



# FULL Stack WEB Development BOOK

Full Stack Web development (Anna University)

# FULL STACK WEB DEVELOPMENT

For II Semester MCA Students



As Per Latest Syllabus of Anna University

## UNIT I INTRODUCTION TO CSS and JAVASCRIPT

9

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

### Introduction to Web: Server

A **web server** accepts and fulfills requests from clients for static content (i.e., HTML pages, files, images, and videos) from a website. Web servers handle HTTP requests and responses *only*.

An **application server** exposes *business logic* to the clients, which generates dynamic content. It is a software framework that transforms data to provide the specialized functionality offered by a business, service, or application. Application servers enhance the interactive parts of a website that can appear differently depending on the context of the request.

The illustration below highlights the difference in their architecture:

The columns below summarize the key differences between the two types of servers:

#### Web Server

- ♣ Deliver static content.
- ♣ Content is delivered using the HTTP protocol only.
- ♣ Serves only web-based applications.
- ♣ No support for multi-threading.
- ♣ Facilitates web traffic that is not very resource intensive.

#### Application Server

- ♣ Delivers dynamic content.
- ♣ Provides business logic to application programs using several protocols (including HTTP).
- ♣ Can serve web and enterprise-based applications.
- ♣ Uses multi-threading to support multiple requests in parallel.
- ♣ Facilitates longer running processes that are very resource-intensive.

#### HTTP:-

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. This is the foundation for data communication for the World Wide Web (i.e. internet) since 1990. HTTP is a generic and stateless protocol which can be used for other purposes as well using extensions of its request methods, error codes, and headers.

Basically, HTTP is a TCP/IP based communication protocol, that is used to deliver data (HTML files, image files, query results, etc.) on the World Wide Web. The default port is TCP 80, but other ports can be used as well. It provides a standardized way for computers to communicate with each other. HTTP specification specifies how clients' request data will be constructed and sent to the server, and how the servers respond to these requests.

## Basic Features

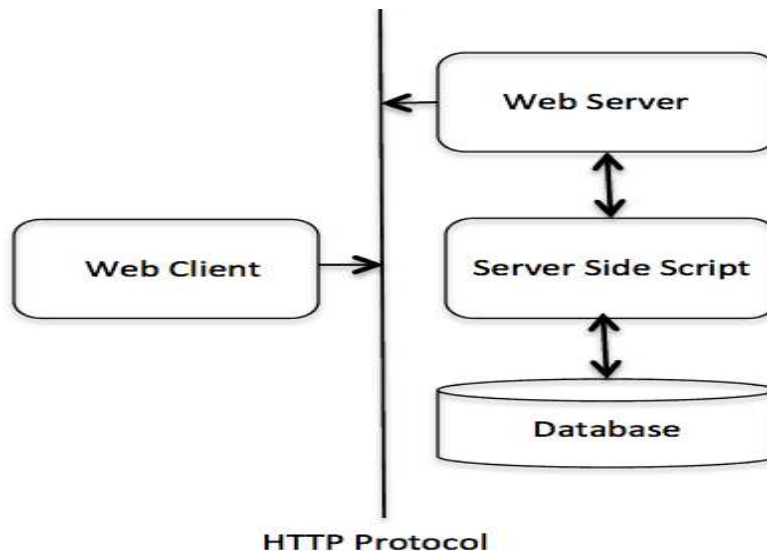
There are three basic features that make HTTP a simple but powerful protocol:

- ❖ **HTTP is connectionless:** The HTTP client, i.e., a browser initiates an HTTP request and after a request is made, the client waits for the response. The server processes the request and sends a response back after which client disconnect the connection. So client and server knows about each other during current request and response only. Further requests are made on new connection like client and server are new to each other.
- ❖ **HTTP is media independent:** It means, any type of data can be sent by HTTP as long as both the client and the server know how to handle the data content. It is required for the client as well as the server to specify the content type using appropriate MIME-type.
- ❖ **HTTP is stateless:** As mentioned above, HTTP is connectionless and it is a direct result of HTTP being a stateless protocol. The server and client are aware of each other only during a current request. Afterwards, both of them forget about each other. Due to this nature of the protocol, neither the client nor the browser can retain information between different requests across the web pages.

HTTP/1.0 uses a new connection for each request/response exchange, where as HTTP/1.1 connection may be used for one or more request/response exchanges.

## Basic Architecture

The following diagram shows a very basic architecture of a web application and depicts where HTTP sits:



The HTTP protocol is a request/response protocol based on the client/server based architecture where web browsers, robots and search engines, etc. act like HTTP clients, and the Web server acts as a server.

### Client

The HTTP client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a TCP/IP connection.

### Server

The HTTP server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity meta information, and possible entity-body content.

### HTML Documents Structure

HTML used predefined tags and attributes to tell browser how to display content, means in which format, style, font-size, images to display. HTML is a case insensitive language. **Case insensitive** means there is no difference in upper case and lower case (capital and small letters) both treated as same, for example 'D' and 'd' both are same here. There are generally two types of tags in HTML:

1. **Paired Tags:** These tags come in pairs. That is they have both opening(< >) and closing(</ >) tags.
2. **Empty Tags:** These tags do not require to be closed.

