def**

1.5 Working with Text and Images with CSS

2.16. 1.TEXT COLOR

Text-color property is used to set the color of the text.

Text-color can be set by using the name "red", hex value "#ff0000" or by its RGB value"rgb(255, 0, 0).

Syntax:

```
body
{
color:color name;
}
```

Example: Html

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

<style>
```

```
h1{

color:red;

}

h2{ color:green;

}

</style></head>

<body>

<h1>
APEC
</h1>
<h2> MCA
</h2>
</body>
</html>
```

APEC

MCA

2.TEXT ALIGNMENT

Text alignment property is used to set the horizontal alignment of the text.

The text can be set to left, right, centered and justified alignment.

In justified alignment, line is stretched such that left and right margins are straight.

Syntax:

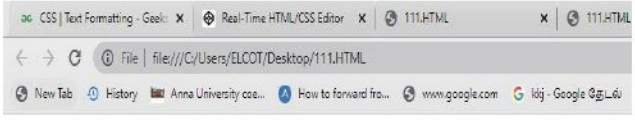
```
body
```

```
{
text-align:alignment type;
}
```

Example: html

```
<!DOCTYPE html>

<html>

<head>

<style> h1

{

color:red;

text-align:center;

}
h2
{
color:gren;

 textalign:left;

}
</style>
</head>
<body>
<h1>

GEEKS FOR GEEKS

</h1>
```

```
<h2>

TEXT FORMATTING

</h2>

</body>

</html>
```



APEC

TEXT FORMATTING

3. TEXT-weight

The font-weight property sets the weight, or thickness, of a font and is dependent either on available font faces within a font family or weights defined by the browser.

```
span {
   font-weight: bold;
}
```

The font-weight property accepts either a keyword value or predefined numeric value. The available keywords are:

- normal
- bold
- bolder
- lighter

The available numeric values are:

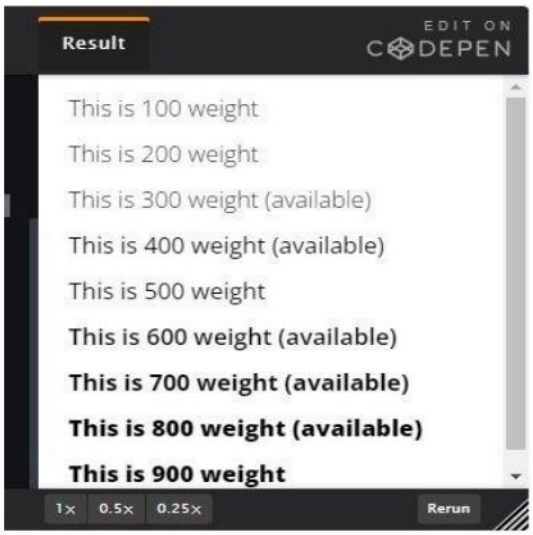
- 100
- 200
- 300
- 400
- 500
- 600
- 700
- 800
- 900

The keyword value normal maps to the numeric value 400 and the value bold maps to 700.

In order to see any effect using values other than 400 or 700, the font being used must have built-in faces that match those specified weights.

If a font has a bold (“700”) or normal (“400”) version as part of the font family, the browser will use that. If those are not available, the browser will mimic its own bold or normal version of the font. It will not mimic the other unavailable weights. Fonts often use names like “Regular” and “Light” to identify any alternate font weights.

The following demo demonstrates the use of the alternate weight values:



4. Text font

The font property is a shorthand property for:

- font-style
- font-variant
- font-weight
- font-size/line-height
- font-family

The font-size and font-family values are required. If one of the other values is missing, their default value are used

Property Values

Property/Value	**Description**
font-style	Specifies the font style. Default value is "normal"
font-variant	Specifies the font variant. Default value is "normal"
font-weight	Specifies the font weight. Default value is "normal"
font-size/line-height	Specifies the font size and the line-height. Default value is "normal"

font-family	Specifies the font family. Default value depends on the browser
Caption	Uses the font that are used by captioned controls (like buttons, drop-downs, etc.)
Icon	Uses the font that are used by icon labels
Menu	Uses the fonts that are used by dropdown menus
message-box	Uses the fonts that are used by dialog boxes
small-caption	A smaller version of the caption font
status-bar	Uses the fonts that are used by the status bar
Initial	Sets this property to its default value. Read about *initial*
Inherit	Inherits this property from its parent element. Read about *inherit*

Example:

```
<!DOCTYPE html>
<html>
<body>
```

```
<h1>The font Property</h1>


<p style="font:caption">The font used in captioned controls.</p>
<p style="font:icon">The font used in icon labels.</p>
```

```
<p style="font:menu">The font used in dropdown menus.</p>

<p style="font:message-box">The font used in dialog boxes.</p>
<p style="font:small-caption">A smaller version of the caption font.</p>
<p style="font:status-bar">The font used in the status bar.</p>
<p><b>Note:</b> The result of the font keywords is browser dependant.</p>


</body>
</html>
```

Output:

The font Property

The font used in captioned controls.

The font used in icon labels.

The font used in dropdown menus.

The font used in dialog boxes.

A smaller version of the caption font.

The font used in the status bar.

Note: The result of the font keywords is browser dependant.

Css working with images

Images play an important role in any webpage. Though it is not recommended to include a lot of images, but it is still important to use good images wherever required.

CSS plays a good role to control image display. You can set the following image properties using CSS.

- The **border** property is used to set the width of an image border.
- The **height** property is used to set the height of an image.
- The **width** property is used to set the width of an image.
- The **-moz-opacity** property is used to set the opacity of an image.

The Image Border Property

The *border* property of an image is used to set the width of an image border. This property can have a value in length or in %.

A width of zero pixels means no border.

Here is the example −

```
  <head>

  </head>


  <body>

    <img style = "border:0px;" src =
"C:\Users\JANAKIRAMAN\OneDrive\Desktop\1.png" />

    <br />

    <img style = "border:3px dashed red;" src =
"C:\Users\JANAKIRAMAN\OneDrive\Desktop\1.png" />

  </body>

</html>
```

It will produce the following result −

The Image Height Property

The *height* property of an image is used to set the height of an image. This property can have a value in length or in %. While giving value in %, it applies it in respect of

Here is an example –

```
<html>
  <head>
  </head>
  <body>
    <img style = "border:1px solid red; height:100px;" src =
"C:\Users\JANAKIRAMAN\OneDrive\Desktop\1.png" />
    <br />
    <img style = "border:1px solid red; height:50%;" src =
"C:\Users\JANAKIRAMAN\OneDrive\Desktop\1.png" />
  </body>
</html>
```

It will produce the following result –

The Image Width Property

The *width* property of an image is used to set the width of an image. This property can have a value in length or in %. While giving value in %, it applies it in respect of the box in which an image is available.

Here is an example −

```
<html>
```

```
  <head>
  </head>

  <body>
    <img style = "border:1px solid red; width:50px;" src =
"C:\Users\JANAKIRAMAN\OneDrive\Desktop\1.png" />
    <br />
    <img style = "border:1px solid red; width:25%;" src =
"C:\Users\JANAKIRAMAN\OneDrive\Desktop\1.png" />
  </body>
</html>
```

It will produce the following result –

1.6 CSS Selectors

There are many different types of CSS selector that allow you to target rules to specific elements in an HTML document. The table on the opposite page introduces the most commonly used CSS selectors. On this page, there is an HTML file to demonstrate which elements these CSS selectors would apply to. CSS selectors are case sensitive, so they must match element names and attribute values exactly. There are some more advanced selectors which allow you to select elements based on attributes and their values, which you will see on page 292. IE 7 was the first version of IE to support the last two selectors in the table (the sibling selectors), so their use is less common than the other selectors shown here.

There are several different types of selectors in CSS.

1. CSS Element Selector

2. CSS Id Selector
3. CSS Class Selector
4. CSS Universal Selector
5. CSS Group Selector

1) CSS Element Selector

The element selector in CSS is used to select HTML elements which are required to be styled. In a selector declaration, there is the name of the HTML element and the CSS properties which are to be applied to that element is written inside the brackets {}.

Syntax:

element {
\\ CSS property
}

Example :

```
<!DOCTYPE html>

<html>

<head>

<title>element selector</title>

<style>

/* h1 element selected here */

 h1 {

color:green;

text-align:center;

} /* h2 element selected here */

 h2 {

text-align:center;

}
```

```
</style>

</head>

<body>

<h1>GeeksforGeeks</h1>
<h2>element Selector</h2>

</body>

</html>
```

Output:

2) CSS class Selector

The ***.class*** **selector** is used to select all elements which belong to a particular class attribute. In order to select the elements with a particular class, use the period (.) character specifying the class name ie., it will match the HTML element based on the contents of their class attribute. The class name is mostly used to set the CSS property to a given class.

Syntax:

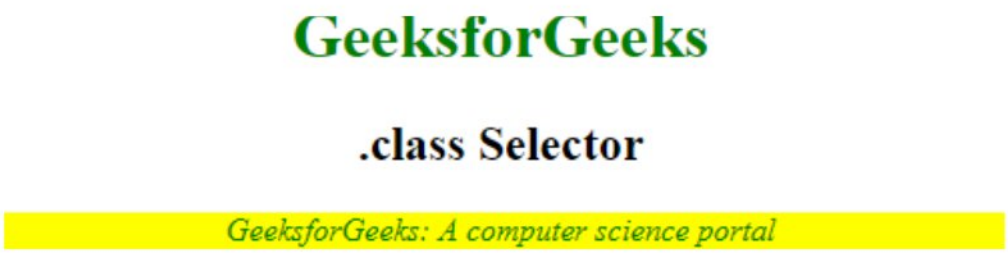
```
.class {
// CSS property
}
```

Example : This example demonstrates the **class Selector** for the specific HTML element.

```
<!DOCTYPE html>

<html><head><style>

.geeks {

color: green;
```

```
.gfg {

background-color: yellow;

font-style: italic; color: green;

}

</style>

</head>

<body style="text-align:center">

<h1 class="geeks">GeeksforGeeks</h1>

<h2>.class Selector</h2>

<div class="gfg">

<p>GeeksforGeeks: A computer science portal</p>

</div></body></html>
```

Output:



3. css id(#) selector

The #id selector is used to set the style of given id. The id attribute is the unique identifier in HTML document. The id selector is used with # character.

Syntax:

```
#id {
// CSS property
}
```

Example:

```
<!DOCTYPE html>

<html><head>

<title>#id selector</title>

<!-- CSS property using id attribute -->

<style>

#gfg1 {

color:green;

text-align:center;

}

#gfg2 {

text-align:center;

}

</style>

</head><body>

<!-- id attribute declare here -->

<h1 id = "gfg1">GeeksforGeeks</h1>

<h2 id = "gfg2">#id selector</h2>

</body></html>
```

Output:

GeeksforGeeks

4. Css universal(*) selector

The * selector in CSS is used to select all the elements in a HTML document. It also selects all elements which are inside under another element. It is also called universal selector.

Syntax:

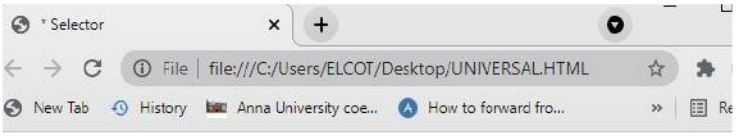
```
* {
// CSS property
}
```

Example :

```
<!DOCTYPE html>

<html><head>

<title>* Selector</title>

<!-- CSS property of * selector -->

<style>

* { color:green;

text-align:center;

}

</style>

</head><body>

<h1>Adhiparasakthi Engineering college</h1>

<h2>*(Universal) Selector</h2>

<div>

<p>MBA</p>

<p>MCA</p>

</div>

<p>COMPUTER APPLICATION</p>
```

```
</body></html>
```

Output:



Adhiparasakthi engineering college

***(Universal) Selector**

MBA

MCA

COMPUTER APPLICATION.

5.Group-selector:

This selector is used to style all comma separated elements with the same style. **style.css:** The following code is used in the above HTML code using the group selector. Suppose you want to apply common styles to different selectors, instead of writing rules separately you can write them in groups as shown below.

```
#div-container, .paragraph-class, h1{ color: white;
background-color: purple; font-family: monospace;
}
```

Example :

```
<!DOCTYPE html>
<html lang="en">

<head>
<link rel="stylesheet" href="style.css">
</head><body>
<h1>

</h1>

<p>

</p>

Sample Heading
```

```
Geeks for Geeks is a computer science

<div id="div-container">
Geeks for geeks is a computer science

</div>

<p class="paragraph-class">Geeks for geeks is a computer science
</p>
</body>

</html>
```

1.7 CSS Flexbox

What is CSS flexbox ?

CSS Flexible Layout Box, popularly known as **Flexbox** is a powerful one-dimensional layout model. It helps to lay, align and distribute items (children) efficiently inside a container (parent).

Important Features:

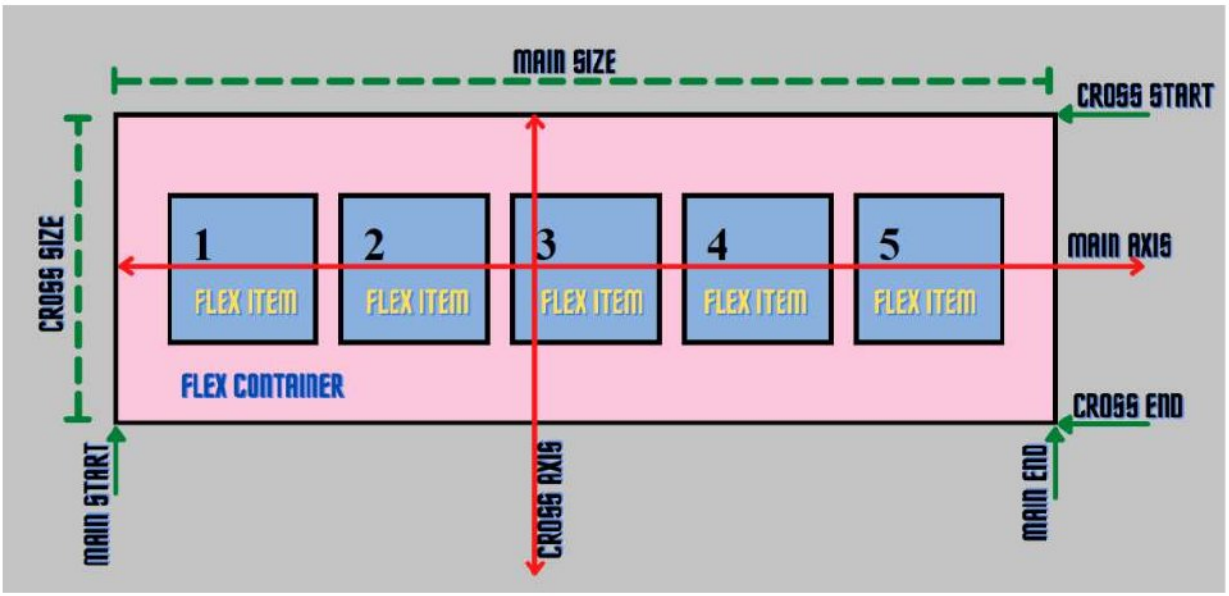
- **One-dimensional layout model:** Flex is a one-dimensional layout model as it can only deal with items either horizontally as rows or vertically as columns. On the contrary, the CSS Grid layout can handle rows and columns together.
- **Creates flexible and responsive layouts**: Flexbox gives flex container the ability to customize the items within it, depending on different screen sizes. A flex container can expand its children's items to fill the available space or it can also shrink them to prevent overflow.
- **Direction-agnostic:** Flexbox is free from any directional constraints unlike Block (vertically biased) and Inline (horizontally biased).
- **Super easy to use:** It is easy to align items in Flexbox, unlike using float and positioning which are a little frustrating and sometimes difficult to use.

Flexbox Architecture:

There are two aspects of a Flexbox: **Flex container** and **Flex item**

The flex items can be laid out either along the main axis (starting from the main start and ending at the main end) or along the cross axis (starting from the cross start and ending at the cross end).

- **Main axis:** Flex items are laid out along this axis, either horizontally or vertically based upon the flex-direction.
- **Cross axis:** It is perpendicular to the main axis and its direction depends on the direction of the main axis.
- **Main size:** It is the width/height of the flex item depending on the main dimension.
- **Cross size:** It is the width/height of the flex item depending on the cross dimension.



To understand the different Flexbox properties, let us take an example by creating an HTML file, along with a CSS file.

Example:

.HTML

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width,
                initial-scale=1.0" />
    <title>CSS Flexbox</title>
    <link rel="stylesheet" href="style.css" />
</head>

<body>
    <div class="container">
        <div class="item item-1">1</div>
        <div class="item item-2">2</div>
        <div class="item item-3">3</div>
        <div class="item item-4">4</div>
    </div>
</body>
```

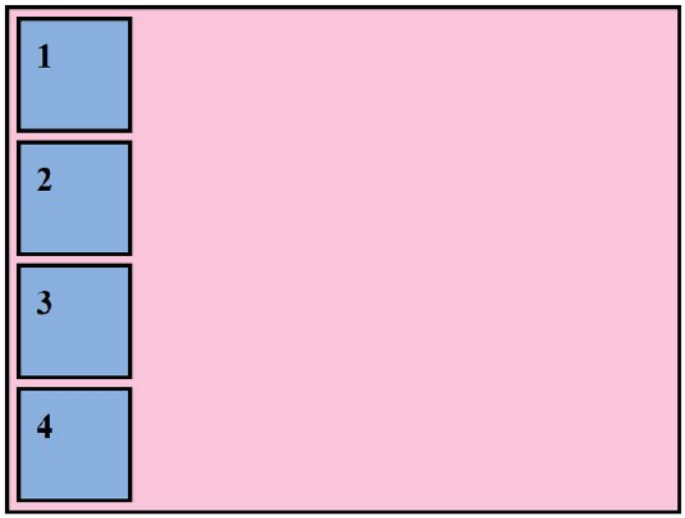
```
</html>
```

This is our CSS code in which we will be styling the flex-container and flex-item.

.CSS

```
.container {
  border: 5px solid rgb(0, 0, 0);
  background-color: rgb(245 197 221);
}
.item {
  border: 5px solid rgb(0, 0, 0);
  background-color: rgb(141, 178, 226);
  margin: 10px;
  padding: 20px;
  height: 100px;
  width: 100px;
  font-weight: bold;
  font-size: 45px;
}
```

Output:



From the above output, the items are aligned vertically, by default, and the default display is *block-level*. The pink area is the *container* and the blue boxes within it are the *items*.

- **flex-direction:** It sets the direction of the flex container's main axis and specifies how items will be placed inside the container.

flex-direction: attribute value

Attribute Values:

- **row:** Flex items are displayed horizontally along a row.
- **column:** Flex items are displayed vertically along a column.
- **row reverse:** Flex items are displayed horizontally along a row but in reverse order.
- **column reverse:** Flex items are displayed vertically along a column but in reverse order.

Note: The display direction, by default, is row.

- **flex-wrap:** It specifies whether the flex container will have a single line or have multiple lines.

Syntax:

flex-wrap: attribute value

Attribute values:

- **nowrap (default):** It specifies that the flex items will not wrap and will be laid out in a single line. It may cause the flex container to overflow.
- **wrap:** It specifies that the flex items will wrap if necessary, and will be laid out in multiple lines.
- **wrap-reverse:** It is the same as a wrap, but the flex items will wrap in reverse order in this case.
- **initial:** It represents the value specified as the property's initial value.
- **inherit:** It represents the computed value of the property on the element's parent.

1.8 JavaScript: Data Types and Variables

What is JavaScript

JavaScript (js) is a light-weight object-oriented programming language which is used by several websites for scripting the webpages. It is an interpreted, full-fledged programming language that enables dynamic interactivity on websites when applied to an HTML document. It was introduced in the year 1995 for adding programs to the webpages in the Netscape Navigator browser. Since then, it has been adopted by all other graphical web browsers. With JavaScript, users can build modern web applications to interact directly without reloading the page every time. The traditional website uses js to provide several forms of interactivity and simplicity.

JavaScript Data Types

JavaScript provides different **data types** to hold different types of values. There are two types of data types in JavaScript.

1. Primitive data type
2. Non-primitive (reference) data type

JavaScript is a **dynamic type language**, means you don't need to specify type of the variable because it is dynamically used by JavaScript engine. You need to use **var** here to specify the data type. It can hold any type of values such as numbers, strings etc.

There are eight basic data types in JavaScript. They are:

Data Types	**Description**	**Example**
String	represents textual data	'hello', "hello world!" etc
Number	an integer or a floating-point number	3, 3.234, 3e-2 etc.
BigInt	an integer with arbitrary precision	900719925124740999n , 1n etc.
Boolean	Any of two values: true or false	true and false
Undefined	a data type whose variable is not initialized	let a;
Null	denotes a null value	let a = null;

	whose instances are unique and immutable	Symbol('hello');
Object	key-value pairs of collection of data	let student = { };

Here, all data types except Object are primitive data types, whereas Object is non-primitive.

JavaScript Variable

1. JavaScript variable
2. JavaScript Local variable
3. JavaScript Global variable

A **JavaScript variable** is simply a name of storage location. There are two types of variables in JavaScript : local variable and global variable.

There are some rules while declaring a JavaScript variable (also known as identifiers).

1. Name must start with a letter (a to z or A to Z), underscore(_), or dollar($) sign.
2. After first letter we can use digits (0 to 9), for example value1.
3. JavaScript variables are case sensitive, for example x and X are different variables.

Example

```
<script>
var x = 10;
var y = 20;
var z=x+y;
document.write(z);
</script>
```

30.

JavaScript local variable

A JavaScript local variable is declared inside block or function. It is accessible within the function or block only. For example:

```
<script>
function abc(){
var x=10;//local variable
}
</script>
```

JavaScript global variable

A **JavaScript global variable** is accessible from any function. A variable i.e. declared outside the function or declared with window object is known as global variable. For example:

```
<script>
var data=200;//gloabal variable
function a(){
document.writeln(data);
}
function b(){
document.writeln(data);
}
a();//calling JavaScript function
b();
</script>
```

Output:

200 200

1.9 Functions

A function is a group of reusable code which can be called anywhere in your program. This eliminates the need of writing the same code again and again. It helps programmers in writing modular codes. Functions allow a programmer to

Like any other advanced programming language, JavaScript also supports all the features necessary to write modular code using functions. You must have seen functions like **alert()** and **write()** in the earlier chapters. We were using these functions again and again, but they had been written in core JavaScript only once.

JavaScript allows us to write our own functions as well. This section explains how to write your own functions in JavaScript.

Function Definition

Before we use a function, we need to define it. The most common way to define a function in JavaScript is by using the **function** keyword, followed by a unique function name, a list of parameters (that might be empty), and a statement block surrounded by curly braces.