Below is an example of (<b>) tag in HTML, which tells the browser to bold the text inside it.

**Tags and attributes:** Tags are individuals of html structure, we have to open and close any tag with forwards slash like this <h1> </h1>. There are some variations with tags, some of them are self-closing tags which aren't required to close and some are empty tags where we can add any attributes. Attributes are additional properties of html tags which define the property of any html tags. i.e. width, height, controls, loops, input and autoplay. These attributes also help us to store information in meta tags by using name, content, type attributes.

## HTML Documents structured mentioned below:

### Structure of an HTML Document

An HTML Document is mainly divided into two parts:

- ❖ **HEAD:** This contains the information about the HTML document. For Example, Title of the page, version of HTML, Meta Data etc.
- ❖ **BODY:** This contains everything you want to display on the Web Page.

```
<!DOCTYPE html>
<html>

<head>
  <title>Page Title</title>
</head>

<body>
  <h2>Heading Content</h2>
  <p>Paragraph Content</p>
</body>

</html>
```

## HTML Document Structure

Let us now have a look at the basic structure of HTML. That is the code that is a must for every webpage to have:

```
<!DOCTYPE html>
<!-- Defines types of documents : Html 5.0 -->
```

```

<html lang="en">
  <!-- DEfines lanuages of content : English -->
  <head>
    <!-- Information about website and creator -->
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <!-- Defines the compatiablity of version with browser -->
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <!-- for make website responsive -->
    <meta name="author" content="Mr.X">
    <meta name="Linkedin profile" content="WWW.linkedin.com/Mr.X_123" >
    <!-- To give information about auther or owner -->
    <meta name="description " content="A better place to learn computer science">
    <!-- to explain about website in few words -->
    <title>GeeksforGeeks</title>
    <!-- Name of website or content to display -->
  </head>
  <body>
    <!-- Main content of website -->
    <h1>GeeksforGeeks</h1>

    <p>A computer science portal for geeks</p>

  </body>
</html>

```

(<https://onecompiler.com/html/3y6644268>)

Every Webpage must contain this code. Below is the complete explanation of each of the tag used in the above piece of HTML code:

**<!DOCTYPE html>:** This tag is used to tells the HTML version. This currently tells that the version is HTML 5.0

**<html> </html> :** <html> is a root element of html. It's a biggest and main element in complete html language, all the tags , elements and attributes inclosed in it or we can say wrap init , which is used to structure a web page. <html> tag is parent tag of <head> and <body> tag , other tags inclosed within <head > and <body>. In <html > tag we use "lang" attributes to define languages of html page such as <html lang="en"> here en represents english language. some of them are : es = spansih , zh-Hans = chinese, fr= french and el= greek etc.

**<head>:** Head tag contains metadata, title, page CSS etc. Data stored in <head> tag is not displayed to user , it just write for refrence purpose aur as a water mark of owner.

**Note:** for better understanding refer above code of html.

**<title>** = to store website name or content to be displayed.

**<link>** = To add/ link css( cascading syle sheet) file.

**<meta>** = 1. to store data about website, organisation , creator/ owner  
 2. for responsive website via attributes  
 3. to tell compatiability of html with browser

**<script>** = to add javascript file.

**<body>:** Body tag is used to enclose all the data which a web page has from texts to links. All the content that you see rendered in the browser is contained within this element. Following tags and elements used in body.

1. <h1> ,<h2> ,<h3> to <h6>
2. <p>
3. <div> and <span>
4. <b> , <i> and <u>
5. <li> ,<ul> and <ol>.
6. <img> , <audio> , <video> and <iframe>
7. <table> <th> , <thead> and <tr>.
8. <form>
9. <label> and <input> others.....

### **Working with Text and Images with CSS**

```
<!DOCTYPE html>
<html>
<title>Online CSS Editor</title>
<head>
<style>
div{
  width:200px;
  height:125px;
  padding:10px;
  background-color:red;
  border:1px solid black;
}
#box{
  transform:rotate(30deg);
  -ms-transform:rotate(30deg); /* IE 9 */
  -moz-transform:rotate(30deg); /* Firefox */
  -webkit-transform:rotate(30deg); /* Safari and Chrome */
  -o-transform:rotate(30deg); /* Opera */
  background-color:yellow;
}
</style>
</head>
<body>
<div>Real Time CSS Editor</div>
<div id="box">Hello, World!</div>
</body>
</html>
```

### **Output:-**

Real Time CSS Editor  
Hello, World!



## **Rounded Images:-**

### **Example:1 Border Radius**

```
<!DOCTYPE html>
<html>
<head>
<style>
img {
  border-radius: 8px;
}
</style>
</head>
<body>

<h2>Rounded Image</h2>

<p>Use the border-radius property to create rounded images:</p>



</body>
</html>
```

[[https://www.w3schools.com/css/tryit.asp?filename=trycss\\_ex\\_images\\_round](https://www.w3schools.com/css/tryit.asp?filename=trycss_ex_images_round)]

### **Example:2 [Circle image]**

```
<!DOCTYPE html>
<html>
<head>
<style>
img {
  border-radius: 50%;
}
</style>
</head>
<body>

<h2>circle Image</h2>

<p>Use the border-radius property to create circled images:</p>
```

```

```

```
</body>
```

```
</html>
```

## Thumbnail Images:-

### Example: 3 [Thumbnail Image]

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
<style>
```

```
img {
```

```
    border: 1px solid #ddd;
```

```
    border-radius: 50px;
```

```
    padding: 20px;
```

```
    width: 550px;
```

```
}
```

```
</style>
```

```
</head>
```

```
<body>
```

```
<h2>Thumbnail Image</h2>
```

```
<p>Use the border property to create thumbnail images:</p>
```

```

```

```
</body>
```

```
</html>
```

## Responsive Images:-

### Example: 4 [Responsive Image]

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
<style>
```

```
img {
```

```
    max-width: 200%;
```

```
    height: auto;
```

```
}
```

```
</style>
</head>
<body>

<h2>Responsive Image</h2>

<p>Responsive images will automatically adjust to fit the size of the screen.</p>
<p>Resize the browser window to see the effect:</p>



</body>
</html>
```

[[https://www.w3schools.com/css/tryit.asp?filename=trycss\\_ex\\_images\\_responsive](https://www.w3schools.com/css/tryit.asp?filename=trycss_ex_images_responsive)]

:-

#### **Example:-**

**<!DOCTYPE html> Transparent Image**

```
<html>
<head>
<style>
img {
  opacity: 1.4;
}
</style>
</head>
<body>
```

**<h1>Image Transparency</h1>**

**<p>The opacity property specifies the transparency of an element. The lower the value, the more transparent:</p>**

```
<p>Image with 50% opacity:</p>

</body>
</html>
```

[[https://www.w3schools.com/css/tryit.asp?filename=trycss\\_ex\\_images\\_opacity](https://www.w3schools.com/css/tryit.asp?filename=trycss_ex_images_opacity)]

## Image Text

### Example:-

```
<!DOCTYPE html>
<html>
<head>
<style>
.container {
  position: relative;
}

.topleft {
  position: absolute;
  top: 8px;
  left: 16px;
  font-size: 20px;
}

img {
  width: 500%;
  height: auto;
  opacity: 1.9;
}
</style>
</head>
<body>

<h2>Image Text</h2>

<p>Add some text to an image in the top left corner:</p>

<div class="container">
  
  <div class="topleft">hai</div>
</div>

</body>
</html>
```

[[https://www.w3schools.com/css/tryit.asp?filename=trycss\\_ex\\_images\\_text\\_top\\_left](https://www.w3schools.com/css/tryit.asp?filename=trycss_ex_images_text_top_left)]

### Image Filters:-

**Example:-**

```
<!DOCTYPE html>
<html>
<head>
<style>
body {
    background-color:white;
}
img {
    width: 33%;
    height: auto;
    float: left;
    max-width: 235px;
}

.blur {filter: blur(4px);}
.brightness {filter: brightness(250%);}
.contrast {filter: contrast(180%);}
.grayscale {filter: grayscale(100%);}
.huerotate {filter: hue-rotate(180deg);}
.invert {filter: invert(100%);}
.opacity {filter: opacity(50%);}
.saturate {filter: saturate(7);}
.sepia {filter: sepia(100%);}
.shadow {filter: drop-shadow(8px 8px 10px green);}
</style>
</head>
<body>
```

<h2>Image Filters</h2>

<p><strong>Note:</strong> The filter property is not supported in Internet Explorer or Edge 12.</p>

```











```

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

[[https://www.w3schools.com/css/tryit.asp?filename=trycss\\_ex\\_images\\_filters](https://www.w3schools.com/css/tryit.asp?filename=trycss_ex_images_filters)]

### **Fade in Overlay:-**

#### **Example 1:-**

```
<html>
<head>
<style>
.container {
  position: relative;
  width: 50%;
}

.image {
  display: block;
  width: 100%;
  height: auto;
}

.overlay {
  position: absolute;
  top: 0;
  bottom: 0;
  left: 0;
  right: 0;
  height: 100%;
  width: 100%;
  opacity: 0;
  transition: .5s ease;
  background-color: #008CBA;
}

.container:hover .overlay {
  opacity: 1;
}

.text {
  color: white;
  font-size: 20px;
  position: absolute;
  top: 50%;
  left: 50%;
```

```
    transform: translate(-50%, -50%);
    -ms-transform: translate(-50%, -50%);
  }
</style>
</head>
<body>
```

```
<h2>Fade in Overlay</h2>
```

```
<div class="container">
  
  <div class="overlay">
    <div class="text">Hai</div>
  </div>
</div>

</body>
</html>
```

[[https://www.w3schools.com/css/tryit.asp?filename=trycss\\_css\\_image\\_overlay\\_fade](https://www.w3schools.com/css/tryit.asp?filename=trycss_css_image_overlay_fade)]