Syntax

The basic syntax is shown here.

```
<script type = "text/javascript">
<!--
function functionname(parameter-list)
{
statements
}
//-->
</script>
```

Example

Try the following example. It defines a function called sayHello that takes no parameters −

```
<script type = "text/javascript">
   <!--
      function sayHello()
 {
         alert("Hello there");
      }
   //-->
</script>
```

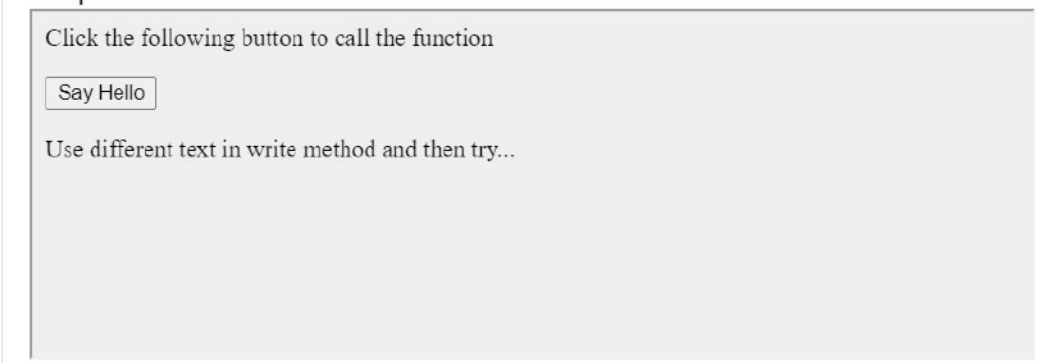
Calling a Function

To invoke a function somewhere later in the script, you would simply need to write the name of that function as shown in the following code.

```
<html>
   <head>
      <script type = "text/javascript">
         function sayHello() {
```

```
      }
    </script>
</head>

  <body>
    <p>Click the following button to call the function</p>
    <form>
      <input type = "button" onclick = "sayHello()" value = "Say Hello">
    </form>
    <p>Use different text in write method and then try...</p>
  </body>
</html>
```

Output



1.10 Events

What is an Event ?

JavaScript's interaction with HTML is handled through events that occur when the user or the browser manipulates a page.

When the page loads, it is called an event. When the user clicks a button, that click too is an event. Other examples include events like pressing any key, closing a window, resizing a window, etc.

Developers can use these events to execute JavaScript coded responses, which cause buttons to close windows, messages to be displayed to users, data to be validated, and virtually any other type of response imaginable.

Events are a part of the Document Object Model (DOM) Level 3 and every HTML element contains a set of events which can trigger JavaScript Code.

Please go through this small tutorial for a better understanding HTML Event Reference. Here we will see a few examples to understand a relation between Event and JavaScript −

onclick Event Type

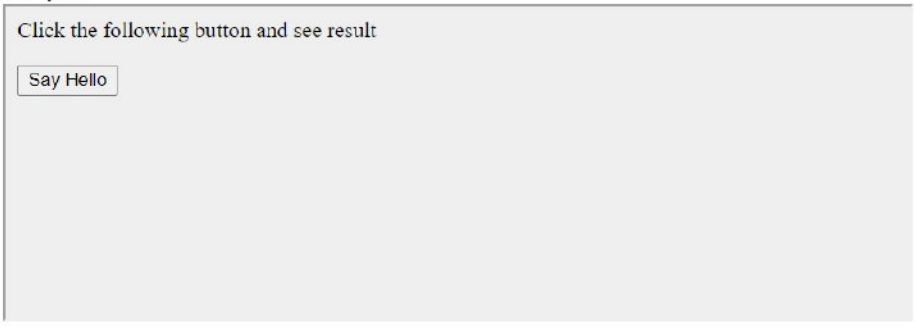
This is the most frequently used event type which occurs when a user clicks the left button of his mouse. You can put your validation, warning etc., against this event type.

Example

Try the following example.

```
<html>
  <head>
    <script type = "text/javascript">
      <!--
         function sayHello() {
           alert("Hello World")
         }
      //-->
    </script>
  </head>

  <body>
    <p>Click the following button and see result</p>
    <form>
      <input type = "button" onclick = "sayHello()" value = "Say Hello" />
    </form>
  </body>
</html>
```

Output



wo commonly used methods for a request-response between a client and server are: GET and POST.

- **GET** - Requests data from a specified resource
- **POST** - Submits data to be processed to a specified resource

GET is basically used for just getting (retrieving) some data from the server. **Note:** The GET method may return cached data.

POST can also be used to get some data from the server. However, the POST method NEVER caches data, and is often used to send data along with the request.

To learn more about GET and POST, and the differences between the two methods, please read our HTTP Methods GET vs POST chapter.

jQuery $.get() Method

The $.get() method requests data from the server with an HTTP GET request.

Syntax:

$.get(*URL,callback*);

The required URL parameter specifies the URL you wish to request.

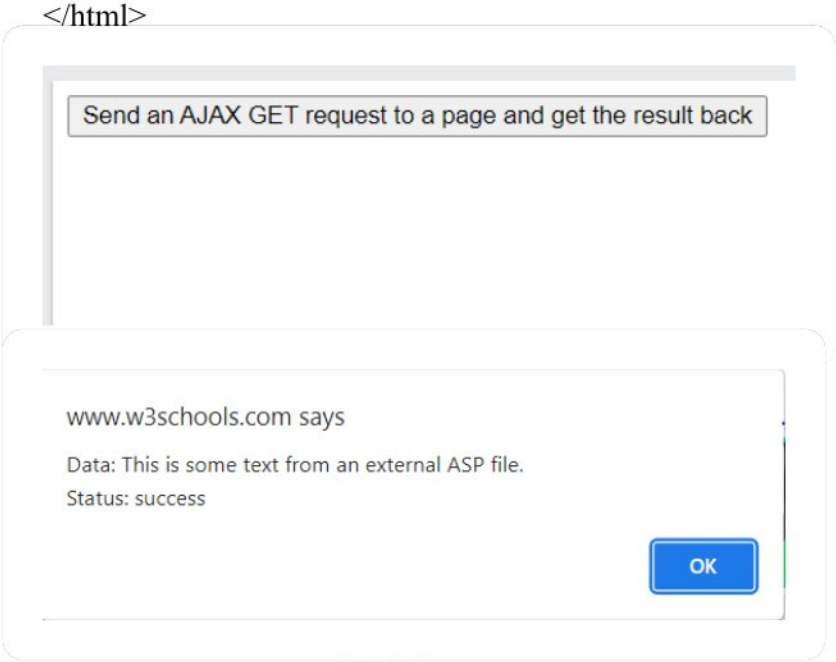
The optional callback parameter is the name of a function to be executed if the request succeeds.

The following example uses the $.get() method to retrieve data from a file on the server:

Example

```
<!DOCTYPE html>

<html>

<head>

<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>

<script>

$(document).ready(function(){
```

```
    $.get("demo_test.asp", function(data, status){
      alert("Data: " + data + "\nStatus: " + status);
    });
  });
});
</script>
</head>
<body>
<button>Send an AJAX GET request to a page and get the result back</button>


</body>
</html>
```



The first parameter of $.get() is the URL we wish to request ("demo_test.asp").

The second parameter is a callback function. The first callback parameter holds the content of the page requested, and the second callback parameter holds the status of the request.

Tip: Here is how the ASP file looks like ("demo_test.asp"):

```
<%
response.write("This is some text from an external ASP file.")
```

jQuery $.post() Method

The $.post() method requests data from the server using an HTTP POST request.

Syntax:

$.post(*URL,data,callback*);

The required URL parameter specifies the URL you wish to request.

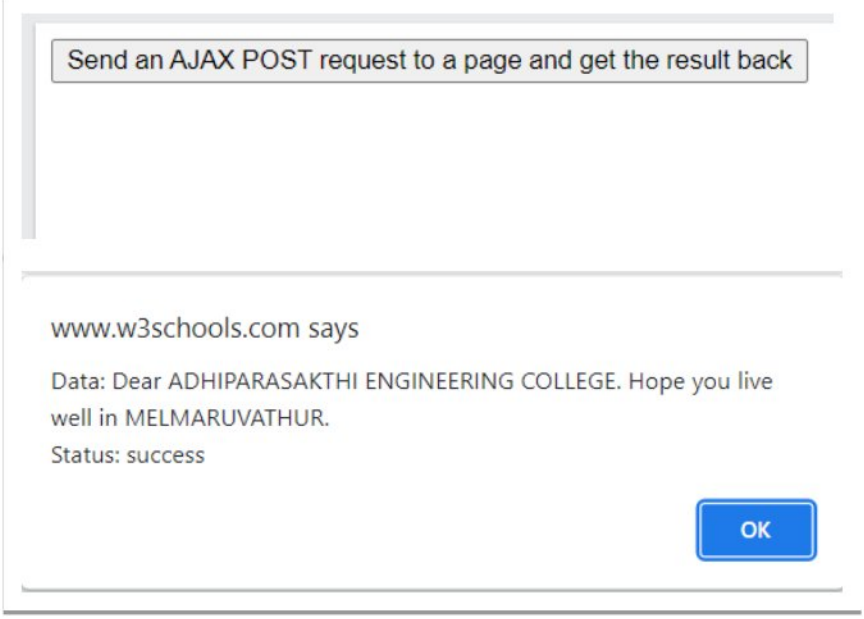
The optional data parameter specifies some data to send along with the request.

The optional callback parameter is the name of a function to be executed if the request succeeds.

The following example uses the $.post() method to send some data along with the request:

Example

```
$("button").click(function(){
  $.post("demo_test_post.asp",
  {
    name: "ADHIPARASAKTHI ENGINEERING COLLEGE",
    city: "MELMARUVATHUR"
  },
  function(data, status){
    alert("Data: " + data + "\nStatus: " + status);
  });
});
```



The first parameter of $.post() is the URL we wish to request ("demo_test_post.asp").

Then we pass in some data to send along with the request (name and city).

The ASP script in "demo_test_post.asp" reads the parameters, processes them, and returns a result.

The third parameter is a callback function. The first callback parameter holds the content of the page requested, and the second callback parameter holds the status of the request.

Tip: Here is how the ASP file looks like ("demo_test_post.asp"):

```
<%
dim fname,city
fname=Request.Form("name")
city=Request.Form("city")
Response.Write("Dear " & fname & ". ")
Response.Write("Hope you live well in " & city & ".")
%>
```

UNIT II SERVER SIDE PROGRAMMING WITH NODE JS

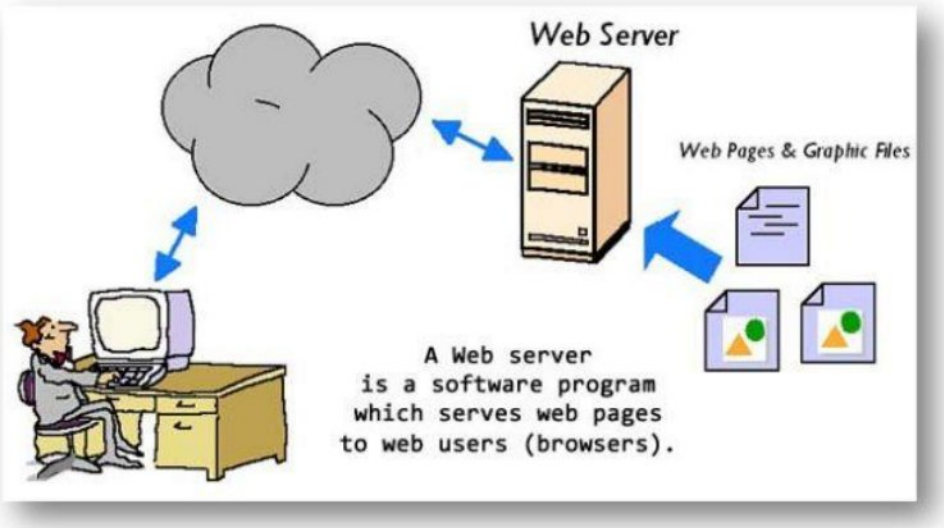
Introduction to Web Servers – Javascript in the Desktop with NodeJS – NPM – Serving files with the http module – Introduction to the Express framework – Server-side rendering with Templating Engines – Static Files - async/await - Fetching JSON from Express.

2.1 Introduction to Web Servers

A web server is a software program that serves web pages to web users (browsers).

A web server delivers requested web pages to users who enter the URL in a web browser. Every computer on the internet that contains a web site must have a web server program.

The computer in which a web server program runs is also usually called a "web server". So, the term "web server" is used to represent both the server program and the computer in which the server program runs.



Characteristics of web servers

A web server computer is just like any other computer.

The basic characteristics of web servers are:

- It is always connected to the internet so that clients can access the web pages hosted by the web server.
- It always has an application called "web server" running.

In short, a "web server" is a computer that is connected to the internet/intranet and has software called "web server". The web server program will always be running in the computer. When a user tries to access a website hosted by the web server, it is actually the web server program that delivers the web page that the client asks for.

All web sites in the internet are hosted in web servers sitting in various parts of the world.

Is a Web Server hardware or software?

Mostly, Web server refers to the software program, that serves the clients request. But sometimes, the computer in which the web server program is installed is also called a "web server".

Web Server, Behind the Scenes

When I type in an URL such as http://www.ASP.NET and click on some link, I dropped into this page.

But what happens behind the scenes to bring you to this page and make you read this line of text.

So now, let's see what is actually happening behind the scenes.

The first you might do is, you type the http://www.asp.net/ in the address bar of your browser and press your return key.

We could break this URL into the following two parts:

1. The protocol we will use to connect to the server (http)
2. The server name (ASP.NET)

And the following process happens:

- The browser breaks up the URL into these parts and then it tries to communicate with the server looking up for the server name.
- The server is identified through a unique IP address but the alias for the IP address is maintained in the DNS Server or the Naming server.
- The browser looks up these naming servers, identifies the IP address of the server requested and gets the site and gets the HTML tags for the web page.
- Finally it displays the HTML Content in the browser.

Where is my web server?

When you try to access a web site, you don't really need to know where the web server is located. The web server may be located in another city or country, but all you need to do is, type the URL of the web site you want to access in a web browser. The web browser will send this information to the internet and find the web server. Once the web server is located, it will request the specific web page from the web server program running in the server. The Web server program will

responsibility of your browser to format and display the web page to you.

How many web servers are needed for a web site?

Typically, there is only one web server required for a web site. But large web sites like Yahoo, Google, MSN and so on will have millions of visitors every minute. One computer cannot process such huge numbers of requests. So, they will have hundreds of servers deployed in various parts of the world so that can provide a faster response.

How many web sites can be hosted in one server?

A web server can host hundreds of web sites. Most of the small web sites in the internet are hosted on shared web servers. There are several web hosting companies who offer shared web hosting. If you buy a shared web hosting from a web hosting company, they will host your web site in their web server along with several other web sites for a fee.

Examples of web server applications:

1. IIS
2. Apache

2.2 Java script in the Desktop with NodeJS

2.3 NPM

NPM – or "Node Package Manager" – is the default package manager for JavaScript's runtime Node.js..

NPM consists of two main parts:

- a CLI (command-line interface) tool for publishing and downloading packages, and
- an online repository that hosts JavaScript packages

Node Package Manager (NPM) provides two main functionalities −

- Online repositories for node.js packages/modules which are searchable on search.nodejs.org
- Command line utility to install Node.js packages, do version management and dependency management of Node.js packages.

NPM comes bundled with Node.js installables after v0.6.3 version. To verify the same, open console and type the following command and see the result −

```
$ npm --version
2.7.1
```

If you are running an old version of NPM then it is quite easy to update it to the latest version. Just use the following command from root −

```
$ sudo npm install npm -g
/usr/bin/npm -> /usr/lib/node_modules/npm/bin/npm-cli.js
```

Installing Modules using NPM

There is a simple syntax to install any Node.js module −

$ npm install <Module Name>

For example, following is the command to install a famous Node.js web framework module called express −

$ npm install express

Now you can use this module in your js file as following −

var express = require('express');

Global vs Local Installation

By default, NPM installs any dependency in the local mode. Here local mode refers to the package installation in node_modules directory lying in the folder where Node application is present. Locally deployed packages are accessible via require() method. For example, when we installed express module, it created node_modules directory in the current directory where it installed the express module.

$ ls -l
total 0
drwxr-xr-x 3 root root 20 Mar 17 02:23 node_modules

Alternatively, you can use **npm ls** command to list down all the locally installed modules.

Globally installed packages/dependencies are stored in system directory. Such dependencies can be used in CLI (Command Line Interface) function of any node.js but cannot be imported using require() in Node application directly. Now let's try installing the express module using global installation.

$ npm install express -g

This will produce a similar result but the module will be installed globally. Here, the first line shows the module version and the location where it is getting installed.

express@4.12.2 /usr/lib/node_modules/express
├── merge-descriptors@1.0.0
├── utils-merge@1.0.0
├── cookie-signature@1.0.6
├── methods@1.1.1
├── fresh@0.2.4
└── cookie@0.1.2

You can use the following command to check all the modules installed globally −

$ npm ls -g

Using package.json

package.json is present in the root directory of any Node application/module and is used to define the properties of a package. Let's open package.json of express package present in **node_modules/express/**

```
{
  "name": "express",
    "description": "Fast, unopinionated, minimalist web framework",
    "version": "4.11.2",
    "author": {

      "name": "TJ Holowaychuk",
      "email": "tj@vision-media.ca"
    },
```

Attributes of Package.json

- **name** − name of the package
- **version** − version of the package
- **description** − description of the package
- **homepage** − homepage of the package
- **author** − author of the package
- **contributors** − name of the contributors to the package
- **dependencies** − list of dependencies. NPM automatically installs all the dependencies mentioned here in the node_module folder of the package.
- **repository** − repository type and URL of the package
- **main** − entry point of the package
- **keywords** − keywords

Uninstalling a Module

Use the following command to uninstall a Node.js module.

$ npm uninstall express

Once NPM uninstalls the package, you can verify it by looking at the content of /node_modules/ directory or type the following command −

$ npm ls

Updating a Module

Update package.json and change the version of the dependency to be updated and run the following command.

$ npm update express

Search a Module

Search a package name using NPM.

$ npm search express

Create a Module

Creating a module requires package.json to be generated. Let's generate package.json using NPM, which will generate the basic skeleton of the package.json.

$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

See 'npm help json' for definitive documentation on these fields
and exactly what they do.

Use 'npm install <pkg> --save' afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (webmaster)

You will need to provide all the required information about your module. You can take help from the above-mentioned package.json file to understand the meanings of various information demanded. Once package.json is generated, use the following command to register yourself with NPM repository site using a valid email address.

$ npm adduser`1`
Username: mcmohd
Password:
Email: (this IS public) mcmohd@gmail.com

It is time now to publish your module −

$ npm publish

2.4 Serving files with the http module

One of the most fundamental uses of an HTTP server is to serve static files to a user's browser, like CSS, JavaScript, or image files. Beyond normal browser usage, there are thousands of other reasons you'd need to serve a static files, like for downloading music or scientific data. Either way, you'll need to come up with a simple way to let the user download these files from your server.

One simple way to do this is to create a Node HTTP server. As you probably know, Node.js excels at handling I/O-intensive tasks, which makes it a natural choice here. You can choose to create your own simple HTTP server from the base `http` module that's shipped with Node, or you can use the popular `serve-static` package, which provides many common features of a static file server.

The end goal of our static server is to let the user specify a file path in the URL and have that file returned as the contents of the page. However, the user shouldn't be able to specify just *any* path on our server, otherwise a malicious user could try to take advantage of

this: `localhost:8080/etc/shadow`. Here the attacker would be requesting the `/etc/shadow` file. To prevent these kinds of attacks, we should be able to tell the server to only allow the user to download certain files, or only files from certain directories (like `/var/www/my-website/public`).

Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).

To include the HTTP module, use the require() method:

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

Node.js as a Web Server

The HTTP module can create an HTTP server that listens to server ports and gives a response back to the client.

Use the createServer() method to create an HTTP server:

Example

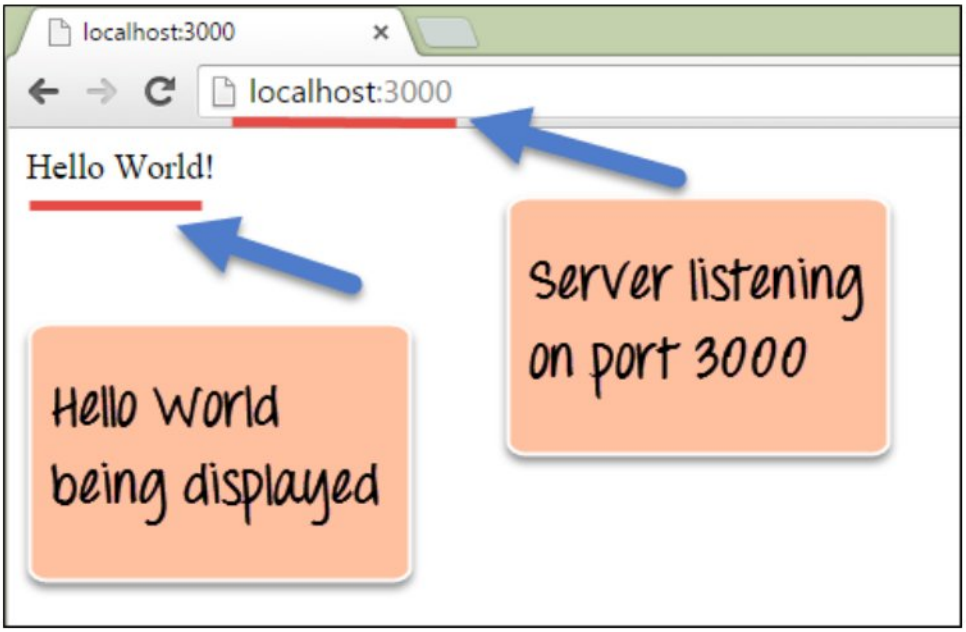
```
var http = require('http');

//create a server object:
http.createServer(function (req, res) {
  res.write('Hello World!'); //write a response to the client
  res.end(); //end the response
}).listen(8080); //the server object listens on port 8080
```

The function passed into the http.createServer() method, will be executed when someone tries to access the computer on port 8080.

Save the code above in a file called "demo_http.js", and initiate the file:

Initiate demo_http.js:

C:\Users*Your Name*>node demo_http.js



2.5 Introduction to the Express framework

What is Express?

Express is a small framework that sits on top of Node.js's web server functionality to simplify its APIs and add helpful new features. It makes it easier to organize your application's functionality with **middle ware and routing**; it adds helpful utilities to Node.js's HTTP objects;it facilitates the rendering of dynamic HTTP objects.

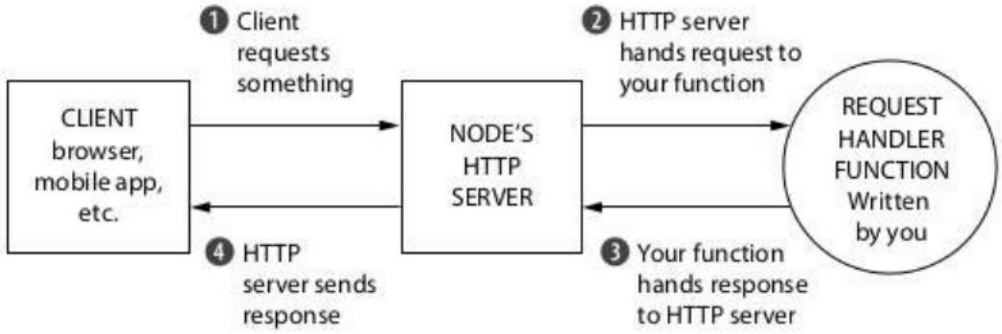
Express is a part of **MEAN** stack, a full stack JavaScript solution used in building fast, robust, and maintainable production web applications.

MongoDB(Database)
ExpressJS(Web Framework)
AngularJS(Front-end Framework)
NodeJS(Application Server)

Node.js is a JavaScript run time environment which is used to **create server-side applications and tools.** Node.js is fast, portable, and written in JavaScript but it does not directly support common tasks such as handling requests, serving files, and handling HTTP methods such as GET and POST. This is where Node.js's rich ecosystem comes to our aid.

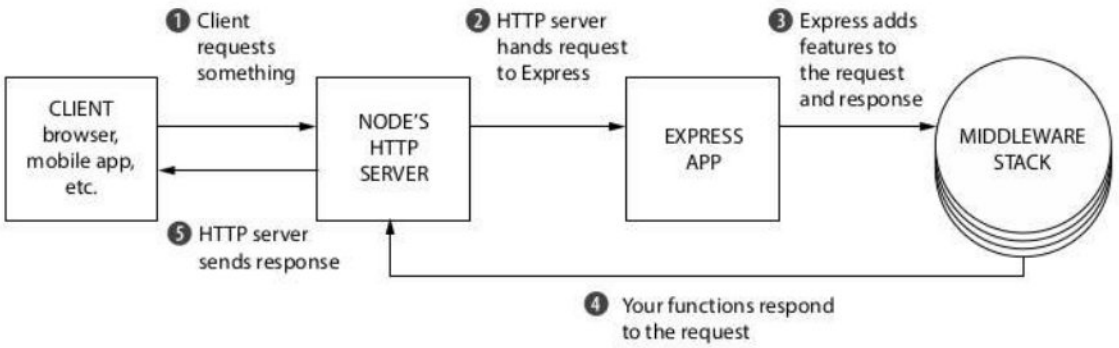
Express.js (Express) is a light web framework which sits on top of Node.js and it adds functionality like (middleware, routing, etc.) and simplicity to Node.js.

When creating a Node.js web application, we write a single JavaScript application which listens to requests from the browser, based on the request, the function will send back some data or an HTML web page.



A *request handler* is a JavaScript function which takes a request and sends an appropriate response.

Node.js APIs can get complex and writing how to handle a single request can end up being over 50 lines of code. Express makes it easier to write Node.js web applications.



Advantages of using Express with Node.js

- Express lets you take away a lot of the complexities of Node.js while adding helpful functions to a Node.js HTTP server.
- Instead of a large request handler function, Express allows us to handle requests by **writing many small modular and maintainable functions**.
- Express is *not opinionated*, meaning Express does not enforce any "right way" of doing things. **You can use any compatible middleware, and you can structure the app as you wish, making it flexible.**
- We can integrate with a template rendering engine (also called a view rendering engine in some articles) of our choice like Jade, Pug, EJS, etc.

A template engine enables you to use static template files and at runtime change the values of variables in those files.

- You can set up "middleware" for request processing.

Installing and using Express

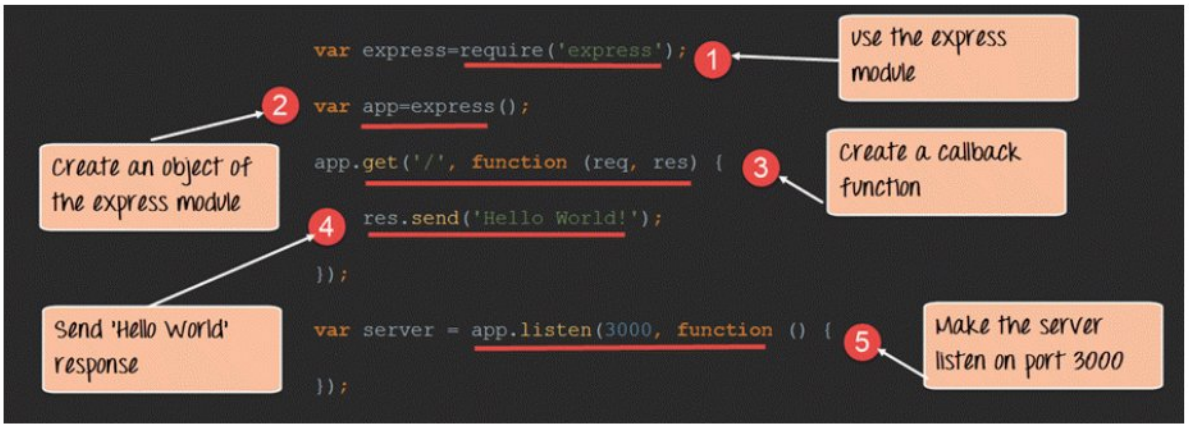
Express gets installed via the Node Package Manager. This can be done by executing the following line in the command line

npm install express

The above command requests the Node package manager to download the required express modules and install them accordingly.

Let's use our newly installed Express framework and create a simple "Hello World" application.

Our application is going to create a simple server module which will listen on port number 3000. In our example, if a request is made through the browser on this port number, then server application will send a 'Hello' World' response to the client.



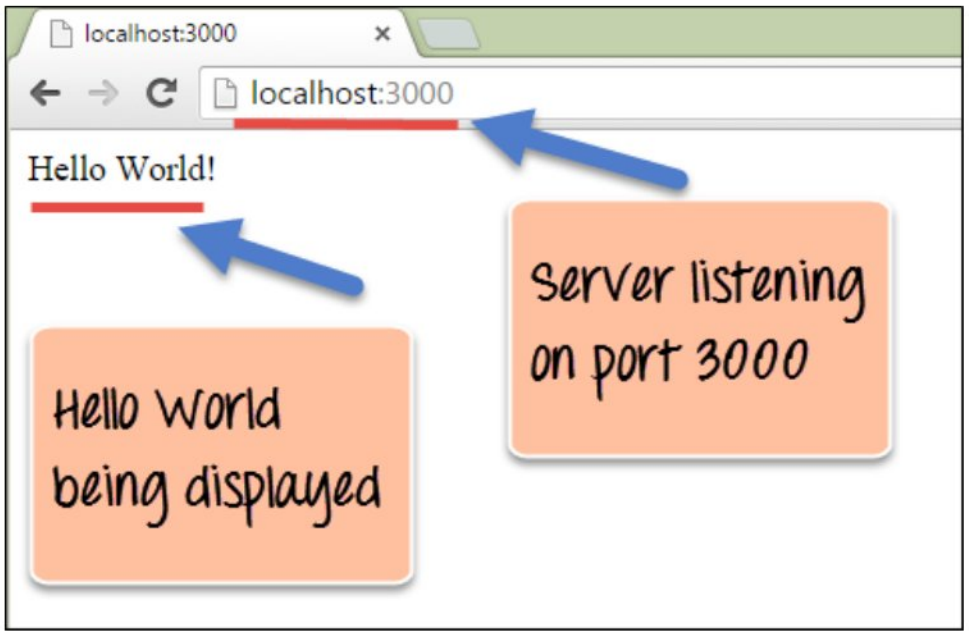

```
var express=require('express');
var app=express();
app.get('/',function(req,res)
{
res.send('Hello World!');
});
var server=app.listen(3000,function() {});
```

Code Explanation:

1. In our first line of code, we are using the require function to include the "express module."
2. Before we can start using the express module, we need to make an object of it.
3. Here we are creating a callback function. This function will be called whenever anybody browses to the root of our web application which is **http://localhost:3000** . The callback function will be used to send the string 'Hello World' to the web page.
4. In the callback function, we are sending the string "Hello World" back to the client. The 'res' parameter is used to send content back to the web page. This 'res' parameter is something that is provided by the 'request' module to enable one to send content back to the web page.
5. We are then using the listen to function to make our server application listen to client requests on port no 3000. You can specify any available port over here.

If the command is executed successfully, the following Output will be shown when you run your code in the browser.

Output:



From the output,

- You can clearly see that we if browse to the URL of localhost on port 3000, you will see the string 'Hello World' displayed on the page.
- Because in our code we have mentioned specifically for the server to listen on port no 3000, we are able to view the output when browsing to this URL.

What are Routes?

Routing determine the way in which an application responds to a client request to a particular endpoint.

For example, a client can make a GET, POST, PUT or DELETE http request for various URL such as the ones shown below;

http://localhost:3000/Books
http://localhost:3000/Students
In the above example,

- If a GET request is made for the first URL, then the response should ideally be a list of books.
- If the GET request is made for the second URL, then the response should ideally be a list of Students.
- So based on the URL which is accessed, a different functionality on the webserver will be invoked, and accordingly, the response will be sent to the client. This is the concept of routing.

Each route can have one or more handler functions, which are executed when the route is matched.

The general syntax for a route is shown below

app.METHOD(PATH, HANDLER)

Wherein,

1) app is an instance of the express module

2) METHOD is an HTTP request method (GET, POST, PUT or DELETE)

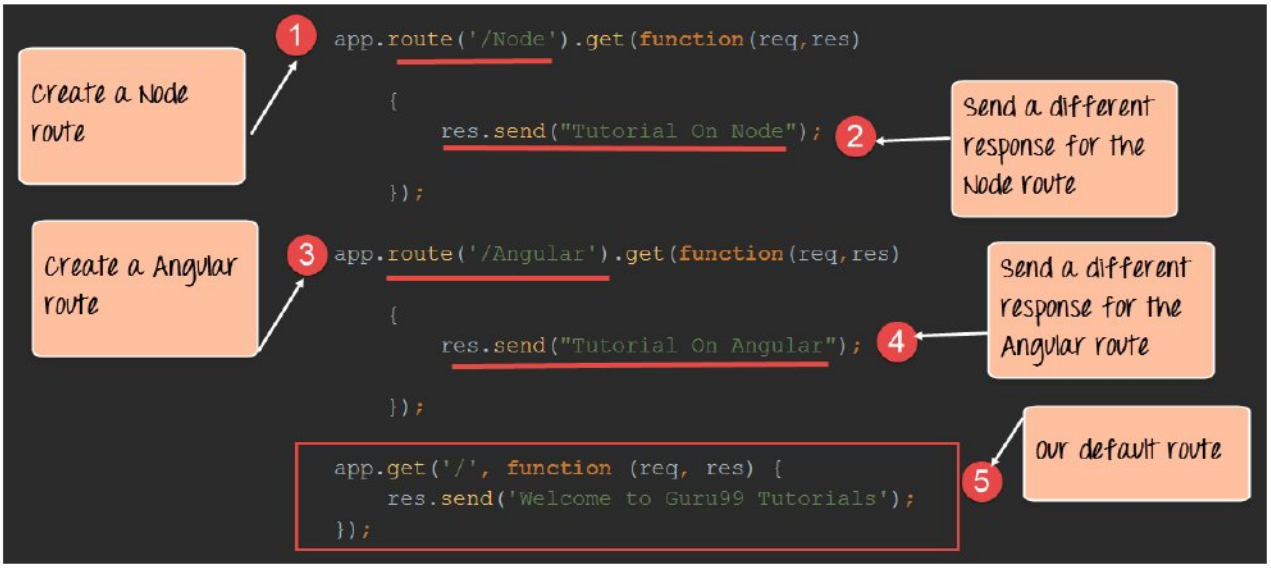
3) PATH is a path on the server.

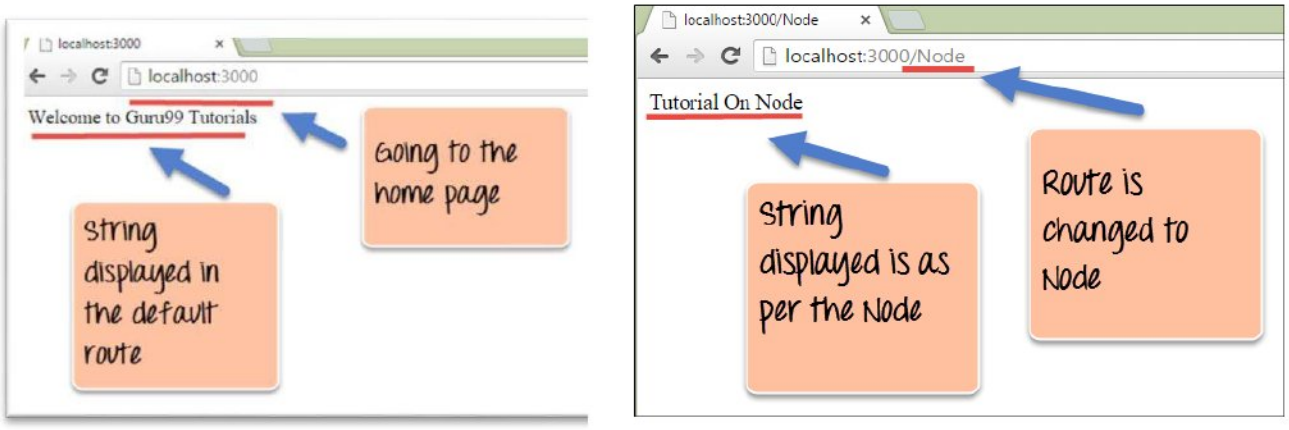
4) HANDLER is the function executed when the route is matched.

Let's look at an example of how we can implement routes in the express. Our example will create 3 routes as

1. A /Node route which will display the string "Tutorial on Node" if this route is accessed
2. A /Angular route which will display the string "Tutorial on Angular" if this route is accessed
3. A default route / which will display the string "Welcome to Guru99 Tutorials."

Our basic code will remain the same as previous examples. The below snippet is an add-on to showcase how routing is implemented.





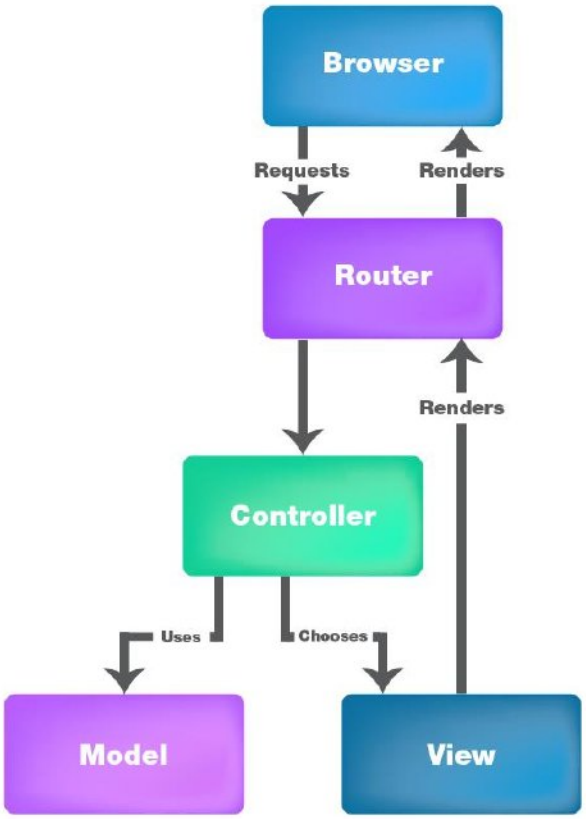
2.6 Server-side rendering with Templating Engines

Regular Web Applications rely on *templates* to dynamically generate HTML pages on the server. These templates are text files that contain static content as well as a special syntax describing how the data should be inserted dynamically. During an HTTP request, the application loads and renders the template using the given contextual data and sends back the page to the client.

This technique is known as *Server-Side Rendering (or SSR).*

Creating a Dynamic Page

Unlike other server frameworks, choosing Node does not implicitly choose the templating engine you'll use for creating pages. There existed several templating engines for JavaScript when Node was created, and that number has only grown since. Thanks to Node, we now have a large number of server-side-only engines as well. Almost every JavaScript library of sufficient size offers a template engine, Underscore probably being the smallest, and there are many standalone options. Node frameworks also tend to have a default. Express uses Jade out of the box, for instance. It doesn't really matter which you choose, as long as it meets your needs and is comfortable for you.



The Router

The router is commonly a front controller, taking in an HTTP request and converting it into the correct controller method to call. It also binds the request's information into the request's input model. This is where a lot of cross-cutting middleware goes, such as authentication strategies and logging. This is also where non-controller resources, such as CSS and static HTML pages, are handled.

The Controller

This is the core unit of work for a web request. The controller's main goal is to map one or more URLs to an executable model. It takes an input model representing the HTTP request and URL, and then it either changes state if it's a POST request or it returns data if it's a GET request. Ideally, these are fairly skinny, delegating most work to the model. There are usually pre- and post-request hooks that you can wire up for cross-cutting concerns, such as authorization and setting up database transactions.

The Model

The model is the meat and potatoes of the system. The goal here is to do the real work. It's as simple as that.

The View

This is the page that the user will see. It renders the model as a browser-readable page, usually HTML. It's often rendered via a template engine with its own syntax, such as Razor.

app.js

```
// Requiring modules
const express = require('express');
const app = express();
const ejs = require('ejs');
var fs = require('fs');
const port = 9910;
// Render index.ejs file
app.get('/', function (req, res) {

    // Render page using renderFile method
    ejs.renderFile('index.ejs', {},
        {}, function (err, template) {
        if (err) {
            throw err;
        } else {
            res.end(template);
        }
    });
});

// Server setup
app.listen(port, function (error) {
    if (error)
        throw error;
    else
```

```
});
```

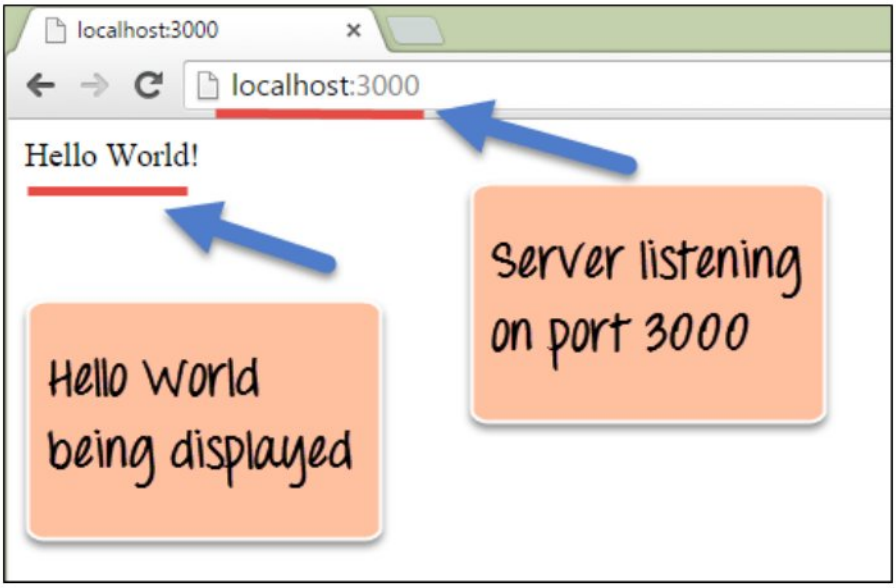
index.ejs

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content=
        "width=device-width, initial-scale=1.0">
</head>
<body>
    <h1>Hello World</h1>
</body>
</html>
```

To Open The Terminal→Type Node Filename.Js→Display The Message From **Server Is Running**

To Open Chrome Browser→ Type Http:/Localhost:Port Number(8080)→Enter→Display The Msg From **Hello World**

Output:



2.7 Static Files

Static files are files that clients download as they are from the server. Create a new directory, **public**. Express, by default does not allow you to serve static files. You need to enable it using the following built-in middleware.

app.use(express.static('public'));

Install Express as a dependency:

1. **npm install express --save**

Within your package.json, update your start script to include node and your index.js file.

package.json

```
{
  "name": "express-static-file-tutorial",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "keywords": [],
  "author": "Paul Halliday",
  "license": "MIT"
}
```

This will allow you to use the npm start command in your terminal to launch your Express server.

Now that your files are set up let's begin your Express server.

In your index.js file, require in an Express instance and implement a GET request:

index.js

```
const express = require('express');
const app = express();
const PORT = 3000;
```

```
  res.send('Hello World!');
});
```

app.listen(PORT, () => console.log(`Server listening on port: ${PORT}`));

Now let's tell Express to handle your static files.

Express provides a built-in method to serve your static files:

app.use(express.static('public'));

When you call app.use(), you're telling Express to use a piece of middleware. *Middleware* is a function that Express passes requests through before sending them to your routing functions, such as your app.get('/') route. express.static() finds and returns the static files requested. The argument you pass into express.static() is the name of the directory you want Express to serve files. Here, the public directory.

In index.js, serve your static files below your PORT variable. Pass in your public directory as the argument:

index.js

```
const express = require('express');
const app = express();
const PORT = 3000;

app.use(express.static('public'));

app.get('/', (req, res) => {
  res.send('Hello World!');
});
```

app.listen(PORT, () => console.log(`Server listening on port: ${PORT}`));

With your Express server set, let's focus on the client-side.

Navigate to your index.html file in the public directory. Populate the file with body and image elements:

[label index.html]