### **Example:2**

```
<!DOCTYPE html>
<html>
<head>
<style>
.container {
  position: relative;
  width: 50%;
}

.image {
  display: block;
  width: 100%;
  height: auto;
}

.overlay {
  position: absolute;
  bottom: 0;
  left: 0;
  right: 0;
  background-color: #008CBA;
  overflow: hidden;
  width: 100%;
  height: 0;
```

```

    transition: .5s ease;
}

.container:hover .overlay {
    height: 100%;
}

.text {
    white-space: nowrap;
    color: white;
    font-size: 20px;
    position: absolute;
    overflow: hidden;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
    -ms-transform: translate(-50%, -50%);
}
</style>
</head>
<body>

```

<h2>Slide in Overlay from the Bottom</h2>

```

<div class="container">
  
  <div class="overlay">
    <div class="text">Hello World</div>
  </div>
</div>

</body>
</html>

```

[[https://www.w3schools.com/css/tryit.asp?filename=trycss\\_css\\_image\\_overlay\\_slidebottom](https://www.w3schools.com/css/tryit.asp?filename=trycss_css_image_overlay_slidebottom)]

## Responsive Image Gallery

```

<!DOCTYPE html>
<html>
<head>
<style>
div.gallery {
    border: 1px solid #ccc;
}

```



```

div.gallery:hover {
  border: 1px solid #777;
}
div.gallery img {
  width: 100%;
  height: auto;
}
div.desc {
  padding: 15px;
  text-align: center;
}
* {
  box-sizing: border-box;
}
.responsive {
  padding: 0 6px;
  float: left;
  width: 24.99999%;
}
@media only screen and (max-width: 700px) {
  .responsive {
    width: 49.99999%;
    margin: 6px 0;
  }
}
@media only screen and (max-width: 500px) {
  .responsive {
    width: 100%;
  }
}
.clearfix:after {
  content: "";
  display: table;
  clear: both;
}
</style>
</head>
<body>
<h2>Responsive Image Gallery</h2>
<h4>Resize the browser window to see the effect.</h4>

```

```

<div class="responsive">
  <div class="gallery">
    <a target="_blank" href="img_5terre.jpg">
      
    </a>
    <div class="desc">Add a description of the image here</div>
  </div>
</div>
<div class="responsive">
  <div class="gallery">
    <a target="_blank" href="img_forest.jpg">
      
    </a>
    <div class="desc">Add a description of the image here</div>
  </div>
</div>
<div class="responsive">
  <div class="gallery">
    <a target="_blank" href="img_lights.jpg">
      
    </a>
    <div class="desc">Add a description of the image here</div>
  </div>
</div>
<div class="responsive">
  <div class="gallery">
    <a target="_blank" href="img_mountains.jpg">
      
    </a>
    <div class="desc">Add a description of the image here</div>
  </div>
</div>
<div class="clearfix"></div>
<div style="padding:6px;">
  <p>This example use media queries to re-arrange the images on different screen sizes: for screens larger than 700px wide, it will show four images side by side, for screens smaller than 700px, it will show two images side by side. For screens smaller than 500px, the images will stack vertically (100%).</p>
  <p>You will learn more about media queries and responsive web design later in our CSS Tutorial.</p>

```

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

## CSS Selectors

CSS selectors are used to "find" (or select) the HTML elements you want to style.

We can divide CSS selectors into five categories:

- Simple selectors (select elements based on name, id, class)
- Combinator selectors (select elements based on a specific relationship between them)
- Pseudo-class selectors (select elements based on a certain state)
- Pseudo-elements selectors (select and style a part of an element)
- Attribute selectors (select elements based on an attribute or attribute value)

### The CSS element Selector:-

The element selector selects HTML elements based on the element name.

### Example:-

Here, all <p> elements on the page will be center-aligned, with a red text color:

```
<!DOCTYPE html>
<html>
<head>
<style>
p {
    text-align: center;
    color: red;
}
</style>
</head>
<body>
<p>Every paragraph will be affected by the style.</p>
<p id="para1">Me too!</p>
<p>And me!</p>
</body>
</html>
```

[[https://www.w3schools.com/css/tryit.asp?filename=trycss\\_syntax\\_element](https://www.w3schools.com/css/tryit.asp?filename=trycss_syntax_element)]

### The CSS id Selector:-

- ♣ The id selector uses the id attribute of an HTML element to select a specific element.

- ♣ The id of an element is unique within a page, so the id selector is used to select one unique element!
- ♣ To select an element with a specific id, write a hash (#) character, followed by the id of the element.

```
<!DOCTYPE html>
<html>
<head>
<style>
#para1 {
  text-align: center;
  color: red;
}
</style>
</head>
<body>

<p id="para1">Hello World!</p>
<p>This paragraph is not affected by the style.</p>

</body>
</html>
```

### The CSS class Selector:-

The class selector selects HTML elements with a specific class attribute.