```
<html>
 <head>
  <title>Hello World!</title>
 </head>
```

```
    <h1>Hello, World!</h1>
    <img src="shark.png" alt="shark">
  </body>
</html>
```

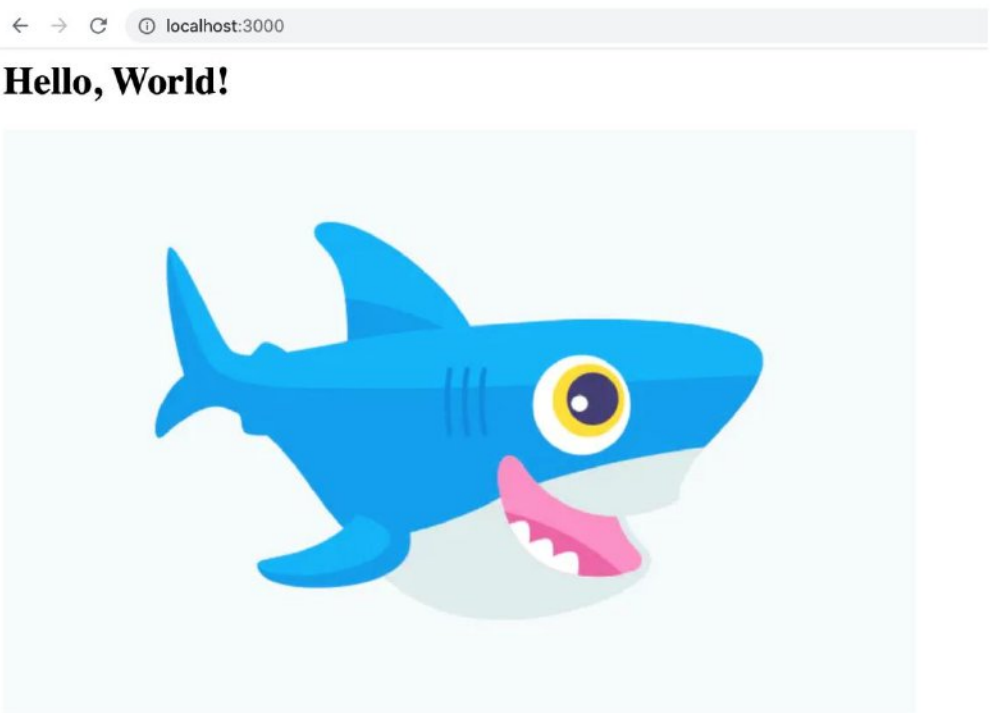
Notice the image element source to shark.png. Since you've served the public directory through Express, you can add the file name as your image source's value.

In your terminal, launch your Express project:

1. npm start

Server listening on port: 3000

Open your web browser, and navigate to http://localhost:3000. You will see your project:



2.8 async/await

Async/Await Function in JavaScript

We all know that JavaScript is Synchronous in nature which means that it has an event loop that allows you to queue up an action that won't take place until the loop is available sometime after the code that queued the action has finished executing.

But there's a lot of functionalities in our program which makes our code Asynchronous and one of them is the Async/Await functionality. Async/Await is the extension of promises which we get as a support in the language.

Following sections will describe more about async and await in detail along with some examples (individual as well as combined examples of async-await):

Async: It simply allows us to write promises based code as if it was synchronous and it checks that we are not breaking the execution thread. It operates asynchronously via the event-loop. Async functions will always return a value. It makes sure that a promise is returned and if it is not returned then JavaScript automatically wraps it in a promise which is resolved with its value.

Example-1:

```
const getData = async() => {

   var data = "Hello World";

   return data;

}

getData().then(data => console.log(data));
```

Output:
Hello World

Await: Await function is used to wait for the promise. It could be used within the async block only. It makes the code wait until the promise returns a result. It only makes the async block wait.

Example-2:

```
const getData = async() => {

   var y = await "Hello World";

   console.log(y);

} console.log(1);

getData();

console.log(2);
```

Output:
1

2

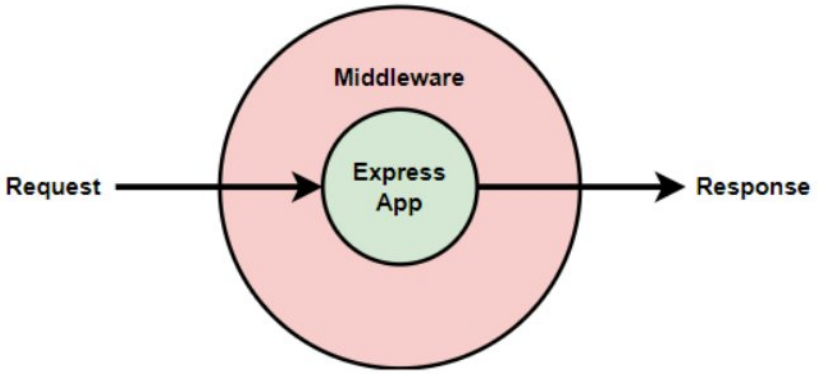
Hello World

Notice that the console prints 2 before the **"Hello World"**. This is due to the usage of the await keyword.

2.9 Fetching JSON from Express

Express.js res.json() Function

The **res.json()** function sends a JSON response. This method sends a response (with the correct content-type) that is the parameter converted to a JSON string using the JSON.stringify() method.



Middleware in Express.js applications

Syntax:

res.json([body])

Parameters: The body parameter is the body which is to be sent in the response.
Return Value: It returns an Object.
Installation of express module:

1. You can visit the link to Install express module. You can install this package by using this command.

npm install express

2. After installing the express module, you can check your express version in command prompt using the command.

npm version express

3. After that, you can just create a folder and add a file for example, index.js. To run this file you need to run the following command.

Filename: index.js (With middleware)

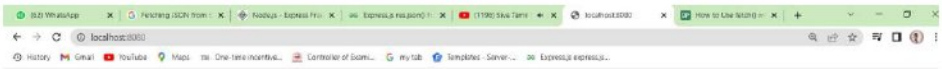
```
var express = require('express');

var app = express();

var PORT = 3000;

// With middleware

app.use('/', function(req, res, next){

   res.json({title: " Welcome To Adhiparasakthi Engineering College "})

   next();

})

app.get('/', function(req, res){

    console.log("Welcome Page")

    res.end();

});

app.listen(PORT, function(err){

   if (err) console.log(err);

   console.log("Server listening on PORT", PORT);

});
```

Run index.js file using below command:

node index.js

Now open a browser and go to *http://localhost:3000/*, now on your screen you will see the following output:

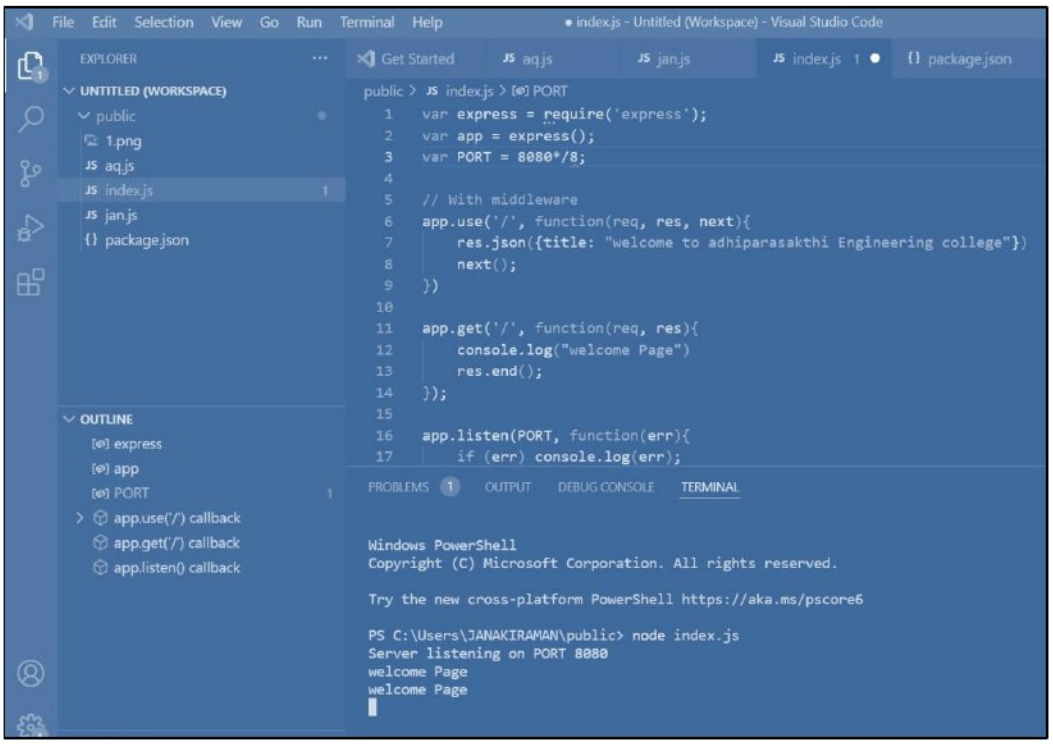
{"title":"Welcome To Adhiparasakthi Engineering College"}



```
{"title":"welcome to adhiparasakthi
Engineering college"}
```

And you will see the following output on your console:

Welcome Page



UNIT IV ADVANCED CLIENT SIDE PROGRAMMING 9

React JS: ReactDOM - JSX - Components - Properties – Fetch API - State and Lifecycle - JS Localstorage - Events - Lifting State Up - Composition and Inheritance

I.ReactJS

React is a declarative, efficient, and flexible JavaScript library for developing user interface (UI) in web application. It's 'V' in MVC. ReactJS is an open-source, component-based front-end library responsible only for the view layer of the application. React is developed and released by Facebook. Facebook is continuously working on the React library and enhancing it by fixing bugs and introducing new features.

It can be used to develop small applications as well as big, complex applications. ReactJS provides minimal and solid feature set to kick-start a web application. React community compliments React library by providing large set of ready-made components to develop web application in a record time. React community also provides advanced concept like state management, routing, etc., on top of the React library.

React versions

The initial version, 0.3.0 of React is released on May, 2013 and the latest version, *17.0.1* is released on October, 2020. The major version introduces breaking changes and the minor version introduces new feature without breaking the existing functionality. Bug fixes are released as and when necessary. React follows the *Sematic Versioning (semver)* principle.

Features

The salient features of *React library* are as follows −

- Solid base architecture
- Extensible architecture
- Component based library
- JSX based design architecture
- Declarative UI library

Features of React.js: There are unique features are available on React because that it is widely popular.

Use JSX: It is faster than normal JavaScript as it performs optimizations while translating to regular JavaScript. It makes it easier for us to create templates.

Virtual DOM: Virtual DOM exists which is like a lightweight copy of the actual DOM. So for every object that exists in the original DOM, there is an object for that in React Virtual DOM. It is exactly the same, but it does not have the power to directly change the layout of the document. Manipulating DOM is slow, but manipulating Virtual DOM is fast as nothing gets drawn on the screen.

One-way Data Binding: This feature gives you better control over your application.

Component: A Component is one of the core building blocks of React. In other words, we can say that every application you will develop in React will be made up of pieces called components. Components make the task of building UIs much easier. A UI is broken down into multiple individual pieces called components and work on them independently and merge them all in a parent component which will be your final UI.

Performance: React.js use JSX, which is faster compared to normal JavaScript and HTML. Virtual DOM is a less time taking procedure to update webpages content.

Simple application

A React application is made of multiple components, each responsible for rendering a small, reusable piece of HTML. Components can be nested within other components to allow complex applications to be built out of simple building blocks.

Note: React is not a framework. It is just a library developed by Facebook to solve some problems that we were facing earlier.

Prerequisites: Download Node packages with their latest version.

Example: Create a new React project by using the command below:

npx create-react-app myapp

Filename App.js: Now change the **App.js** file with the given below code:

Javascript

```
import React,{ Component } from 'react';

class App extends Component {
```

```
render() {
        return (
        <div>
                <h1>Hello, Learner.Welcome to GeeksforGeeks.</h1>
        </div>
        );
}
}
export default App;
```

How does it work: While building client-side apps, a team of Facebook developers realized that the DOM is slow (The Document Object Model (DOM) is an application programming interface (API) for HTML and XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated.). So, to make it faster, React implements a virtual DOM that is basically a DOM tree representation in JavaScript. So when it needs to read or write to the DOM, it will use the virtual representation of it. Then the virtual DOM will try to find the most efficient way to update the browser's DOM.

Unlike browser DOM elements, React elements are plain objects and are cheap to create. React DOM takes care of updating the DOM to match the React elements. The reason for this is that JavaScript is very fast and it's worth keeping a DOM tree in it to speed up its manipulation. Although React was conceived to be used in the browser, because of its design it can also be used in the server with Node.js.

Benefits

Few benefits of using *React library* are as follows −

- Easy to learn
- Easy to adept in modern as well as legacy application
- Faster way to code a functionality
- Availability of large number of ready-made component
- Large and active community

Applications

- *Facebook*, popular social media application
- *Instagram*, popular photo sharing application
- *Netflix*, popular media streaming application
- *Code Academy*, popular online training application
- *Reddit*, popular content sharing application

ReactJS – Installation

React provides CLI tools for the developer to fast forward the creation, development and deployment of the React based web application. React CLI tools depends on the Node.js and must be installed in your system. Hopefully, you have installed Node.js on your machine. We can check it using the below command −

node --version

You could see the version of Nodejs you might have installed. It is shown as below for me,

v14.2.0

If Nodejs is not installed, you can download and install by visiting https://nodejs.org/en/download/.

II.Introduction to JSX

React is a declarative, efficient, and flexible JavaScript library for building user interfaces. But instead of using regular JavaScript, React code should be written in something called JSX.

Let us see a sample JSX code:

constele = <h1>This is sample JSX</h1>;

The above code snippet somewhat looks like HTML and it also uses a JavaScript-like variable but is neither HTML nor JavaScript, it is JSX. JSX is basically a syntax extension of regular JavaScript and is used to create React elements. These elements are then rendered to the React DOM.

Why JSX?

- It is faster than normal JavaScript as it performs optimizations while translating to regular JavaScript.

- It makes it easier for us to create templates.

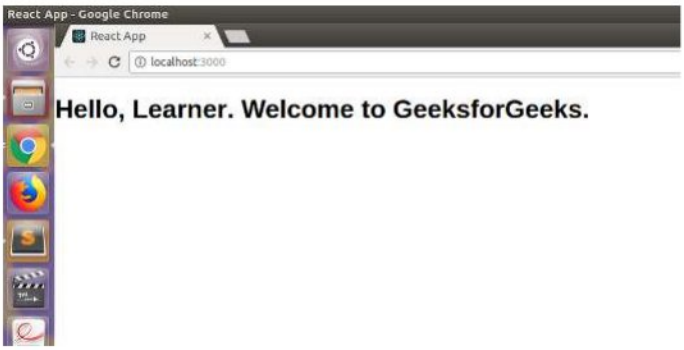
- Instead of separating the markup and logic in separated files, React uses *components* for this purpose. We will learn about components in detail in further articles.

Using JavaScript expressions in JSX: In React we are allowed to use normal JavaScript expressions with JSX. To embed any JavaScript expression in a piece of code written in JSX we will have to wrap that expression in curly braces {}. Consider the below program, written in the index.js file:

Javascript

```
import React from 'react';
import ReactDOM from 'react-dom';
const name = "Learner";
const element = <h1>Hello,
{ name }.Welcome to GeeksforGeeks.< /h1>;
ReactDOM.render(
        element,
document.getElementById("root")
);
```

Output:

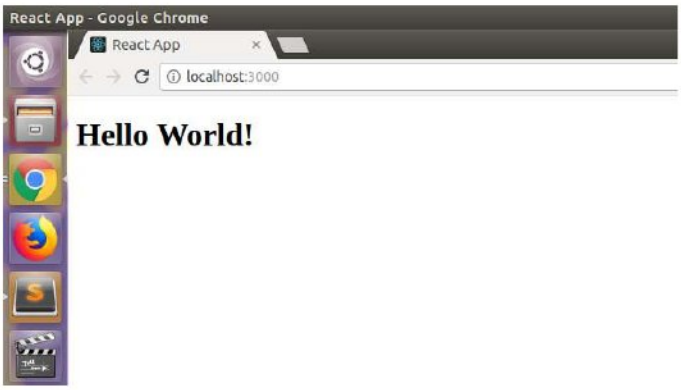


In the above program we have embedded the javascript expression *const name = "Learner";* in our JSX code. We embed the use of any JavaScript expression in JSX by wrapping them in curly braces except if-else statements. But we can use conditional statements instead of if-else statements in JSX. Below is the example where conditional expressing is embedded in JSX:

javascript

```
import React from 'react';
import ReactDOM from 'react-dom';
let i = 1;
const element = <h1>{ (i == 1) ? 'Hello World!' : 'False!' } < /h1>;
ReactDOM.render(
        element,
        document.getElementById("root")
);
```

Output:



In the above example, the variable i is checked if for the value 1. As it equals 1 so the string 'Hello World!' is returned to the JSX code. If we modify the value of the variable i then the string 'False' will be returned.

Attributes in JSX: JSX allows us to use attributes with the HTML elements just like we do with normal HTML. But instead of the normal naming convention of HTML, JSX uses camelcase convention for attributes. For example, *class* in HTML becomes *className* in JSX. The main reason behind this is that some attribute names in HTML like 'class' are reserved keywords in JavaScripts. So, in order to avoid this problem, JSX uses the camel case naming convention for attributes. We can also use custom attributes in JSX. For custom attributes, the names of such attributes should be prefixed by **data-**. In the below example, we have used a custom attribute with name **data-sampleAttribute** for the <h2> tag.

Javascript

```
import React from 'react';
```

```
import ReactDOM from 'react-dom';
 const element = <div><h1 className = "hello">Hello Geek</h1>
        <h2 data-sampleAttribute="sample">Custom attribute</h2>< /div>;
 ReactDOM.render(
element,
   document.getElementById("root")
);
```

Specifying attribute values: JSX allows us to specify attribute values in two ways:

1. **As for string literals:** We can specify the values of attributes as hard-coded strings using quotes: constele = <h1 className = "firstAttribute">Hello!</h1>;

1. **As expressions:** We can specify attributes as expressions using curly braces {}:

```
constele = <h1 className = {varName}>Hello!</h1>;
```

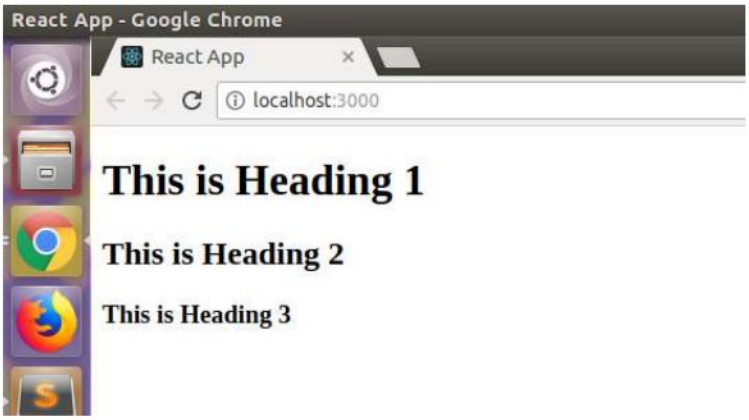
Wrapping elements or Children in JSX: Consider a situation where you want to render multiple tags at a time. To do this we need to wrap all of this tag under a parent tag and then render this parent element to the HTML. All the subtags are called child tags or children of this parent element.

Notice in the below example how we have wrapped h1, h2, and h3 tags under a single div element and rendered them to HTML:

Javascript

```
import React from 'react';
import ReactDOM from 'react-dom';
 const element = <div>
            <h1>This is Heading 1 < /h1>
            <h2>This is Heading 2</h2 >
            <h3>This is Heading 3 < /h3>
          </div > ;
ReactDOM.render(
   element,
```

Output:



Comments in JSX: JSX allows us to use comments as it allows us to use JavaScript expressions. Comments in JSX begins with /* and ends with */. We can add comments in JSX by wrapping them in curly braces {} just like we did in the case of expressions. Below example shows how to add comments in JSX:

javascript

```
import React from 'react';
import ReactDOM from 'react-dom';
 const element = <div><h1>Hello World !</h1>
          {/ * This is a comment in JSX * /}
       </div>;
 ReactDOM.render(
   element,
   document.getElementById("root"));
```

III. ReactDOM

We can use JSX to store HTML markups within Javascript variables. Now, ReactJS is a library to build active User Interfaces thus rendering is one of the integral parts of ReactJS. React provides the developers with a package **react-dom**. ReactDOM to access and modify the DOM

What is DOM?

DOM, abbreviated as Document Object Model, is a World Wide Web Consortium standard logical representation of any webpage. In easier words, DOM is a tree-like structure that contains all the elements and it's properties of a website as its nodes. DOM provides a language-neutral interface that allows accessing and updating of the content of any element of a webpage.

Before React, Developers directly manipulated the DOM elements which resulted in frequent DOM manipulation, and each time an update was made the browser had to recalculate and repaint the whole view according to the particular CSS of the page, which made the total process to consume a lot of time. As a betterment, React brought into the scene the virtual DOM.
The **Virtual DOM** can be referred to as a copy of the actual DOM representation that is used to hold the updates made by the user and finally reflect it over to the original Browser DOM at once consuming much lesser time.

What is ReactDOM?

ReactDOM is a package that provides DOM specific methods that can be used at the top level of a web app to enable an efficient way of managing DOM elements of the web page. ReactDOM provides the developers with an API containing the following methods and a few more.

i. render()
ii. findDOMNode()
iii. unmountComponentAtNode()
iv. hydrate()
v. createPortal()

Pre-requisite: To use the ReactDOM in any React web app we must first import ReactDOM from the react-dom package by using the following code snippet:

```
import ReactDOM from 'react-dom'
```

i)render() Function

This is one of the most important methods of ReactDOM. This function is used to render a single React Component or several Components wrapped together in a Component or a div element.

This function uses the efficient methods of React for updating the DOM by being able to change only a subtree, efficient diff methods, etc.

Syntax:

ReactDOM.render(element, container, callback)

Parameters: This method can take a maximum of three parameters as described below.

- **element:** This parameter expects a JSX expression or a React Element to be rendered.
- **container:** This parameter expects the container in which the element has to be rendered.
- **callback:** This is an optional parameter that expects a function that is to be executed once the render is complete.
- **Return Type:** This function returns a reference to the component or null if a stateless component was rendered.

ii)findDOMNode() Function

This function is generally used to get the DOM node where a particular React component was rendered. This method is very less used like the following can be done by adding a ref attribute to each component itself.

Syntax:

ReactDOM.findDOMNode(component)

Parameters: This method takes a single parameter component that expects a React Component to be searched in the Browser DOM.

Return Type: This function returns the DOM node where the component was rendered on success otherwise null.

iii)unmountComponentAtNode() Function

This function is used to unmount or remove the React Component that was rendered to a particular container. As an example, you may think of a notification component, after a brief amount of time it is better to remove the component making the web page more efficient.

Syntax:

ReactDOM.unmountComponentAtNode(container)

Parameters: This method takes a single parameter container which expects the DOM container from which the React component has to be removed.

Return Type: This function returns true on success otherwise false.

iv)hydrate() Function

This method is equivalent to the render() method but is implemented while using server-side rendering.

Syntax:

ReactDOM.hydrate(element, container, callback)

Parameters: This method can take a maximum of three parameters as described below.

- **element:** This parameter expects a JSX expression or a React Component to be rendered.
- **container:** This parameter expects the container in which the element has to be rendered.
- **callback:** This is an optional parameter that expects a function that is to be executed once the render is complete.
- **Return Type:** This function attempts to attach event listeners to the existing markup and returns a reference to the component or null if a stateless component was rendered.

v)createPortal() Function

Usually, when an element is returned from a component's render method, it's mounted on the DOM as a child of the nearest parent node which in some cases may not be desired. Portals allow us to render a component into a DOM node that resides outside the current DOM hierarchy of

Syntax: ***ReactDOM.createPortal(child, container)***

Parameters: This method takes two parameters as described below.

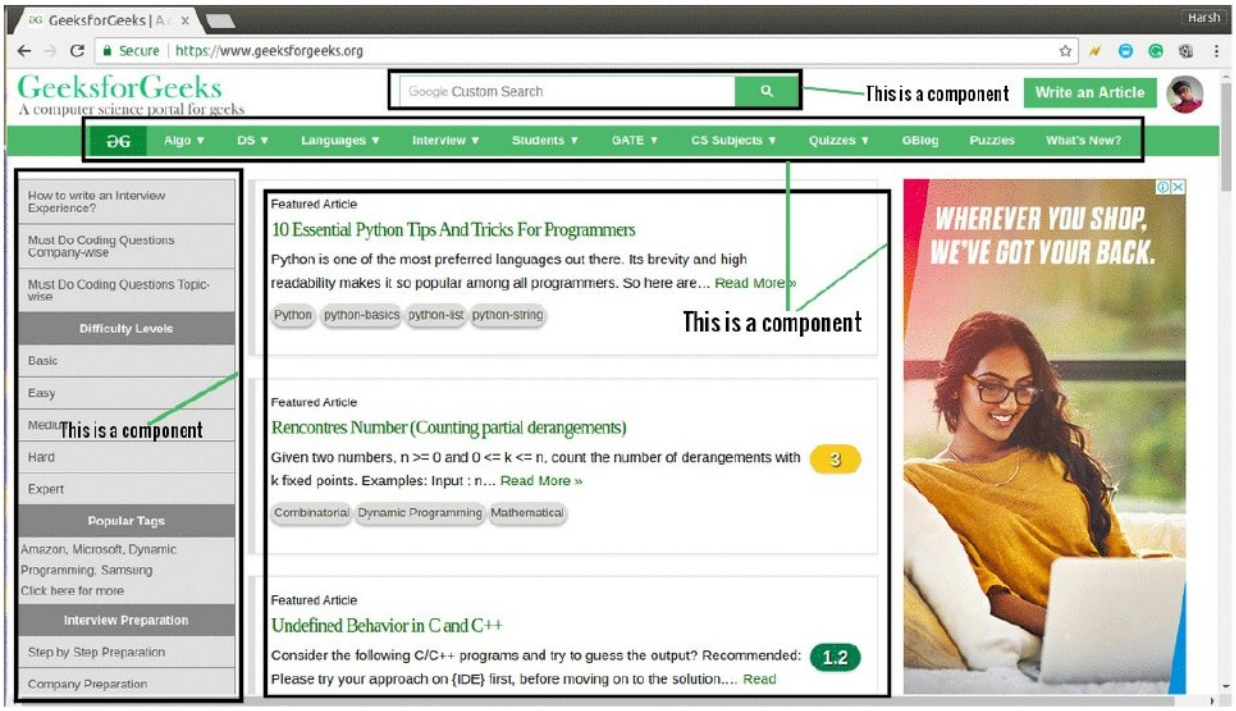
- **child:** This parameter expects a JSX expression or a React Component to be rendered.
- **container:** This parameter expects the container in which the element has to be rendered.
- **Return Type:** This function returns nothing.

Important Points to Note:

- **ReactDOM.render()** replaces the child of the given container if any. It uses a highly efficient diff algorithm and can modify any subtree of the DOM.
- **findDOMNode()** function can only be implemented upon mounted components thus Functional components cannot be used in findDOMNode() method.
- **ReactDOM** uses observables thus provides an efficient way of DOM handling.
- **ReactDOM**can be used on both the client-side and server-side.

IV. ReactJS Components

A **Component** is one of the core building blocks of React.Components make the task of building UIs much easier. You can see a UI broken down into multiple individual pieces called components and work on them independently and merge them all in a parent component which will be your final UI.



Google's custom search page can be seen as an individual component, the navigation bar can be seen as an individual component, the sidebar is an individual component, the list of articles or post is also an individual component and finally, we can merge all of these individual components to make a parent component which will be the final UI for the homepage. Components in React basically return a piece of JSX code that tells what should be rendered on the screen. In React, we mainly have two types of components:

1. **Functional Components**: Functional components are simply javascript functions. We can create a functional component in React by writing a javascript function. These functions may or may not receive data as parameters. Below example shows a valid functional component in React:

```
constDemocomponent=()=>
{
    return <h1>Welcome Message!</h1>;
}
```

2. **Class Components**: The class components are a little more complex than the functional components. The functional components are not aware of the other components in your

program whereas the class components can work with each other. We can pass data from one class component to other class components. We can use JavaScript ES6 classes to create class-based components in React. Below example shows a valid class-based component in React:

```
class Democomponent extends React.Component
{
render(){
        return <h1>Welcome Message!</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 class component. We will use functional component only when we are sure that our component does not require interacting or work with any other component. That is, these components do not require data from other components however we can compose multiple functional components under a single functional component. We can also use class-based components for this purpose but it is not recommended as using class-based components without need will make your application in-efficient.

3.Rendering Components

We know how elements initialized with DOM tags are rendered using ReactDOM.render() method. React is also 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.

Below syntax shows how to initialize a component to an element:

constelementName = <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 with html tags.

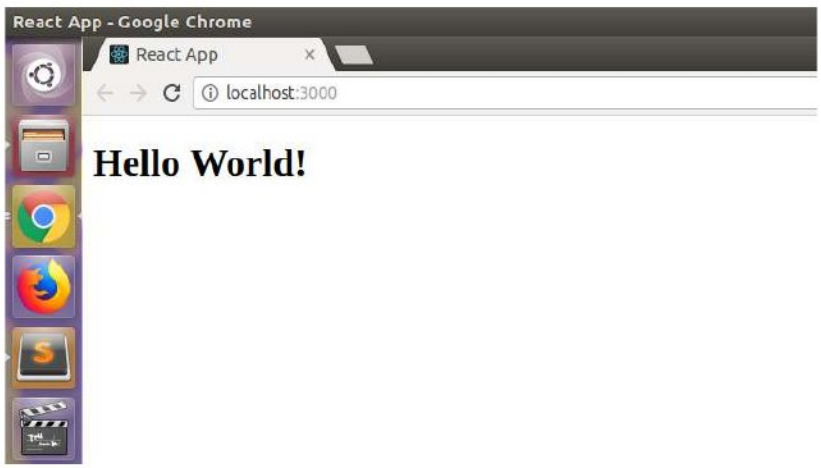
Below example renders a component named Welcome to the screen:

Open your **index.js** file from your project directory, and make the given below changes:

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:



Let us see step-wise what is happening in the above example:

1. We call the ReactDOM.render() as the first parameter.
2. React then calls the component Welcome, which returns <h1>Hello World!</h1>; as the result.
3. 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”.
4.

4.Composing Components: “we can merge all of these individual components to make a parent component”. This is what we call composing components. We will now create individual components named Navbar, Sidebar, ArticleList and merge them to create a parent component named App and then render this App component.

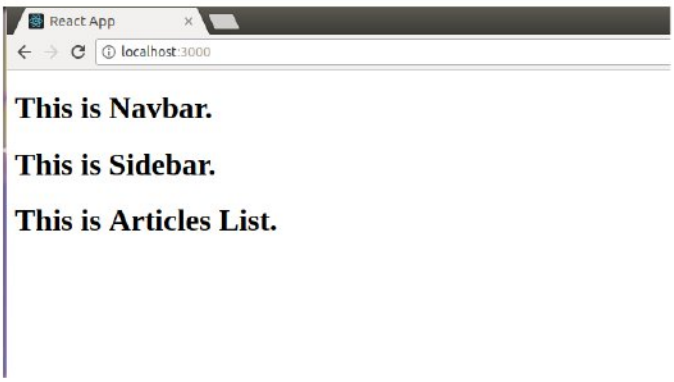
The below code in the index.js file explains how to do this:

Filename- App.js:

```
import React from 'react';
import ReactDOM from 'react-dom';

// Navbar Component
constNavbar=()=>
{
        return <h1>This is Navbar.< /h1>
}

// Sidebar Component
const Sidebar=()=> {
        return <h1>This is Sidebar.</h1>
}

// Article list Component
constArticleList=()=>
{
        return <h1>This is Articles List.</h1>
```

```
}

// App Component
const App=()=>
{
        return(
                        <div>
                                <Navbar />
                                <Sidebar />
                                <ArticleList />
                        </div>
                );
}
ReactDOM.render(
        <App />,
        document.getElementById("root")
);
```

Output:



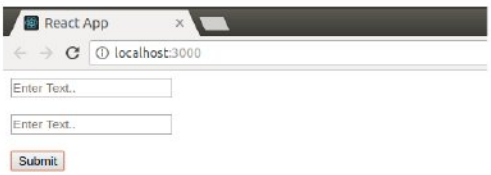
5.Decomposing Components: Decomposing a Component means breaking down the component into smaller components. we want to make a component for an HTML form. Let's say our form will have two input fields and a submit button. We can create a form component as shown below:

Filename- App.js:

```
import React from 'react';
```

```
import ReactDOM from 'react-dom';

const Form=()=>
{
	return (
		<div>
			<input type = "text" placeholder = "Enter Text.." />
			<br />
			<br />
			<input type = "text" placeholder = "Enter Text.." />
			<br />
			<br />
			<button type = "submit">Submit</button>
		</div>
	);
}
ReactDOM.render(
	<Form />,
	document.getElementById("root")
);
```

Output:



The above code works well to create a form. But let us say now we need some other form with three input fields. To do this we will have to again write the complete code with three input fields now. But what if we have broken down the Form component into two smaller components, one for the input field and another one for the button? This could have increased our code reusability

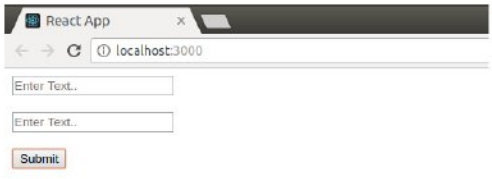
smallest possible units and then merge them to create a parent component to increase the code modularity and reusability. In the below code the component Form is broken down into smaller components Input and Button.

Filename- App.js:

```
import React from 'react';
import ReactDOM from 'react-dom';

// Input field component
const Input=()=>
{
	return(
		<div>
			<input type="text" placeholder="Enter Text.." />
			<br />
			<br />
		</div>
	);
}

// Button Component
const Button=()=>
{
	return <button type = "submit">Submit</button>;
}

// Form component
const Form=()=>
{
	return (
		<div>
```

```
                    <Input />
                    <Input />
                    <Button />
                </div>
        );
}
ReactDOM.render(
        <Form />,
        document.getElementById("root")
);
```

Output:



V. React Props

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 **main.js** file of your ReactJS project and used it inside the component in which you need. It can be explained in the below example.

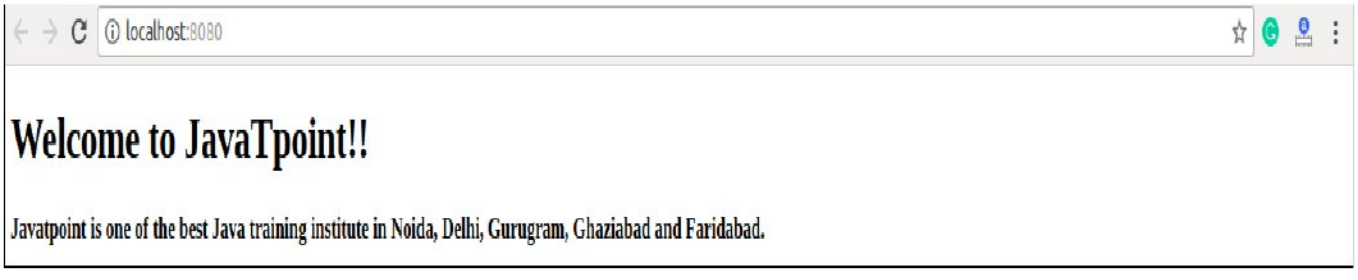
Example: App.js

```
1.  import React, { Component } from 'react';
2.  class App extends React.Component {
3.    render() {
4.      return (
5.        <div>
6.          <h1> Welcome to { this.props.name } </h1>
7.          <p> <h4> Javatpoint is one of the best Java training institute in Noida, Delhi, Gur
    ugram, Ghaziabad and Faridabad. </h4> </p>
8.        </div>
9.      );
10.   }
11. }
12. export default App;
```

Main.js

```
1.  import React from 'react';
2.  import ReactDOM from 'react-dom';
3.  import App from './App.js';
4.  ReactDOM.render(<App name = "JavaTpoint!!" />, document.getElementById('app'));
```

Output



VI. React State

The state is an updatable structure that is used to contain data or information about the component. The state in a component can change over time. The change in state over time can happen as a response to user action or system event. A component with the state is known as stateful components. It is the heart of the react component which determines the behavior of the component and how it will render. They are also responsible for making a component dynamic and interactive.

A state must be kept as simple as possible. It can be set by using the **setState()** method and calling setState() method triggers UI updates. A state represents the component's local state or information. It can only be accessed or modified inside the component or by the component directly. To set an initial state before any interaction occurs, we need to use the **getInitialState()** method.

For example, if we have five components that need data or information from the state, then we need to create one container component that will keep the state for all of them.

VII. Defining State

To define a state, you have to first declare a default set of values for defining the component's initial state. To do this, add a class constructor which assigns an initial state using this.state. The '**this.**state' property can be rendered inside **render()** method.

Example

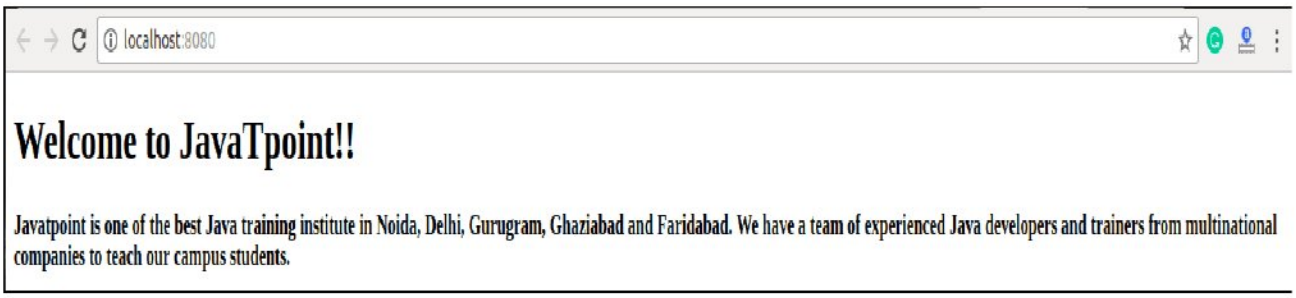
The below sample code shows how we can create a stateful component using ES6 syntax.

```
1.  import React, { Component } from 'react';
2.  class App extends React.Component {
3.   constructor() {
4.      super();
5.      this.state = { displayBio: true };
6.      }
7.      render() {
```

```
9.                  <div>
10.                   <p><h3>Javatpoint is one of the best Java training institute in Noida, Delhi, G
    urugram, Ghaziabad and Faridabad. We have a team of experienced Java developers and trainers
    from multinational companies to teach our campus students.</h3></p>
11.                 </div>
12.                  ) : null;
13.                  return (
14.                    <div>
15.                      <h1> Welcome to JavaTpoint!! </h1>
16.                      { bio }
17.                    </div>
18.                  );
19.              }
20.          }
21.          export default App;
```

To set the state, it is required to call the super() method in the constructor. It is because this.state is uninitialized before the super() method has been called.

Output



VIII. Changing the State

We can change the component state by using the setState() method and passing a new state object as the argument. Now, create a new method toggleDisplayBio() in the above example and bind this keyword to the toggleDisplayBio() method otherwise we can't access this inside toggleDisplayBio() method.

```
1.          this.toggleDisplayBio = this.toggleDisplayBio.bind(this);
```

Example

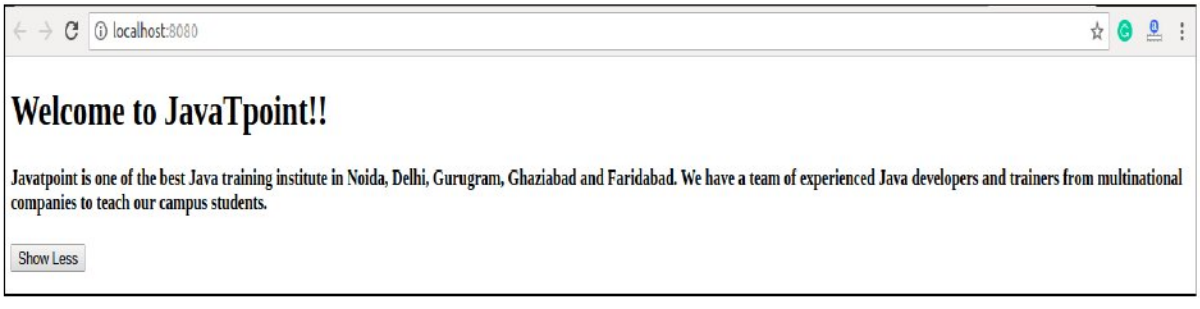
In this example, we are going to add a **button** to the **render**() method. Clicking on this button triggers the toggleDisplayBio() method which displays the desired output.

```
1.          import React, { Component } from 'react';
2.          class App extends React.Component {
3.           constructor() {
4.              super();
5.              this.state = { displayBio: false };
6.              console.log('Component this', this);
7.              this.toggleDisplayBio = this.toggleDisplayBio.bind(this);
8.              }
9.              toggleDisplayBio(){
10.                this.setState({displayBio: !this.state.displayBio});
11.                }
12.             render() {
13.                return (
14.                   <div>
15.                      <h1>Welcome to JavaTpoint!!</h1>
16.                      {
17.                         this.state.displayBio ? (
18.                            <div>
19.                               <p><h4>Javatpoint is one of the best Java training institute in Noida,
    Delhi, Gurugram, Ghaziabad and Faridabad. We have a team of experienced Java developers and
    trainers from multinational companies to teach our campus students.</h4></p>
20.                               <button onClick={this.toggleDisplayBio}> Show Less </button>
21.                         </div>
22.                          ) : (
23.                            <div>
```

```
24.                         <button onClick={this.toggleDisplayBio}> Read More </button>
25.                     </div>
26.                 )
27.             }
28.         </div>
29.     )
30.   }
31. }
32. export default App;
```

Output:

When you click the **Read More** button, you will get the below output, and when you click the **Show Less** button, you will get the output as shown in the above image.



IX. State and Props

It is possible to combine both state and props in your app. You can set the state in the parent component and pass it in the child component using props. It can be shown in the below example.

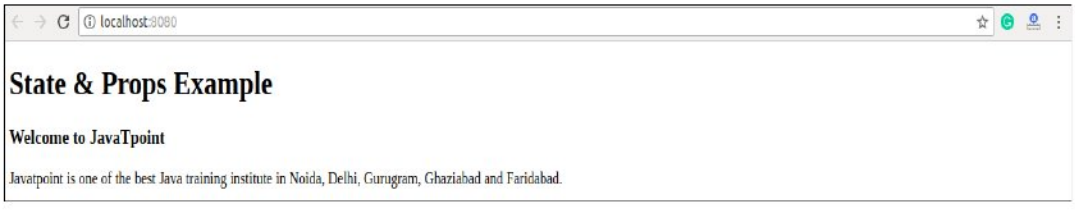
Example: App.js

```
1.  import React, { Component } from 'react';
2.  class App extends React.Component {
3.    constructor(props) {
4.      super(props);
5.      this.state = {
6.        name: "JavaTpoint",
7.      }
8.    }
9.    render() {
10.     return (
11.       <div>
12.         <JTP jtpProp = {this.state.name}/>
13.       </div>
14.     );
15.   }
16. }
17. class JTP extends React.Component {
18.   render() {
19.     return (
20.       <div>
21.         <h1>State & Props Example</h1>
22.         <h3>Welcome to {this.props.jtpProp}</h3>
23.         <p>Javatpoint is one of the best Java training institute in Noida, Delhi, Gurugra
    m, Ghaziabad and Faridabad.</p>
24.       </div>
25.     );
26.   }
27. }
28. export default App;
```

Main.js

```
1.  import React from 'react';
2.  import ReactDOM from 'react-dom';
3.  import App from './App.js';
4.
5.  ReactDOM.render(<App/>, document.getElementById('app'));
```

Output:



X. React Component API

ReactJS component is a top-level API. It makes the code completely individual and reusable in the application. It includes various methods for:

- Creating elements
- Transforming elements
- Fragments

Here, we are going to explain the three most important methods available in the React component API.

1. setState()
2. forceUpdate()
3. findDOMNode()

setState()

This method is used to update the state of the component. This method does not always replace the state immediately. Instead, it only adds changes to the original state. It is a primary method that is used to update the user interface(UI) in response to event handlers and server responses.

Syntax

```
this.stateState(object newState[, function callback]);
```

In the above syntax, there is an optional **callback** function which is executed once setState() is completed and the component is re-rendered.