To select elements with a specific class, write a period (.) character, followed by the class name.

```
<!DOCTYPE html>
<html>
<head>
<style>
.center {
  text-align: center;
  color: red;
}
</style>
</head>
<body>

<h1 class="center">Red and center-aligned heading</h1>
<p class="center">Red and center-aligned paragraph.</p>

</body>
```

```
</html>
```

### **The CSS Universal Selector:-**

```
<!DOCTYPE html>
<html>
<head>
<style>
* {
  text-align: left;
  color: green;
}
</style>
</head>
<body>

<h1>Hello world!</h1>

<p>Every element on the page will be affected by the style.</p>
<p id="para1">Me too!</p>
<p>And me!</p>

</body>
</html>
```

[[https://www.w3schools.com/css/tryit.asp?filename=trycss\\_syntax\\_universal](https://www.w3schools.com/css/tryit.asp?filename=trycss_syntax_universal)]

### **The CSS Grouping Selector:-**

The grouping selector selects all the HTML elements with the same style definitions.

Look at the following CSS code (the h1, h2, and p elements have the same style definitions):

```
<!DOCTYPE html>
<html>
<head>
<style>
h1, h2, p {
  text-align: center;
  color: red;
}
</style>
</head>
<body>

<h1>Hello World!</h1>
```

```
<h2>Smaller heading!</h2>
<p>This is a paragraph.</p>

</body>
</html>
[https://www.w3schools.com/css/tryit.asp?filename=trycss_grouping]
```

## CSS Flexbox

To start using the Flexbox model, you need to first define a flex container.

```
<!DOCTYPE html>
<html>
<head>
<style>
.flex-container {
  display: flex;
  background-color: DodgerBlue;
}
.flex-container > div {
  background-color: #f1f1f1;
  margin: 10px;
  padding: 20px;
  font-size: 30px;
}
</style>
</head>
<body>
<h1>Create a Flex Container</h1>
<div class="flex-container">
  <div>1</div>
  <div>2</div>
  <div>3</div>
</div>
<p>A Flexible Layout must have a parent element with the <em>display</em> property set to
<em>flex</em>.</p>
<p>Direct child element(s) of the flexible container automatically becomes flexible items.</p>
</body>
</html>
[https://www.w3schools.com/css/tryit.asp?filename=trycss3_flexbox]
```

## The flex-direction Property:-

The `flex-direction` property defines in which direction the container wants to stack the flex items.

```
<!DOCTYPE html>
<html>
<head>
<style>
.flex-container {
  display: flex;
  flex-direction: column;
  background-color: DodgerBlue;
}
.flex-container > div {
  background-color: #f1f1f1;
  width: 100px;
  margin: 10px;
  text-align: center;
  line-height: 75px;
  font-size: 30px;
}
</style>
</head>
<body>
<h1>The flex-direction Property</h1>
<p>The "flex-direction: column;" stacks the flex items vertically (from top to bottom):</p>
<div class="flex-container">
  <div>1</div>
  <div>2</div>
  <div>3</div>
</div>
</body>
</html>
[https://www.w3schools.com/css/tryit.asp?filename=trycss3_flexbox_flex-direction_column]
```

**The "flex-direction: row-reverse;" stacks the flex items horizontally (but from right to left):**

```
<!DOCTYPE html>
<html>
<head>
<style>
.flex-container {
  display: flex;
  flex-direction: row-reverse;
  background-color: DodgerBlue;
}

```

```

.flex-container > div {
  background-color: #f1f1f1;
  width: 100px;
  margin: 10px;
  text-align: center;
  line-height: 75px;
  font-size: 30px;
}
</style>
</head>
<body>

```

<h1>The flex-direction Property</h1>

<p>The "flex-direction: row-reverse;" stacks the flex items horizontally (but from right to left):</p>

```

<div class="flex-container">
  <div>1</div>
  <div>2</div>
  <div>3</div>
</div>

</body>
</html>

```

[[https://www.w3schools.com/css/tryit.asp?filename=trycss3\\_flexbox\\_flex-direction\\_row-reverse](https://www.w3schools.com/css/tryit.asp?filename=trycss3_flexbox_flex-direction_row-reverse)]

## The flex-wrap Property:-

The flex-wrap property specifies whether the flex items should wrap or not.

The examples below have 12 flex items, to better demonstrate the flex-wrap property.

```

<!DOCTYPE html>
<html>
<head>
<style>
.flex-container {
  display: flex;
  flex-wrap: wrap;
  background-color: DodgerBlue;
}

```



```

.flex-container > div {
  background-color: #f1f1f1;
  width: 100px;
  margin: 10px;
  text-align: center;
  line-height: 75px;
  font-size: 30px;
}
</style>
</head>
<body>
<h1>The flex-wrap Property</h1>
<p>The "flex-wrap: wrap;" specifies that the flex items will wrap if necessary:</p>
<div class="flex-container">
  <div>1</div>
  <div>2</div>
  <div>3</div>
  <div>4</div>
  <div>5</div>
  <div>6</div>
  <div>7</div>
  <div>8</div>
  <div>9</div>
  <div>10</div>
  <div>11</div>
  <div>12</div>
</div>
<p>Try resizing the browser window.</p>
</body>
</html>

```

[[https://www.w3schools.com/css/tryit.asp?filename=trycss3\\_flexbox\\_flex-wrap\\_wrap](https://www.w3schools.com/css/tryit.asp?filename=trycss3_flexbox_flex-wrap_wrap)]