Example

```
1.  import React, { Component } from 'react';
2.  import PropTypes from 'prop-types';
3.  class App extends React.Component {
4.    constructor() {
5.      super();
6.      this.state = {
7.        msg: "Welcome to JavaTpoint"
8.      };
9.      this.updateSetState = this.updateSetState.bind(this);
10.   }
11.   updateSetState() {
12.     this.setState({
13.       msg:"Its a best ReactJS tutorial"
14.     });
15.   }
16.   render() {
17.     return (
18.       <div>
19.         <h1>{this.state.msg}</h1>
20.         <button onClick = {this.updateSetState}>SET STATE</button>
21.       </div>
22.     );
23.   }
24. }
```

Main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';
ReactDOM.render(<App/>, document.getElementById('app'));
```

Output:

When you click on the **SET STATE** button, you will see the following screen with the updated message.

forceUpdate()

This method allows us to update the component manually.

Syntax

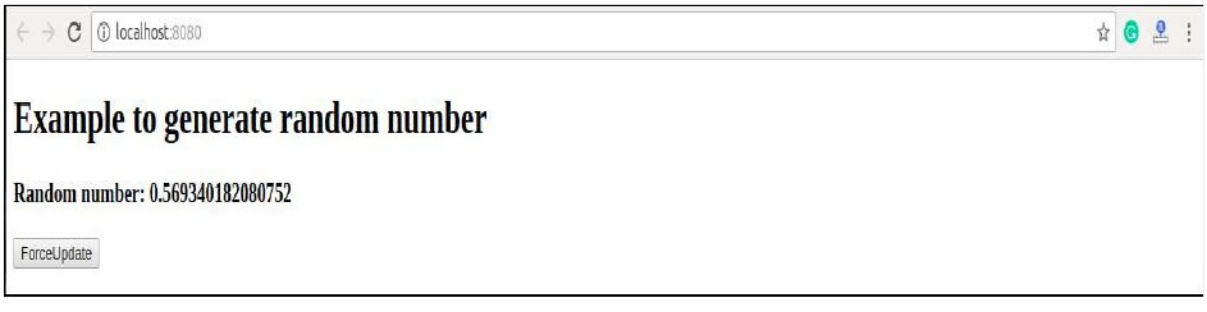
```
Component.forceUpdate(callback);
```

Example: App.js

```
import React, { Component } from 'react';
class App extends React.Component {
  constructor() {
    super();
    this.forceUpdateState = this.forceUpdateState.bind(this);
  }
```

```
7.          forceUpdateState() {
8.            this.forceUpdate();
9.          };
10.         render() {
11.           return (
12.             <div>
13.               <h1>Example to generate random number</h1>
14.               <h3>Random number: {Math.random()}</h3>
15.               <button onClick = {this.forceUpdateState}>ForceUpdate</button>
16.             </div>
17.           );
18.         }
19.       }
20.       export default App;
```

Output:



Each time when you click on **ForceUpdate** button, it will generate the **random** number. It can be shown in the below image.

XI. React Component Life-Cycle

In ReactJS, every component creation process involves various lifecycle methods. These lifecycle methods are termed as component's lifecycle. These lifecycle methods are not very complicated and called at various points during a component's life. The lifecycle of the component is divided into **four phases.** They are:

1. Initial Phase
2. Mounting Phase
3. Updating Phase
4. Unmounting Phase

Each phase contains some lifecycle methods that are specific to the particular phase. Let us discuss each of these phases one by one.

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.

- **getDefaultProps()**
  It is used to specify the default value of this.props. It is invoked before the creation of the component or any props from the parent is passed into it.
- **getInitialState()**
  It is used to specify the default value of this.state. It is invoked before the creation of the component.

2. Mounting Phase

In this phase, the instance of a component is created and inserted into the DOM. It consists of the

- **componentWillMount()**

  This is invoked immediately before a component gets rendered into the DOM. In the case, when you call **setState()** inside this method, the component will not **re-render**.

- **componentDidMount()**

  This is invoked immediately after a component gets rendered and placed on the DOM. Now, you can do any DOM querying operations.

- **render()**

  This method is defined in each and every component. It is responsible for returning a single root **HTML node** element. If you don't want to render anything, you can return a **null** or **false** value.

3. Updating Phase

It is the next phase of the lifecycle of a react component. Here, we get new **Props** and change **State**. This phase also allows to handle user interaction and provide communication with the components hierarchy. The main aim of this phase is to ensure that the component is displaying the latest version of itself. Unlike the Birth or Death phase, this phase repeats again and again. This phase consists of the following methods.

- **componentWillRecieveProps()**

  It is invoked when a component receives new props. If you want to update the state in response to prop changes, you should compare this.props and nextProps to perform state transition by using **this.setState()** method.

- **shouldComponentUpdate()**

  It is invoked when a component decides any changes/updation to the DOM. It allows you to control the component's behavior of updating itself. If this method returns true, the component will update. Otherwise, the component will skip the updating.

- **componentWillUpdate()**

  It is invoked just before the component updating occurs. Here, you can't change the component state by invoking **this.setState()** method. It will not be called, if **shouldComponentUpdate()** returns false.

- **render()**

  It is invoked to examine **this.props** and **this.state** and return one of the following types: React elements, Arrays and fragments, Booleans or null, String and Number. If shouldComponentUpdate() returns false, the code inside render() will be invoked again to ensure that the component displays itself properly.

- **componentDidUpdate()**

  It is invoked immediately after the component updating occurs. In this method, you can put any code inside this which you want to execute once the updating occurs. This method is not invoked for the initial render.

4. Unmounting Phase

It is the final phase of the react component lifecycle. It is called when a component instance is **destroyed** and **unmounted** from the DOM. This phase contains only one method and is given below.

- **componentWillUnmount()**

  This method is invoked immediately before a component is destroyed and unmounted permanently. It performs any necessary **cleanup** related task such as invalidating timers, event listener, canceling network requests, or cleaning up DOM elements. If a component instance is unmounted, you cannot mount it again.

Example

```
1.  import React, { Component } from 'react';
2.
3.  class App extends React.Component {
4.    constructor(props) {
5.      super(props);
6.      this.state = {hello: "JavaTpoint"};
7.      this.changeState = this.changeState.bind(this)
8.    }
9.    render() {
```

```
10.         return (
11.           <div>
12.             <h1>ReactJS component's Lifecycle</h1>
13.             <h3>Hello {this.state.hello}</h3>
14.             <button onClick = {this.changeState}>Click Here!</button>
15.           </div>
16.         );
17.       }
18.       componentWillMount() {
19.         console.log('Component Will MOUNT!')
20.       }
21.       componentDidMount() {
22.         console.log('Component Did MOUNT!')
23.       }
24.       changeState(){
25.         this.setState({hello:"All!!- Its a great reactjs tutorial."});
26.       }
27.       componentWillReceiveProps(newProps) {
28.         console.log('Component Will Recieve Props!')
29.       }
30.       shouldComponentUpdate(newProps, newState) {
31.         return true;
32.       }
33.       componentWillUpdate(nextProps, nextState) {
34.         console.log('Component Will UPDATE!');
35.       }
36.       componentDidUpdate(prevProps, prevState) {
37.         console.log('Component Did UPDATE!')
38.       }
39.       componentWillUnmount() {
40.         console.log('Component Will UNMOUNT!')
```

```
41.             }
42.         }
43.         export default App;
```

Output:

When you click on the **Click Here** Button, you get the updated result which is shown in the below screen.

XII. React Events

Just like HTML DOM events, React can perform actions based on user events. React has the same events as HTML: click, change, mouseover etc.

Adding Events

React events are written in camelCase syntax:

onClick instead of onclick.

React event handlers are written inside curly braces:

onClick={shoot} instead of onClick="shoot()".

React:

<button onClick={shoot}>Take the Shot!</button>

HTML:

```
<button onclick="shoot()">Take the Shot!</button>
```

Example:

Put the shoot function inside the Football component:

```
function Football() {
  const shoot = () => {
    alert("Great Shot!");
  }
  return (
    <button onClick={shoot}>Take the shot!</button>
  );
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Football />);
```

Passing Arguments

To pass an argument to an event handler, use an arrow function.

Example:

Send "Goal!" as a parameter to the shoot function, using arrow function:

```
function Football() {
  const shoot = (a) => {
    alert(a);
  }
  return (
    <button onClick={() => shoot("Goal!")}>Take the shot!</button>
```

```
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Football />);
```

XIII. ReactLocalStorage

Installation

```
npm install reactjs-localstorage
    or
yarn add reactjs-localstorage
```

LocalStorage is a web storage object to store the data on the user's computer locally, which means the stored data is saved across browser sessions and the data stored has no expiration time.

Syntax

```
// To store data
localStorage.setItem('Name', 'Rahul');

// To retrieve data
localStorage.getItem('Name');

// To clear a specific item
localStorage.removeItem('Name');

// To clear the whole data stored in localStorage
localStorage.clear();
```

Set, retrieve and remove data in localStorage

In this example, we will build a React application which takes the username and password from the user and stores it as an item in the localStorage of the user's computer.

Example: **App.jsx**

```
import React, { useState } from 'react';
const App = () => {
  const [name, setName] = useState('');
  const [pwd, setPwd] = useState('');

  const handle = () => {
    localStorage.setItem('Name', name);
    localStorage.setItem('Password', pwd);
  };
  const remove = () => {
    localStorage.removeItem('Name');
    localStorage.removeItem('Password');
  };
  return (
    <div className="App">
      <h1>Name of the user:</h1>
      <input
        placeholder="Name"
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
      <h1>Password of the user:</h1>
      <input
        type="password"
        placeholder="Password"
        value={pwd}
        onChange={(e) => setPwd(e.target.value)}
      />
      <div>
        <button onClick={handle}>Done</button>
```

```
        </div>
        {localStorage.getItem('Name') && (
          <div>
            Name: <p>{localStorage.getItem('Name')}</p>
          </div>
        )}
        {localStorage.getItem('Password') && (
          <div>
            Password: <p>{localStorage.getItem('Password')}</p>
          </div>
        )}
        <div>
          <button onClick={remove}>Remove</button>
        </div>
      </div>
    );
};
export default App;
```

In the above example, when the **Done** button is clicked, the handle function is executed which will set the items in the localStorage of the user and display it. But when the **Remove** button is clicked, the **remove** function is executed which will remove the items from the localStorage.

Output

This will produce the following result.

XIV. Lifting state up in React.js

Often there will be a need to share state between different components. The common approach to share state between two components is to move the state to common parent of the two components. This approach is called as lifting state up in React.js

With the shared state, changes in state reflect in relevant components simultaneously. This is a single source of truth for shared state components.

Example

We have an App component containing PlayerContent and PlayerDetails component. PlayerContent shows the player name buttons. PlayerDetails shows the details of the in one line. The app component contains the state for both the component. The selected player is shown once we click on the one of the player button.

App.js

```
import React from 'react';
import PlayerContent from './PlayerContent';
import PlayerDetails from './PlayerDetails';
import './App.css';
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { selectedPlayer:[0,0], playerName: ''};
    this.updateSelectedPlayer = this.updateSelectedPlayer.bind(this);
  }
  updateSelectedPlayer(id, name) {
    var arr = [0, 0, 0, 0];
    arr[id] = 1;
    this.setState({
      playerName: name,
      selectedPlayer: arr
    });
  }
  render () {
    return (
      <div>
        <PlayerContent active={this.state.selectedPlayer[0]}
        clickHandler={this.updateSelectedPlayer} id={0} name="David"/>
        <PlayerContent active={this.state.selectedPlayer[1]}
        clickHandler={this.updateSelectedPlayer} id={1} name="Steve"/>
        <PlayerDetails name={this.state.playerName}/>
      </div>
    );
  }
```

```
export default App;
```

PlayerContent.js

```
import React , { Component} from 'react';
class PlayerContent extends Component {
  render () {
    return (
      <button onClick={() => {this.props.clickHandler(this.props.id, this.props.name)}}
style={{color: this.props.active? 'red': 'blue'}}>{this.props.name}
      </button>
    );
  }
}
export default PlayerContent;
```

PlayerDetails.js

```
import React, { Component } from 'react';
class PlayerDetails extends Component {
  render () {
    return (
      <div>{this.props.name}
      </div>
    );
  }
}
export default PlayerDetails;
```

Output

the output displays as −

XV. Composition and Inheritance

Composition and inheritance are the approaches to use multiple components together in React.js. This helps in code reuse. React recommend using composition instead of inheritance as much as possible and inheritance should be used in very specific cases only.

Example to understand it −

We have a component to input username.

Inheritance

```
class UserNameForm extends React.Component {
  render() {
    return (
      <div>
        <input type="text" />
      </div>
    );
  }
}
ReactDOM.render(
  < UserNameForm />,
  document.getElementById('root'));
```

This is simple to just input the name. We will have two more components to create and update the username field.

```
class UserNameForm extends React.Component {
  render() {
    return (
      <div>
        <input type="text" />
      </div>
    );
  }
}
class CreateUserName extends UserNameForm {
  render() {
    const parent = super.render();
    return (
      <div>
        {parent}
        <button>Create</button>
      </div>
    )
  }
}
class UpdateUserName extends UserNameForm {
  render() {
    const parent = super.render();
    return (
      <div>
        {parent}
        <button>Update</button>
      </div>
    )
```

```
}
ReactDOM.render(
  (<div>
    < CreateUserName />
    < UpdateUserName />
  </div>), document.getElementById('root')
);
```

We extended the UserNameForm component and extracted its method in child component using super.render();

Composition

```
class UserNameForm extends React.Component {
  render() {
    return (
      <div>
        <input type="text" />
      </div>
    );
  }
}
class CreateUserName extends React.Component {
  render() {
    return (
      <div>
        < UserNameForm />
        <button>Create</button>
      </div>
    )
  }
```

```
}
class UpdateUserName extends React.Component {
  render() {
    return (
      <div>
        < UserNameForm />
        <button>Update</button>
      </div>
    )
  }
}
ReactDOM.render(
  (<div>
    <CreateUserName />
    <UpdateUserName />
  </div>), document.getElementById('root')
);
```

Use of composition is simpler than inheritance and easy to maintain the complexity.

UNIT III ADVANCED NODE JS AND DATABASE

Introduction to NoSQL databases – MongoDB system overview - Basic querying with MongoDB shell – Request body parsing in Express – NodeJS MongoDB connection – Adding and retrieving data to MongoDB from NodeJS – Handling SQL databases from NodeJS – Handling Cookies in NodeJS – Handling User Authentication with node js

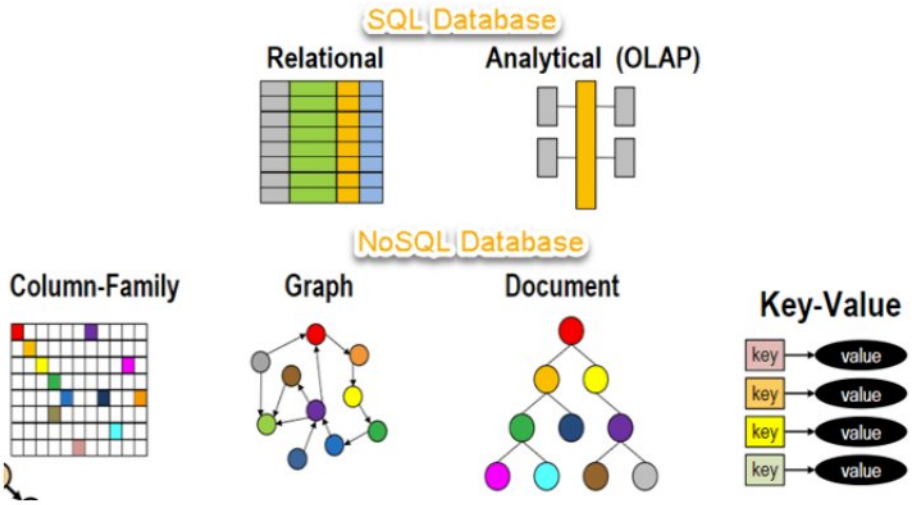
3.1 Introduction to NoSQL databases

What is NoSQL?

NoSQL Database is a non-relational Data Management System, that does not require a fixed schema. It avoids joins, and is easy to scale. The major purpose of using a NoSQL database is for distributed data stores with humongous data storage needs. NoSQL is used for Big data and real-time web apps. For example, companies like Twitter, Facebook and Google collect terabytes of user data every single day.

NoSQL database stands for "Not Only SQL" or "Not SQL." Though a better term would be "NoREL", NoSQL caught on. Carl Strozz introduced the NoSQL concept in 1998.

Traditional RDBMS uses SQL syntax to store and retrieve data for further insights. Instead, a NoSQL database system encompasses a wide range of database technologies that can store structured, semi-structured, unstructured and polymorphic data. Let's understand about NoSQL with a diagram in this NoSQL database tutorial:



Why NoSQL?

The concept of NoSQL databases became popular with Internet giants like Google, Facebook, Amazon, etc. who deal with huge volumes of data. The system response time

To resolve this problem, we could “scale up” our systems by upgrading our existing hardware. This process is expensive.

Brief History of NoSQL Databases

- 1998- Carlo Strozzi use the term NoSQL for his lightweight, open-source relational database
- 2000- Graph database Neo4j is launched
- 2004- Google BigTable is launched
- 2005- CouchDB is launched
- 2007- The research paper on Amazon Dynamo is released
- 2008- Facebooks open sources the Cassandra project
- 2009- The term NoSQL was reintroduced

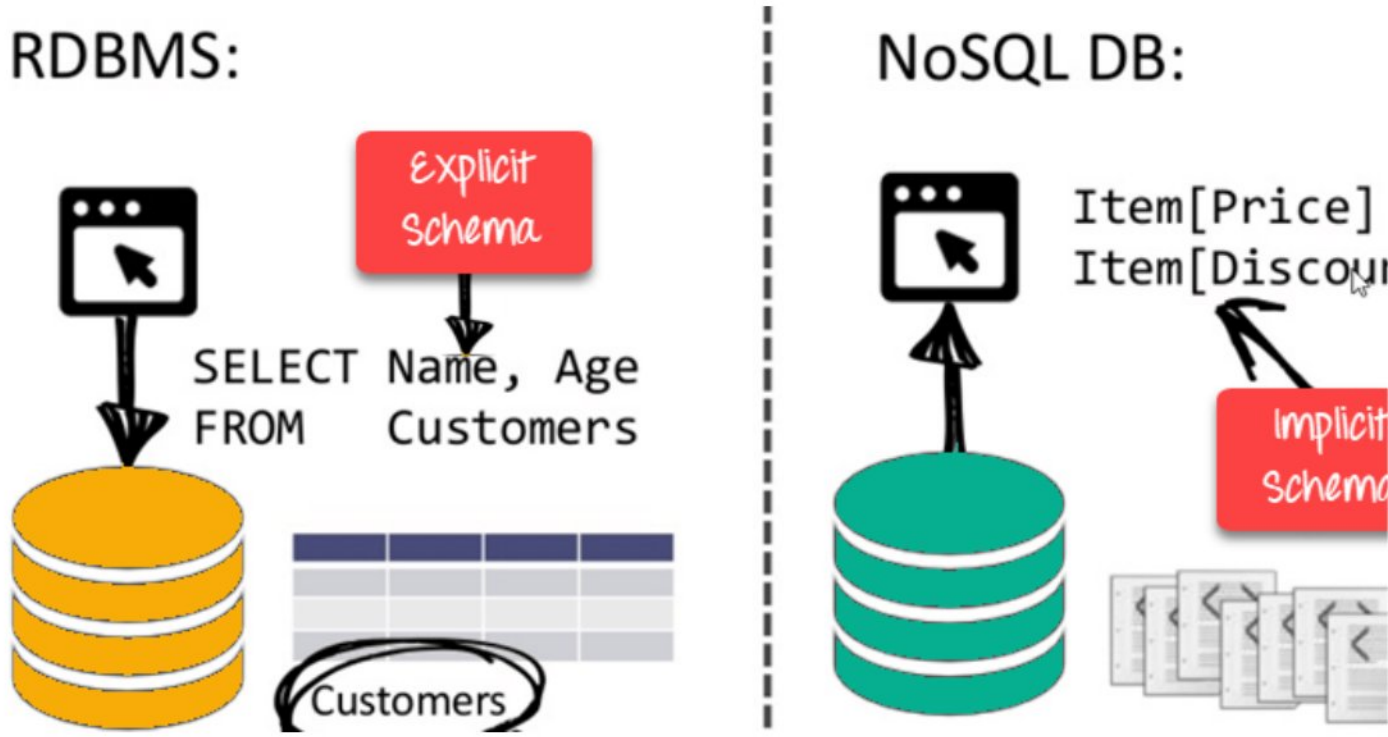
Features of NoSQL

Non-relational

- NoSQL databases never follow the relational model
- Never provide tables with flat fixed-column records
- Work with self-contained aggregates or BLOBs
- Doesn’t require object-relational mapping and data normalization
- No complex features like query languages, query planners,referential integrity joins, ACID

Schema-free

- NoSQL databases are either schema-free or have relaxed schemas
- Do not require any sort of definition of the schema of the data
- Offers heterogeneous structures of data in the same domain



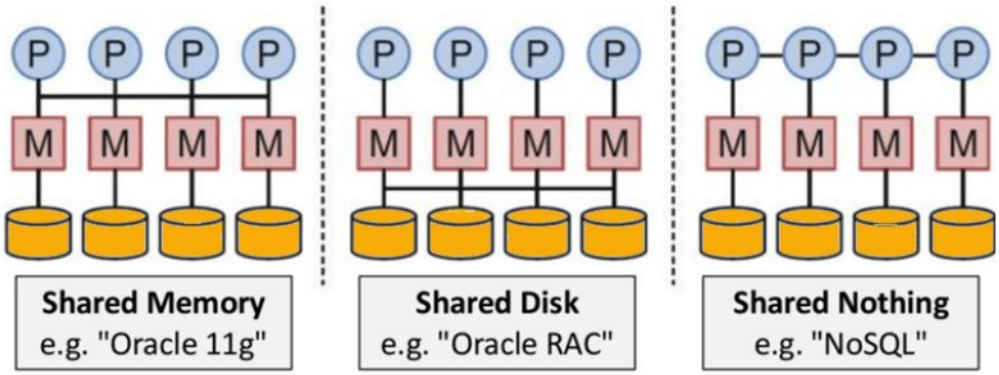
NoSQL is Schema-Free

Simple API

- Offers easy to use interfaces for storage and querying data provided
- APIs allow low-level data manipulation & selection methods
- Text-based protocols mostly used with HTTP REST with JSON
- Mostly used no standard based NoSQL query language
- Web-enabled databases running as internet-facing services

Distributed

- Multiple NoSQL databases can be executed in a distributed fashion
- Offers auto-scaling and fail-over capabilities
- Often ACID concept can be sacrificed for scalability and throughput
- Mostly no synchronous replication between distributed nodes Asynchronous Multi-Master Replication, peer-to-peer, HDFS Replication
- Only providing eventual consistency
- Shared Nothing Architecture. This enables less coordination and higher distribution.

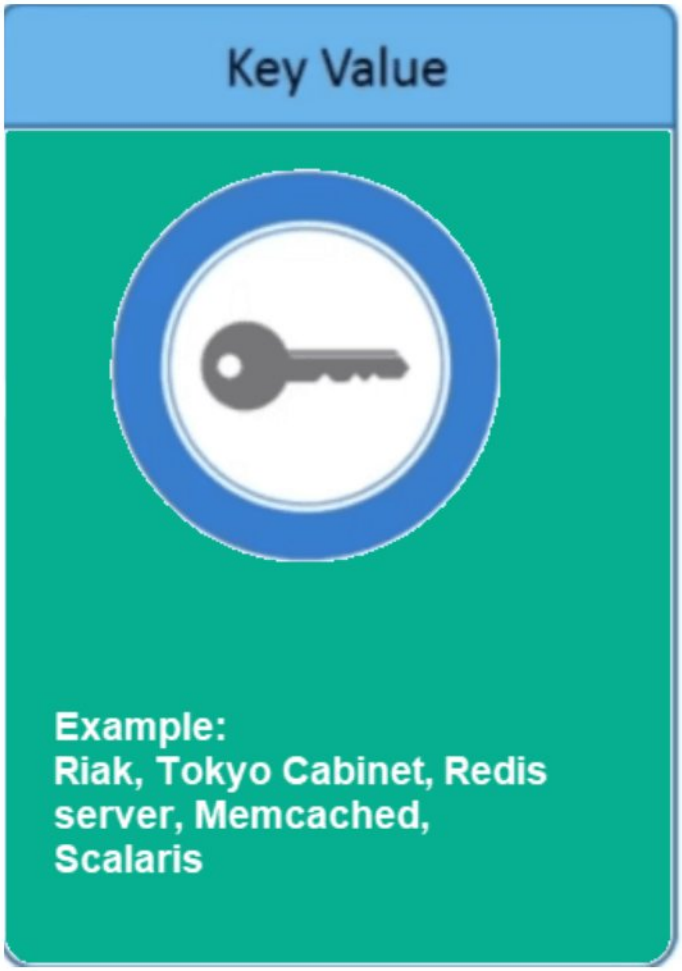


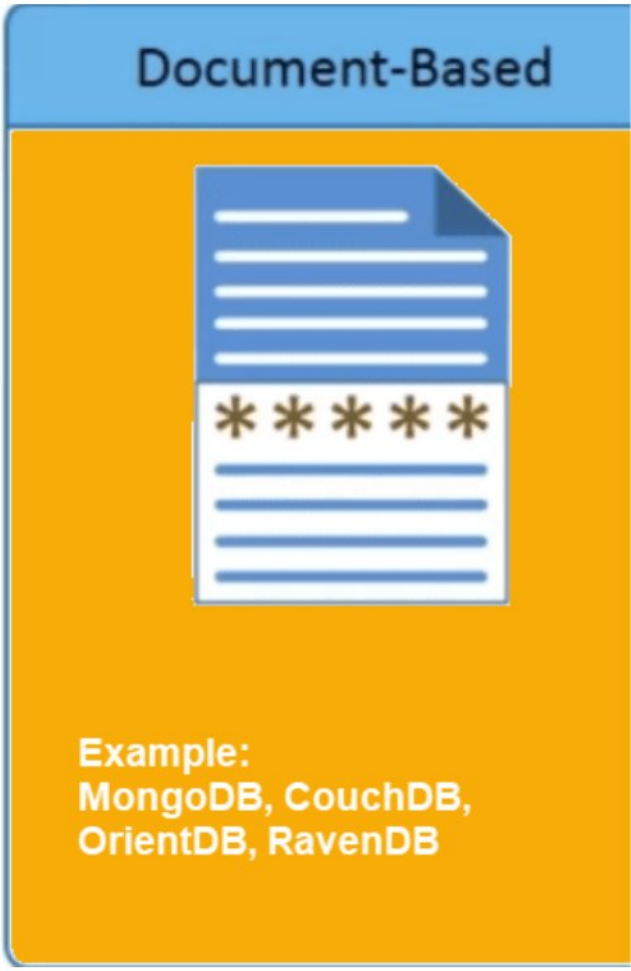
NoSQL is Shared Nothing.

Types of NoSQL Databases

NoSQL Databases are mainly categorized into four types: Key-value pair, Column-oriented, Graph-based and Document-oriented. Every category has its unique attributes and limitations. None of the above-specified database is better to solve all the problems. Users should select the database based on their product needs.
Types of NoSQL Databases:

- Key-value Pair Based
- Column-oriented Graph
- Graphs based
- Document-oriented





Key Value Pair Based

Data is stored in key/value pairs. It is designed in such a way to handle lots of data and heavy load.

Key-value pair storage databases store data as a hash table where each key is unique, and the value can be a JSON, BLOB(Binary Large Objects), string, etc.

For example, a key-value pair may contain a key like “Website” associated with a value like “Guru99”.

Key	Value
Name	Joe Bloggs
Age	42
Occupation	Stunt Double
Height	175cm
Weight	77kg

It is one of the most basic NoSQL database example. This kind of NoSQL database is used as a collection, dictionaries, associative arrays, etc. Key value stores help the developer to store schema-less data. They work best for shopping cart contents.

Redis, Dynamo, Riak are some NoSQL examples of key-value store DataBases. They are all based on Amazon's Dynamo paper.

Column-based

Column-oriented databases work on columns and are based on BigTable paper by Google. Every column is treated separately. Values of single column databases are stored contiguously.

ColumnFamily			
Row Key	Column Name		
	Key	Key	Key
	Value	Value	Value
	Column Name		
	Key	Key	Key
	Value	Value	Value

Column based NoSQL database

They deliver high performance on aggregation queries like SUM, COUNT, AVG, MIN etc. as the data is readily available in a column.

Column-based NoSQL databases are widely used to manage data warehouses, business intelligence, CRM, Library card catalogs,

HBase, Cassandra, HBase, Hypertable are NoSQL query examples of column based database.

Document-Oriented:

Document-Oriented NoSQL DB stores and retrieves data as a key value pair but the value part is stored as a document. The document is stored in JSON or XML formats. The value is understood by the DB and can be queried.

Relational Vs. Document

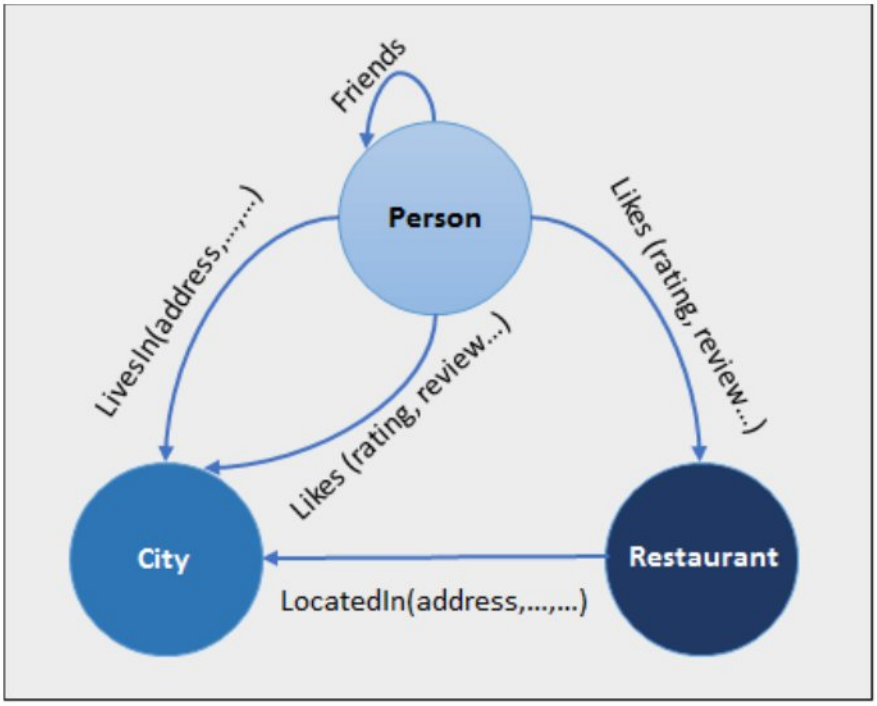
In this diagram on your left you can see we have rows and columns, and in the right, we have a document database which has a similar structure to JSON. Now for the relational database, you have to know what columns you have and so on. However, for a document database, you have data store like JSON object. You do not require to define which make it flexible.

The document type is mostly used for CMS systems, blogging platforms, real-time analytics & e-commerce applications. It should not use for complex transactions which require multiple operations or queries against varying aggregate structures.

Amazon SimpleDB, CouchDB, MongoDB, Riak, Lotus Notes, MongoDB, are popular Document originated DBMS systems.

Graph-Based

A graph type database stores entities as well the relations amongst those entities. The entity is stored as a node with the relationship as edges. An edge gives a relationship between nodes. Every node and edge has a unique identifier.



Compared to a relational database where tables are loosely connected, a Graph database is a multi-relational in nature. Traversing relationship is fast as they are already captured into the DB, and there is no need to calculate them.

Graph base database mostly used for social networks, logistics, spatial data.

Neo4J, Infinite Graph, OrientDB, FlockDB are some popular graph-based databases.

Advantages of NoSQL

- Can be used as Primary or Analytic Data Source
- Big Data Capability
- No Single Point of Failure
- Easy Replication
- No Need for Separate Caching Layer
- It provides fast performance and horizontal scalability.
- Can handle structured, semi-structured, and unstructured data with equal effect
- Object-oriented programming which is easy to use and flexible
- NoSQL databases don't need a dedicated high-performance server
- Support Key Developer Languages and Platforms
- Simple to implement than using RDBMS
- It can serve as the primary data source for online applications.
- Handles big data which manages data velocity, variety, volume, and complexity
- Excels at distributed database and multi-data center operations
- Eliminates the need for a specific caching layer to store data
- Offers a flexible schema design which can easily be altered without downtime or service disruption

Disadvantages of NoSQL

- No standardization rules
- Limited query capabilities
- RDBMS databases and tools are comparatively mature
- It does not offer any traditional database capabilities, like consistency when multiple transactions are performed simultaneously.
- When the volume of data increases it is difficult to maintain unique values as keys become difficult
- Doesn't work as well with relational data
- The learning curve is stiff for new developers
- Open source options so not so popular for enterprises.

3.2 MongoDB system overview

What is MongoDB?

MongoDB is a document-oriented NoSQL database used for high volume data storage. Instead of using tables and rows as in the traditional relational databases, MongoDB makes use of collections and documents. Documents consist of key-value pairs which are the basic unit of data in MongoDB. Collections contain sets of documents and function which is the equivalent of relational database tables. MongoDB is a database which came into light around the mid-2000s.

MongoDB Features

1. Each database contains collections which in turn contains documents. Each document can be different with a varying number of fields. The size and content of each document can be different from each other.
2. The document structure is more in line with how developers construct their classes and objects in their respective programming languages. Developers will often say that their classes are not rows and columns but have a clear structure with key-value pairs.
3. The rows (or documents as called in MongoDB) doesn't need to have a schema defined beforehand. Instead, the fields can be created on the fly.
4. The data model available within MongoDB allows you to represent hierarchical relationships, to store arrays, and other more complex structures more easily.
5. Scalability – The MongoDB environments are very scalable. Companies across the world have defined clusters with some of them running 100+ nodes with around millions of documents within the database

Key Components of MongoDB Architecture

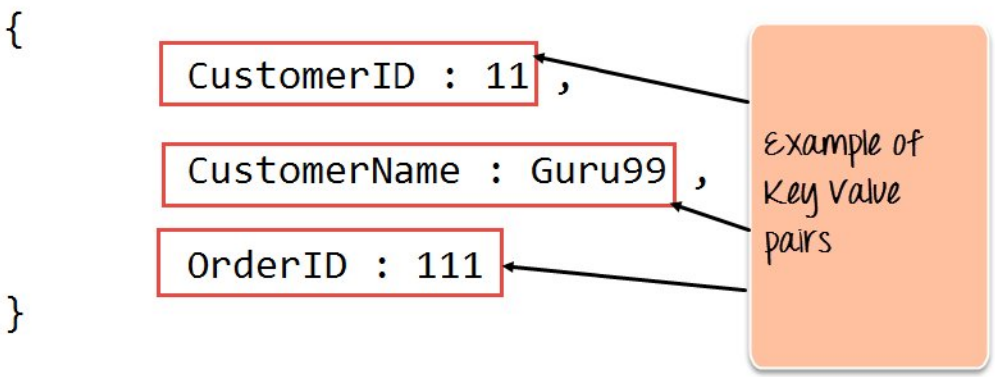
Below are a few of the common terms used in MongoDB

1. **_id** – This is a field required in every MongoDB document. The _id field represents a unique value in the MongoDB document. The _id field is like the document's primary

create the field. So for example, if we see the example of the above customer table, Mongo DB will add a 24 digit unique identifier to each document in the collection.

_Id	CustomerID	CustomerName	OrderID
563479cc8a8a4246bd27d784	11	Guru99	111
563479cc7a8a4246bd47d784	22	Trevor Smith	222
563479cc9a8a4246bd57d784	33	Nicole	333

2. **Collection** – This is a grouping of MongoDB documents. A collection is the equivalent of a table which is created in any other RDMS such as Oracle or MS SQL. A collection exists within a single database. As seen from the introduction collections don't enforce any sort of structure.
3. **Cursor** – This is a pointer to the result set of a query. Clients can iterate through a cursor to retrieve results.
4. **Database** – This is a container for collections like in RDMS wherein it is a container for tables. Each database gets its own set of files on the file system. A MongoDB server can store multiple databases.
5. **Document** – A record in a MongoDB collection is basically called a document. The document, in turn, will consist of field name and values.
6. **Field** – A name-value pair in a document. A document has zero or more fields. Fields are analogous to columns in relational databases.The following diagram shows an example of Fields with Key value pairs. So in the example below CustomerID and 11 is one of the key value pair's defined in the document.



7. **JSON** – This is known as JavaScript Object Notation. This is a human-readable, plain text format for expressing structured data. JSON is currently supported in many programming languages.

3.3 Basic querying with MongoDB shell

3.4 Request body parsing in Express

Express body-parser is an npm library used to process data sent through an HTTP request body. It exposes four express middlewares for parsing text, JSON, url-encoded and raw data set through an HTTP request body. These **middlewares** are functions that process incoming

body-parser doesn't have to be installed as a separate package because it is a dependency of express version 4.16.0+. body-parser isn't a dependency between version 4.0.0 and 4.16.0, so it will be installed separately in projects locked to those versions. body-parser middlewares will be required by express in versions of express with body-parser dependency. Versions of Express without body-parser will have to install it separately.

3.5 NodeJS MongoDB connection

Access MongoDB in Node.js

Learn how to access document-based database MongoDB using Node.js in this section.

In order to access MongoDB database, we need to install MongoDB drivers. To install native mongodb drivers using NPM, open command prompt and write the following command to install MongoDB driver in your application.

```
npm install mongodb --save
```

This will include mongodb folder inside node_modules folder. Now, start the MongoDB server using the following command. (Assuming that your MongoDB database is at C:\MyNodeJSConsoleApp\MyMongoDB folder.)

```
mongod -dbpath C:\MyNodeJSConsoleApp\MyMongoDB
```

Connecting MongoDB

The following example demonstrates connecting to the local MongoDB database.

app.js

```
var MongoClient = require('mongodb').MongoClient;

// Connect to the db
MongoClient.connect("mongodb://localhost:27017/MyDb", function (err, db) {

    if(err) throw err;

    //Write databse Insert/Update/Query code here..

});
```

In the above example, we have imported mongodb module (native drivers) and got the

reference of specified MongoDB database. The specified URL "mongodb://localhost:27017/MyDb" points to your local MongoDB database created in MyMongoDB folder. The connect() method returns the database reference if the specified database is already exists, otherwise it creates a new database.

Now you can write insert/update or query the MongoDB database in the callback function of the connect() method using db parameter.

Insert Documents

The following example demonstrates inserting documents into MongoDB database.

app.js

```
var MongoClient = require('mongodb').MongoClient;

// Connect to the db
MongoClient.connect("mongodb://localhost:27017/MyDb", function (err, db) {

   db.collection('Persons', function (err, collection) {

      collection.insert({ id: 1, firstName: 'Steve', lastName: 'Jobs' });
      collection.insert({ id: 2, firstName: 'Bill', lastName: 'Gates' });
      collection.insert({ id: 3, firstName: 'James', lastName: 'Bond' });
      db.collection('Persons').count(function (err, count) {
         if (err) throw err;

         console.log('Total Rows: ' + count);
      });
   });

});
```

In the above example, db.collection() method creates or gets the reference of the specified collection. Collection is similar to table in relational database. We created a collection called Persons in the above example and insert three documents (rows) in it. After that, we display the count of total documents stored in the collection.

Running the above example displays the following result.