## The flex-flow Property

The flex-flow property is a shorthand property for setting both the flex-direction and flex-wrap properties.

```

<!DOCTYPE html>
<html>
<head>
<style>
.flex-container {
  display: flex;
  justify-content: center;
  background-color: DodgerBlue;

```

```

}

.flex-container > div {
  background-color: #f1f1f1;
  width: 100px;
  margin: 10px;
  text-align: center;
  line-height: 75px;
  font-size: 30px;
}
</style>
</head>
<body>

```

<h1>The justify-content Property</h1>

<p>The "justify-content: center;" aligns the flex items at the center of the container:</p>

```

<div class="flex-container">
  <div>1</div>
  <div>2</div>
  <div>3</div>
</div>

</body>
</html>

```

### The align-items Property:-

```

<!DOCTYPE html>
<html>
<head>
<style>
.flex-container {
  display: flex;
  height: 200px;
  align-items: center;
  background-color: DodgerBlue;
}

.flex-container > div {
  background-color: #f1f1f1;
  width: 100px;
  margin: 10px;
  text-align: center;
  line-height: 75px;
}

```

```

    font-size: 30px;
}
</style>
</head>
<body>

```

<h1>The align-items Property</h1>

<p>The "align-items: center;" aligns the flex items in the middle of the container:</p>

```

<div class="flex-container">
  <div>1</div>
  <div>2</div>
  <div>3</div>
</div>

```

```

</body>
</html>

```

Property	Description
➤ align-content	➤ Modifies the behavior of the flex-wrap property. It is similar to align-items, but instead of aligning flex items, it aligns flex lines
➤ align-items	➤ Vertically aligns the flex items when the items do not use all available space on the cross-axis
➤ display	➤ Specifies the type of box used for an HTML element
➤ flex-direction	➤ Specifies the direction of the flexible items inside a flex container
➤ flex-flow	➤ A shorthand property for flex-direction and flex-wrap
➤ flex-wrap	➤ Specifies whether the flex items should wrap or not, if there is not enough room for them on one flex line
➤ justify-content	➤ Horizontally aligns the flex items when the items do not use all available space on the main-axis

## The CSS Flexbox Items Properties

The following table lists all the CSS Flexbox Items properties:

Property	Description
align-self	Specifies the alignment for a flex item (overrides the flex container's align-items property)

flex	A shorthand property for the flex-grow, flex-shrink, and the flex-basis properties
flex-basis	Specifies the initial length of a flex item
flex-grow	Specifies how much a flex item will grow relative to the rest of the flex items inside the same container
flex-shrink	Specifies how much a flex item will shrink relative to the rest of the flex items inside the same container
order	Specifies the order of the flex items inside the same container

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
<style>
```

```
.flex-container {
  display: flex;
  align-items: stretch;
  background-color: #f1f1f1;
}
```

```
.flex-container > div {
  background-color: DodgerBlue;
  color: white;
  margin: 10px;
  text-align: center;
  line-height: 75px;
  font-size: 30px;
}
```

```
</style>
```

```
</head>
```

```
<body>
```

```
<h1>The flex-grow Property</h1>
```

```
<p>Make the third flex item grow eight times faster than the other flex items:</p>
```

```
<div class="flex-container">
  <div style="flex-grow: 1">1</div>
  <div style="flex-grow: 1">2</div>
  <div style="flex-grow: 8">3</div>
</div>
```

```
</body>
```

```
</html>
```

[[https://www.w3schools.com/css/tryit.asp?filename=trycss3\\_flexbox\\_flex-grow](https://www.w3schools.com/css/tryit.asp?filename=trycss3_flexbox_flex-grow)]

### The flex-shrink Property:-

```
<!DOCTYPE html>
<html>
<head>
<style>
.flex-container {
  display: flex;
  align-items: stretch;
  background-color: #f1f1f1;
}

.flex-container>div {
  background-color: DodgerBlue;
  color: white;
  width: 100px;
  margin: 10px;
  text-align: center;
  line-height: 75px;
  font-size: 30px;
}
</style>
</head>
<body>

<h1>The flex-shrink Property</h1>

<p>Do not let the third flex item shrink as much as the other flex items:</p>

<div class="flex-container">
  <div>1</div>
  <div>2</div>
  <div style="flex-shrink: 0">3</div>
  <div>4</div>
  <div>5</div>
  <div>6</div>
  <div>7</div>
  <div>8</div>
  <div>9</div>
  <div>10</div>
</div>

</body>
```

</html>

[[https://www.w3schools.com/css/tryit.asp?filename=trycss3\\_flexbox\\_flex-shrink](https://www.w3schools.com/css/tryit.asp?filename=trycss3_flexbox_flex-shrink)]

## 1.6.JavaScript: Data Types and Variables - Functions – Events

JavaScript variables can hold different data types: numbers, strings, objects and more:

```
let length = 16;           // Number
let lastName = "Johnson";  // String
let x = {firstName:"John", lastName:"Doe"}; // Object
```

### The Concept of Data Types

In programming, data types is an important concept.

To be able to operate on variables, it is important to know something about the type.

#### Example

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript</h2>
<p>When adding a number and a string, JavaScript will treat the number as a string.</p>
<p id="demo"></p>
<script>
let x = 67 + "Volvo";
document.getElementById("demo").innerHTML = x;
</script>
</body>
</html>
```

#### Output:-

When adding a number and a string, JavaScript will treat the number as a string.

67Volvo

JavaScript evaluates expressions from left to right. Different sequences can produce different results:

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

```
<h2>JavaScript</h2>
<p>JavaScript evaluates expressions from left to right. Different sequences can produce different
results:</p>
<p id="demo"></p>
<script>
let x = 80 + 4 + "Volvo";
document.getElementById("demo").innerHTML = x;
</script>
</body>
</html>
Output:-
```

JavaScript evaluates expressions from left to right. Different sequences can produce different results:

84Volvo

### JavaScript Types are Dynamic

JavaScript has dynamic types. This means that the same variable can be used to hold different data types:

```
<h2>JavaScript Data Types</h2>
<p>JavaScript has dynamic types. This means that the same variable can be used to hold
different data types:</p>
<p id="demo"></p>
```

```
<script>
let x;      // Now x is undefined
x = 5;      // Now x is a Number
x = "Karthi"; // Now x is a String
document.getElementById("demo").innerHTML = x;
</script>
</body>
</html>
```

**Output:-**

### JavaScript Data Types

JavaScript has dynamic types. This means that the same variable can be used to hold different data types:

Karthi

## JavaScript Strings

A string (or a text string) is a series of characters like "John Doe".

Strings are written with quotes. You can use single or double quotes:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Strings</h2>
<p>Strings are written with quotes. You can use single or double quotes:</p>
<p id="demo"></p>
<script>
let carName1 = "Karthi";
let carName2 = 'Kanna';
document.getElementById("demo").innerHTML =
carName1 + "<br>" +
carName2;
</script>
</body>
</html>
```

**Output:-**

## JavaScript Strings

Strings are written with quotes. You can use single or double quotes:

Karthi  
Kanna

## JavaScript Numbers

JavaScript has only one type of numbers.

Example

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Numbers</h2>
<p>Numbers can be written with, or without decimals:</p>
<p id="demo"></p>
<script>
let x1 = 34.00;
let x2 = 34;
let x3 = 3.14;
```



```
document.getElementById("demo").innerHTML =  
x1 + "<br>" + x2 + "<br>" + x3;  
</script>  
</body>  
</html>
```

### Output:-

Numbers can be written with, or without decimals:

34  
34  
3.14

### JavaScript Events

- HTML events are **"things"** that happen to HTML elements.
- When JavaScript is used in HTML pages, JavaScript can **"react"** on these events.

### HTML Events

- ✓ An HTML event can be something the browser does, or something a user does.
- ✓ Here are some examples of HTML events:
- ✓ An HTML web page has finished loading
- ✓ An HTML input field was changed
- ✓ An HTML button was clicked

Often, when events happen, you may want to do something.

JavaScript lets you execute code when events are detected.

HTML allows event handler attributes, **with JavaScript code**, to be added to HTML elements.

With single quotes:

```
<element event='some JavaScript'>
```

With double quotes:

```
<element event="some JavaScript">
```

In the following example, an `onclick` attribute (with code), is added to a `<button>` element:

### Java Script Events:-

```
<!DOCTYPE html>
```

```
<html>
<body>

<button onclick="document.getElementById('demo').innerHTML=Date()">The time
is?</button>

<p id="demo"></p>

</body>
</html>
```

### Output:-

Sat Jun 04 2022 12:39:02 GMT+0530 (India Standard Time)  
([https://www.w3schools.com/js/tryit.asp?filename=tryjs\\_event\\_onclick1](https://www.w3schools.com/js/tryit.asp?filename=tryjs_event_onclick1))

## JavaScript Event Handlers

Event handlers can be used to handle and verify user input, user actions, and browser actions:

- Things that should be done every time a page loads
- Things that should be done when the page is closed
- Action that should be performed when a user clicks a button
- Content that should be verified when a user inputs data

Many different methods can be used to let JavaScript work with events:

- HTML event attributes can execute JavaScript code directly
- HTML event attributes can call JavaScript functions
- You can assign your own event handler functions to HTML elements
- You can prevent events from being sent or being handled

## JavaScript Functions

- ✓ A JavaScript function is a block of code designed to perform a particular task.
- ✓ A JavaScript function is executed when "something" invokes it (calls it).

## JavaScript Function Syntax

A JavaScript function is defined with the function keyword, followed by a **name**, followed by parentheses (). Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).

The parentheses may include parameter names separated by commas:

**(parameter1, parameter2, ...)**

The code to be executed, by the function, is placed inside curly brackets: {}

```
function name(parameter1, parameter2, parameter3) {  
  // code to be executed  
}
```

- ✓ Function **parameters** are listed inside the parentheses () in the function definition.
- ✓ Function **arguments** are the **values** received by the function when it is invoked.
- ✓ Inside the function, the arguments (the parameters) behave as local variables.