```
> node app.js
```

```
Total Rows: 3
```

3.6 Adding and retrieving data to MongoDB from NodeJS

3.7 Handling SQL databases from NodeJS

3.8 Handling Cookies in NodeJS

3.9 Handling User Authentication with node js

UNIT V APP IMPLEMENTATION IN CLOUD

Cloud providers Overview – Virtual Private Cloud – Scaling (Horizontal and Vertical) – Virtual Machines, Ethernet and Switches – Docker Container – Kubernetes

5.1 Cloud providers Overview

Overview

Cloud service providers are companies that establish public clouds, manage private clouds, or offer on-demand cloud computing components (also known as cloud computing services) like Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service(SaaS). Cloud services can reduce business process costs when compared to on-premise IT.

These clouds aren't usually deployed as a standalone infrastructure solution, but rather as part of a hybrid cloud.

Why use a cloud provider?

Using a cloud provider is a helpful way to access computing services that you would otherwise have to provide on your own, such as:

- **Infrastructure:** The foundation of every computing environment. This infrastructure could include networks, database services, data management, data storage (known in this context as cloud storage), servers (cloud is the basis for serverless computing), and virtualization.
- **Platforms:** The tools needed to create and deploy applications. These platforms could include operating systems like Linux®, middleware, and runtime environments.
- **Software:** Ready-to-use applications. This software could be custom or standard applications provided by independent service providers.

Certified cloud providers

There are a handful of well-known, major public cloud companies—such as Alibaba Cloud, Amazon Web Services (AWS), Google Cloud Platform (GCP), IBM Cloud, Oracle Cloud, and Microsoft Azure—but there are also hundreds of other cloud computing providers all over the world.

What are the benefits of using a cloud service provider?

Using a cloud provider has benefits and challenges. Companies considering using these services should think about how these factors would affect their priorities and risk profile, for both the present and long term. Individual CSPs have their own strengths and weaknesses, which are worth considering.

Benefits

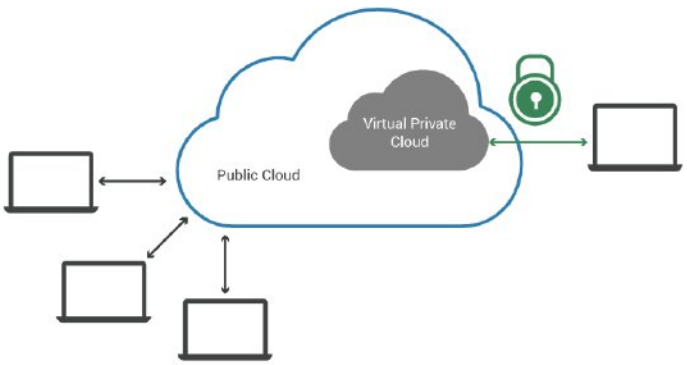
- **Cost and flexibility.** The pay-as-you-go model of cloud services enables organizations to only pay for the resources they consume. Using a cloud service provider also eliminates

the need for IT-related capital equipment purchases. Organizations should review the details of cloud pricing to accurately break down cloud costs.

- **Scalability.** Customer organizations can easily scale up or down the IT resources they use based on business demands.
- **Mobility.** Resources and services purchased from a cloud service provider can be accessed from any physical location that has a working network connection.
- **Disaster recovery.** Cloud services typically offer quick and reliable disaster recovery.

5.2 Virtual Private Cloud

What is a virtual private cloud (VPC)?



A virtual private cloud (VPC) is a secure, isolated private cloud hosted within a public cloud. VPC customers can run code, store data, host websites, and do anything else they could do in an ordinary private cloud, but the private cloud is hosted remotely by a public cloud provider. (Not all private clouds are hosted in this fashion.) VPCs combine the scalability and convenience of public cloud computing with the data isolation of private cloud computing.

Imagine a public cloud as a crowded restaurant, and a virtual private cloud as a reserved table in that crowded restaurant. Even though the restaurant is full of people, a table with a "Reserved" sign on it can only be accessed by the party who made the reservation. Similarly, a public cloud is crowded with various cloud customers accessing computing resources – but a VPC reserves some of those resources for use by only one customer.

What is a public cloud? What is a private cloud?

A public cloud is shared cloud infrastructure. Multiple customers of the cloud vendor access that same infrastructure, although their data is not shared – just like every person in a restaurant orders from the same kitchen, but they get different dishes. Public cloud service

The technical term for multiple separate customers accessing the same cloud infrastructure is "multitenancy" A private cloud, however, is single-tenant. A private cloud is a cloud service that is exclusively offered to one organization. A virtual private cloud (VPC) is a private cloud within a public cloud; no one else shares the VPC with the VPC customer.

How is a VPC isolated within a public cloud?

A VPC isolates computing resources from the other computing resources available in the public cloud. The key technologies for isolating a VPC from the rest of the public cloud are:

Subnets: A subnet is a range of IP addresses within a network that are reserved so that they're not available to everyone within the network, essentially dividing part of the network for private use. In a VPC these are private IP addresses that are not accessible via the public Internet, unlike typical IP addresses, which are publicly visible.

VLAN: A LAN is a local area network, or a group of computing devices that are all connected to each other without the use of the Internet. A VLAN is a virtual LAN. Like a subnet, a VLAN is a way of partitioning a network, but the partitioning takes place at a different layer within the OSI model (layer 2 instead of layer 3).

VPN: A virtual private network (VPN) uses encryption to create a private network over the top of a public network. VPN traffic passes through publicly shared Internet infrastructure – routers, switches, etc. – but the traffic is scrambled and not visible to anyone.

A VPC will have a dedicated subnet and VLAN that are only accessible by the VPC customer. This prevents anyone else within the public cloud from accessing computing resources within the VPC – effectively placing the "Reserved" sign on the table. The VPC customer connects via VPN to their VPC, so that data passing into and out of the VPC is not visible to other public cloud users.

Some VPC providers offer additional customization with:

- **Network Address Translation (NAT):** This feature matches private IP addresses to a public IP address for connections with the public Internet. With NAT, a public-facing website or application could run in a VPC.

- **BGP route configuration:** Some providers allow customers to customize BGP routing tables for connecting their VPC with their other infrastructure. (Learn how BGP works.)

What are the advantages of using a VPC instead of a private cloud?

Scalability: Because a VPC is hosted by a public cloud provider, customers can add more computing resources on demand.

Easy hybrid cloud deployment: It's relatively simple to connect a VPC to a public cloud or to on-premises infrastructure via the VPN. (Learn about hybrid clouds and their advantages.)

Better performance: Cloud-hosted websites and applications typically perform better than those hosted on local on-premises servers.

Better security: The public cloud providers that offer VPCs often have more resources for updating and maintaining the infrastructure, especially for small and mid-market businesses. For large enterprises or any companies that face extremely tight data security regulations, this is less of an advantage.

5.3 Scaling (Horizontal and Vertical)

If your business website or application becomes popular in the market or there is an increase in demand, you must broaden your view of user accessibility and performance. What do you do? The answer for this is usually some type of scalability of your IT infrastructure. When talking about scalability in cloud computing, you will often hear about **two ways of scaling: horizontal or vertical**. we will look deeper into these terms and also into **AWS (Amazon Web Services) scalability** and which services you can use.

What is scalability?
Cloud scalability refers to the ability to increase or decrease IT resources (virtual machines, databases, networks) as needed to meet changing needs. Scalability is one of the **main advantages of the cloud** and the main driving force for its popularity in businesses.

Public cloud providers such as **AWS (Amazon Web Services) already have all the infrastructure in place**; in the past, when scaling had to be done using on-premises infrastructure, the process could take weeks or months and require capital investment. Systems have four general areas that **scalability can apply to:**

- **CPU**
- **Disk I/O**
- **Memory**
- **Network I/O**

The **main benefit of the scalable architecture is performance** and the ability to handle bursts of traffic or heavy loads with little or no notice.

What is horizontal scaling?
To scale horizontally (scaling in or out), you add more resources like virtual machines to your system to spread out the workload across them. Horizontal scaling is especially important for companies that need **high availability** services with a requirement for minimal downtime.

Benefits of horizontal scaling
Horizontal scaling **increases high availability** because as long as you are spreading your infrastructure across multiple areas, if one machine fails, you can just use one of the other ones.

Because you're adding a machine, you need **fewer periods of downtime** and don't have to switch the old machine off while scaling. There may never be a need for downtime if you scale effectively.

And here are some simpler advantages of horizontal scaling:

- Easy to resize according to your needs
- Immediate and continuous availability
- Cost can be linked to usage and you don't always have to pay for peak demand

Disadvantages of horizontal scaling

The main disadvantage of horizontal scaling is that it **increases the complexity of the maintenance and operations** of your architecture, but there are services in the AWS environment to solve this issue.

- Architecture design and deployment can be very complicated
- A limited amount of software that can take advantage of horizontal scaling

What is vertical scaling?

Through vertical scaling (scaling up or down), you can increase or decrease the capacity of existing services/instances by upgrading the memory (RAM), storage, or processing power (CPU). Usually, this means that the expansion has an upper limit based on the capacity of the server or machine being expanded.

Vertical scaling benefits

- **No changes have to be made to the application code** and no additional servers need to be added; you just make the server you have more powerful or downsize again.
- **Less complex network** – when a single instance handles all the layers of your services, it will not have to synchronize and communicate with other machines to work. This may result in faster responses.
- **Less complicated maintenance** – the maintenance is easier and less complex because of the number of instances you will need to manage.

Vertical scaling disadvantages

- **A maintenance window with downtime is required** – unless you have a backup server that can handle operations and requests, you will need some considerable downtime to upgrade your machine.
- **Single point of failure** – having all your operations on a single server increases the risk of losing all your data if a hardware or software failure were to occur.
- **Upgrade limitations** – there is a limitation to how much you can upgrade a machine/instance.

Horizontal scaling vs. vertical scaling

In the cloud, you will usually use both of these methods, but horizontal scaling is usually considered a long-term solution, while vertical scaling is usually considered a short-term solution. The reason for this distinction is that you can usually add as many servers to the infrastructure as you need, but sometimes hardware upgrades are just not possible anymore.

Both horizontal and vertical scaling have their benefits and limitations. Here are some factors to consider:

- **Upgradability and flexibility** – if you run your application layer on separate machines (horizontally scaled), they are easier to decouple and upgrade without

- **Worldwide distribution** – if you plan to have national or global customers, it is unreasonable to expect them to access your services from one location. In this case, you need to scale resources horizontally.
- **Reliability and availability** – horizontal scaling can provide you with a more reliable system. It increases redundancy and ensures that you are not dependent on one machine.
- **Performance** – sometimes it's better to leave the application as is and upgrade the hardware to meet demand (vertically scale). Horizontal scaling may require you to rewrite code, which can add complexity.

5.4 Virtual Machines, Ethernet and Switches

What is a virtual machine?

A **Virtual Machine** (VM) is a compute resource that uses software instead of a physical computer to run programs and deploy apps. One or more virtual "guest" machines run on a physical "host" machine. Each virtual machine runs its own operating system and functions separately from the other VMs, even when they are all running on the same host. This means that, for example, a virtual MacOS virtual machine can run on physical PC.

Virtual machine technology is used for many use cases across on-premises and cloud environments. More recently, public cloud services are using virtual machines to provide virtual application resources to multiple users at once, for even more cost efficient and flexible compute.

What are virtual machines used for?

Virtual machines (VMs) allow a business to run an operating system that behaves like a completely separate computer in an app window on a desktop. VMs may be deployed to accommodate different levels of processing power needs, to run software that requires a different operating system, or to test applications in a safe, sandboxed environment.

Virtual machines have historically been used for server virtualization, which enables IT teams to consolidate their computing resources and improve efficiency. Additionally, virtual machines can perform specific tasks considered too risky to carry out in a host environment, such as accessing virus-infected data or testing operating systems. Since the virtual machine is separated from the rest of the system, the software inside the virtual machine cannot tamper with the host computer.

How do virtual machines work?

The virtual machine runs as a process in an application window, similar to any other application, on the operating system of the physical machine. Key files that make up a virtual

Advantages of virtual machines

Virtual machines are easy to manage and maintain, and they offer several advantages over physical machines:

- VMs can run multiple operating system environments on a single physical computer, saving physical space, time and management costs.
- Virtual machines support legacy applications, reducing the cost of migrating to a new operating system. For example, a Linux virtual machine running a distribution of Linux as the guest operating system can exist on a host server that is running a non-Linux operating system, such as Windows.
- VMs can also provide integrated disaster recovery and application provisioning options.

Disadvantages of virtual machines

While virtual machines have several advantages over physical machines, there are also some potential disadvantages:

- Running multiple virtual machines on one physical machine can result in unstable performance if infrastructure requirements are not met.
- Virtual machines are less efficient and run slower than a full physical computer. Most enterprises use a combination of physical and virtual infrastructure to balance the corresponding advantages and disadvantages.

What is an Ethernet Switch?

Ethernet switching connects wired devices such as computers, laptops, routers, servers, and printers to a local area network (LAN). Multiple Ethernet switch ports allow for faster connectivity and smoother access across many devices at once.

An Ethernet switch creates networks and uses multiple ports to communicate between devices in the LAN. Ethernet switches differ from routers, which connect networks and use only a single LAN and WAN port. A full wired and wireless corporate infrastructure provides wired connectivity and Wi-Fi for wireless connectivity.

Hubs are similar to Ethernet switches in that connected devices on the LAN will be wired to them, using multiple ports. The big difference is that hubs share bandwidth equally among ports, while Ethernet switches can devote more bandwidth to certain ports without degrading network performance. When many devices are active on a network, Ethernet switching provides more robust performance.

Routers connect networks to other networks, most commonly connecting LANs to wide area networks (WANs). Routers are usually placed at the gateway between networks and route data packets along the network.

Most corporate networks use combinations of switches, routers, and hubs, and wired and wireless technology.

What Ethernet Switches Can Do For Your Network

Ethernet switches provide many advantages when correctly installed, integrated, and managed. These include:

1. Reduction of network downtime
2. Improved network performance and increased available bandwidth on the network
3. Relieving strain on individual computing devices
4. Protecting the overall corporate network with more robust security
5. Lower IT capex and opex costs thanks to remote management and consolidated wiring
6. Right-sizing IT infrastructure and planning for future expansion using modular switches

Most corporate networks support a combination of wired and wireless technologies, including Ethernet switching as part of the wired infrastructure. Dozens of devices can connect to a network using an Ethernet switch, and administrators can monitor traffic, control communications among machines, securely manage user access, and rapidly troubleshoot.

The switches come in a wide variety of options, meaning organizations can almost always find a solution right-sized for their network. These range from basic unmanaged network switches offering plug-and-play connectivity, to feature-rich Gigabit Ethernet switches that perform at higher speeds than wireless options.

How Ethernet Switches Work: Terms and Functionality

Frames are sequences of information, travel over Ethernet networks to move data between computers. An Ethernet frame includes a destination address, which is where the data is traveling to, and a source address, which is the location of the device sending the frame. In a standard seven-layer Open Systems Interconnection (OSI) model for computer networking, frames are part of Layer 2, also known as the data-link layer. These are sometimes known as “link layer devices” or “Layer 2 switches.”

Transparent Bridging is the most popular and common form of bridging, crucial to Ethernet switch functionality. Using transparent bridging, a switch automatically begins working without requiring any configuration on a switch or changes to the computers in the network (i.e. the operation of the switch is transparent).

Address Learning -- Ethernet switches control how frames are transmitted between switch ports, making decisions on how traffic is forwarded based on 48-bit media access control (MAC) addresses that are used in LAN standards. An Ethernet switch can learn which devices are on which segments of the network using the source addresses of the frames it receives.

Every port on a switch has a unique MAC address, and as frames are received on ports, the software in the switch looks at the source address and adds it to a table of addresses

reachable on which ports.) This table is also known as a forwarding database, which is used by the switch to make decisions on how to filter traffic to reach certain destinations. That the Ethernet switch can "learn" in this manner makes it possible for network administrators to add new connected endpoints to the network without having to manually configure the switch or the endpoints.

Traffic Filtering -- Once a switch has built a database of addresses, it can smoothly select how it filters and forwards traffic. As it learns addresses, a switch checks frames and makes decisions based on the destination address in the frame. Switches can also isolate traffic to only those segments needed to receive frames from senders, ensuring that traffic does not unnecessarily flow to other ports.

Multicast Traffic -- LANs are not only able to transmit frames to single addresses, but also capable of sending frames to multicast addresses, which are received by groups of endpoint destinations. Broadcast addresses are a specific form of multicast address; they group all of the endpoint destinations in the LAN. Multicasts and broadcasts are commonly used for functions such as dynamic address assignment, or sending data in multimedia applications to multiple users on a network at once, such as in online gaming. (Streaming applications such as video, which send high rates of multicast data and generate a lot of traffic, can hog network bandwidth.

5.5 Docker Container

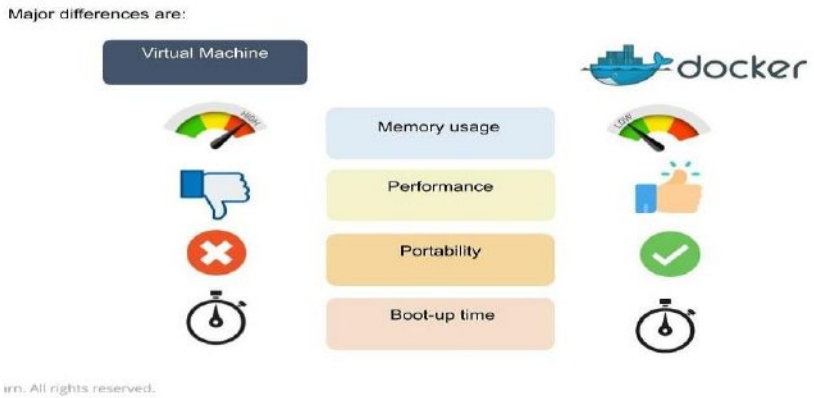
Docker is a container management service. The keywords of Docker are **develop, ship** and **run** anywhere. The whole idea of Docker is for developers to easily develop applications, ship them into containers which can then be deployed anywhere.

The initial release of Docker was in March 2013 and since then, it has become the buzzword for modern world development, especially in the face of Agile-based projects.

Features of Docker

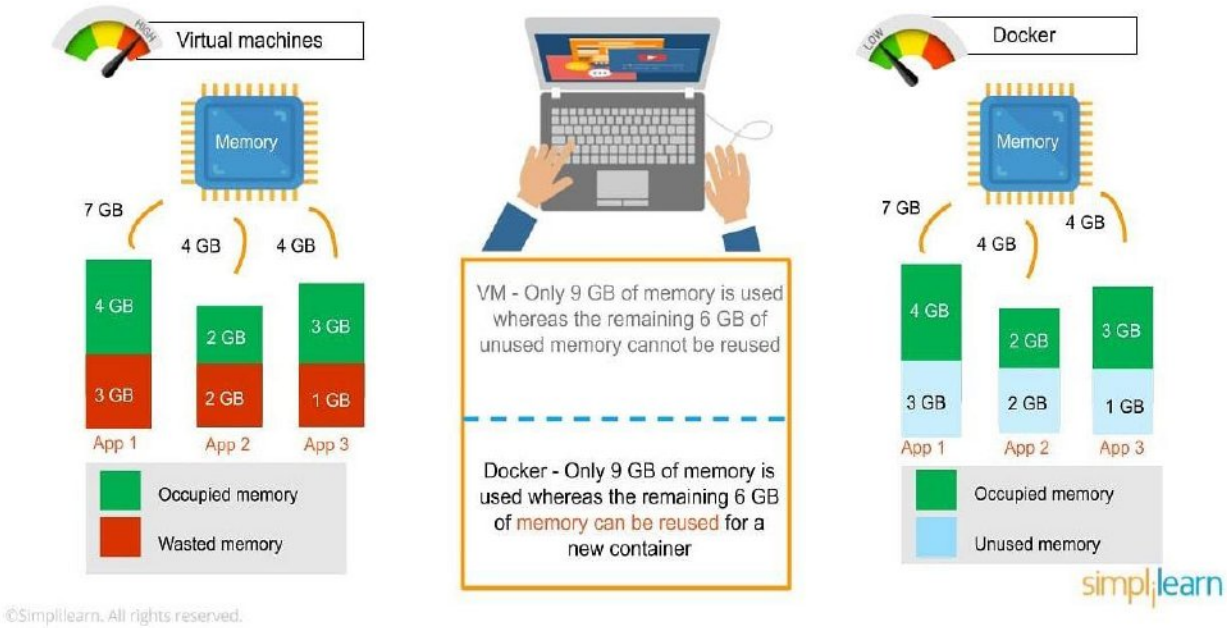
- Docker has the ability to reduce the size of development by providing a smaller footprint of the operating system via containers.
- With containers, it becomes easier for teams across different units, such as development, QA and Operations to work seamlessly across applications.
- You can deploy Docker containers anywhere, on any physical and virtual machines and even on the cloud.
- Since Docker containers are pretty lightweight, they are very easily scalable.

Docker vs Virtual Machines



In the image, you'll notice some major differences, including:

- The virtual environment has a hypervisor layer, whereas Docker has a Docker engine layer.
- There are additional layers of libraries within the virtual machine, each of which compounds and creates very significant differences between a Docker environment and a virtual machine environment.
- With a virtual machine, the memory usage is very high, whereas, in a Docker environment, memory usage is very low.
- In terms of performance, when you start building out a virtual machine, particularly when you have more than one virtual machine on a server, the performance becomes poorer. With Docker, the performance is always high because of the single Docker engine.
- In terms of portability, virtual machines just are not ideal. They're still dependent on the host operating system, and a lot of problems can happen when you use virtual machines for portability. In contrast, Docker was designed for portability. You can actually build solutions in a Docker container, and the solution is guaranteed to work as you have built it no matter where it's hosted.
- The boot-up time for a virtual machine is fairly slow in comparison to the boot-up time for a Docker environment, in which boot-up is almost instantaneous.



- One of the other challenges of using a virtual machine is that if you have unused memory within the environment, you cannot reallocate it. If you set up an environment that has 9 gigabytes of memory, and 6 of those gigabytes are free, you cannot do anything with that unused memory. With Docker, if you have free memory, you can reallocate and reuse it across other containers used within the Docker environment.
- Running multiples of them in a single environment can lead to instability and performance issues. Docker, on the other hand, is designed to run multiple containers in the same environment—it actually gets better with more containers run in that hosted single Docker engine.
- Virtual machines have portability issues; the software can work on one machine, but if you move that virtual machine to another machine, suddenly some of the software won't work, because some dependencies will not be inherited correctly. Docker is designed to be able to run across multiple environments and to be deployed easily across systems.
- The boot-up time for a virtual machine is about a few minutes, in contrast to the milliseconds it takes for a Docker environment to boot up.

5.6 Kubernetes

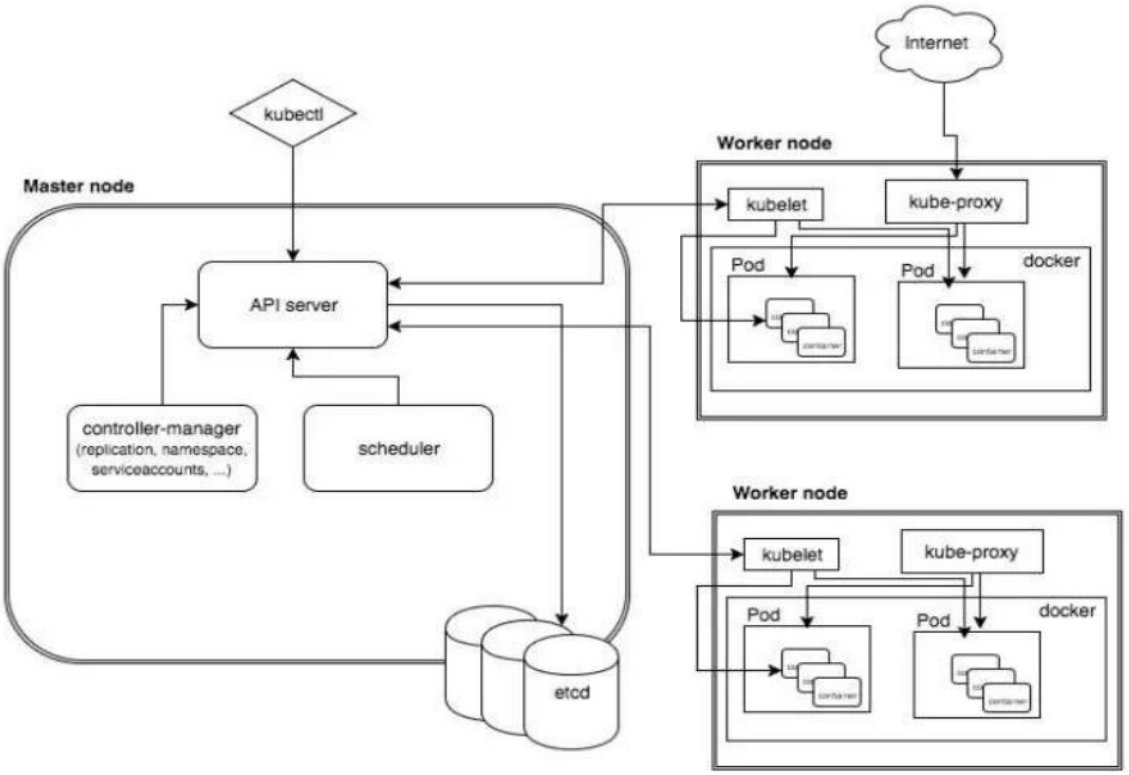
What is Kubernetes?

Kubernetes is an open source orchestration tool developed by Google for managing microservices or containerized applications across a distributed cluster of nodes. Kubernetes provides highly resilient infrastructure with zero downtime deployment capabilities, automatic rollback, scaling, and self-healing of containers (which consists of auto-placement, auto-restart, auto-replication , and scaling of containers on the basis of CPU usage).

The main objective of Kubernetes is to hide the complexity of managing a fleet of containers by providing REST APIs for the required functionalities. Kubernetes is portable in nature, meaning it can run on various public or private cloud platforms such as AWS, Azure, OpenStack, or Apache Mesos. It can also run on bare metal machines.

Kubernetes Components and Architecture

Kubernetes follows a client-server architecture. It's possible to have a multi-master setup (for high availability), but by default there is a single master server which acts as a controlling node and point of contact. The master server consists of various components including a kube-apiserver, an etcd storage, a kube-controller-manager, a cloud-controller-manager, a kube-scheduler, and a DNS server for Kubernetes services. Node components include kubelet and kube-proxy on top of Docker.



A Kubernetes control plane is the control plane for a Kubernetes cluster. Its components include:

- **kube-apiserver**. As its name suggests the API server exposes the Kubernetes API, which is communications central. External communications via command line interface (CLI) or other user interfaces (UI) pass to the kube-apiserver, and all control planes to node communications also goes through the API server.
- **etcd**: The key value store where all data relating to the cluster is stored. etcd is highly available and consistent since all access to etcd is through the API server. Information in etcd is generally formatted in human-readable YAML (which stands for the recursive "YAML Ain't Markup Language").

- **kube-scheduler**: When a new Pod is created, this component assigns it to a node for execution based on resource requirements, policies, and 'affinity' specifications regarding geolocation and interference with other workloads.
- **kube-controller-manager**: Although a Kubernetes cluster has several controller functions, they are all compiled into a single binary known as kube-controller-manager.

What is Kubernetes node architecture?

Nodes are the machines, either VMs or physical servers, where Kubernetes place Pods to execute. Node components include:

kubelet: Every node has an agent called kubelet. It ensures that the container described in PodSpecs are up and running properly.

kube-proxy: A network proxy on each node that maintains network nodes which allows for the communication from Pods to network sessions, whether inside or outside the cluster, using operating system (OS) packet filtering if available.

container runtime: Software responsible for running the containerized applications. Although Docker is the most popular, Kubernetes supports any runtime that adheres to the Kubernetes CRI (Container Runtime Interface).

Kubernetes Concepts

Making use of Kubernetes requires understanding the different abstractions it uses to represent the state of the system, such as services, pods, volumes, namespaces, and deployments.

- **Pod** – generally refers to one or more containers that should be controlled as a single application. A pod encapsulates application containers, storage resources, a unique network ID and other configuration on how to run the containers.
- **Service** – pods are volatile, that is Kubernetes does not guarantee a given physical pod will be kept alive (for instance, the replication controller might kill and start a new set of pods). Instead, a service represents a logical set of pods and acts as a gateway, allowing (client) pods to send requests to the service without needing to keep track of which physical pods actually make up the service.
- **Volume** – similar to a container volume in Docker, but a Kubernetes volume applies to a whole pod and is mounted on all containers in the pod. Kubernetes guarantees data is

preserved across container restarts. The volume will be removed only when the pod gets destroyed. Also, a pod can have multiple volumes (possibly of different types) associated.

- **Namespace** – a virtual cluster (a single physical cluster can run multiple virtual ones) intended for environments with many users spread across multiple teams or projects, for isolation of concerns. Resources inside a namespace must be unique and cannot access resources in a different namespace. Also, a namespace can be allocated a resource quota to avoid consuming more than its share of the physical cluster's overall resources.
- **Deployment** – describes the desired state of a pod or a replica set, in a yaml file. The deployment controller then gradually updates the environment (for example, creating or deleting replicas) until the current state matches the desired state specified in the deployment file. For example, if the yaml file defines 2 replicas for a pod but only one is currently running, an extra one will get created. Note that replicas managed via a deployment should not be manipulated directly, only via new deployments.

➔ END ➔