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

```
<h2>JavaScript Functions</h2>
```

```
<p>This example calls a function which performs a calculation, and returns the result:</p>
```

```
<p id="demo"></p>
```

```
<script>  
function myFunction(p1, p2) {  
  return p1 + p2;  
}  
document.getElementById("demo").innerHTML = myFunction(27, 3);  
</script>
```

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

### Output:-

#### JavaScript Functions

This example calls a function which performs a calculation, and returns the result:

30

#### JavaScript Variables:-

##### What are Variables?

Variables are containers for storing data (storing data values).

In this example, `x`, `y`, and `z`, are variables, declared with the `var` keyword:

Coding:-

```

<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Variables</h1>
<p>In this example, x, y, and z are variables.</p>
<p id="demo"></p>
<script>
var x = 28;
var y = 6;
var z = x + y;
document.getElementById("demo").innerHTML =
"The value of z is: " + z;
</script>
</body>
</html>
(https://www.w3schools.com/js/tryit.asp?filename=tryjs_variables)

```

## 1.7.AJAX: GET and POST

Ajax is an acronym for Asynchronous Javascript and XML. It is used to communicate with the server without refreshing the web page and thus increasing the user experience and better performance.

### How does it work?

First, let us understand what does asynchronous actually mean. There are two types of requests synchronous as well as asynchronous. Synchronous requests are the one which follows sequentially i.e if one process is going on and in the same time another process wants to be executed, it will not be allowed that means the only one process at a time will be executed. This is not good because in this type most of the time CPU remains idle such as during I/O operation in the process which are the order of magnitude slower than the CPU processing the instructions. Thus to make the full utilization of the CPU and other resources use asynchronous calls. For more information visit this [link](#). Why the word javascript is present here. Actually, the requests are made through the use of javascript functions. Now the term XML which is used to create

**XML Http Request object:** Thus the summary of the above explanation is that Ajax allows web pages to be updated asynchronously by exchanging small amounts of data with the server behind the scenes. Now discuss the important part and its implementation. For implementing Ajax, only be aware of XMLHttpRequest object. Now, what actually it is. It is an object used to exchange data with the server behind the scenes. Try to remember the paradigm of OOP which says that object communicates through calling methods (or in general sense message passing). The same case applied here as well. Usually, create this object and use it to call the methods which result in effective communication. All modern browsers support the XMLHttpRequest object.

**Basic Syntax:** The syntax of creating the object is given below

```
req = new XMLHttpRequest();
```

There are two types of methods open() and send(). Uses of these methods explained below.

```
req.open("GET", "abc.php", true);  
req.send();
```

The above two lines described the two methods. req stands for the request, it is basically a reference variable. The GET parameter is as usual one of two types of methods to send the request. Use POST as well depending upon whether send the data through POST or GET method. The second parameter being the name of the file which actually handles the requests and processes them. The third parameter is true, it tells that whether the requests are processed asynchronously or synchronously. It is by default true which means that requests are asynchronous. The open() method prepares the request before sending it to the server. The send method is used to send the request to the server. Sending the parameter through getting or POST request. The syntax is given below

```
req.open("GET", "abc.php?x=25", true);  
req.send();
```

In the above lines of code, the specified query in the form of URL followed by ? which is further followed by the name of the variable then = and then the corresponding value. If sending two or more variables use ampersand(&) sign between the two variables. The above method as shown applies for GET request. Sending the data through the POST, then send it in the send method as shown below.

```
req.send("name=johndoe&marks=99");
```

Use of setRequestHeader() method as shown below.

```
req.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
```

### **Events and handling mechanism:**

Any action performs on the clicking button, hovering over elements, page loading etc all are termed as events. Also aware of fact that javascript can detect events. So bind the code of specific event with its action which can be implemented by javascript. These are basically event handlers.

Implementing event handlers which actually hold the events. Events handlers are basically functions written in javascript which act on or set into action when an event is fired by the user. When sending the request through send method usually get the response from the server later. But getting of response time is not known. So track it. Therefore to keep a track of the response onreadystatechange event which is binding with the event handler(function) which will get executed when a response comes.

When a request to the server is sending perform actions based on the response. The onreadystatechange event is triggered every time the readyState changes.

So what actually a ready state is and when will the onreadystatechange event actually occur and how many times it will occur between the request and response? The XMLHttpRequest object has a property called as readyState whose value changes in the complete request-response journey i.e when a request is prepared, sent, resolves, processed and when the response comes. That's why it is called as onreadystatechange. The onreadystatechange stores a function (or the name of the function) to be called automatically each time the readyState property changes. The readyState holds different values ranging from 0 to 4.

1. request not initialized
2. server connection established
3. request received
4. processing request
5. request finished and response is ready

XMLHttpRequest also has a property named as status. The status has following values

- 200: "OK"
- 404: "Page not found"

Now remember it always that when readyState is 4 and status is 200, the response is ready. The whole thing described above is implemented in coding as given below

```
<p id = "dis">< /p>
req.onreadystatechange = function(){
    if(req.readyState == 4 && req.status == 200){
        document.getElementById("dis").innerHTML = req.responseText;
    }
}
```

#### **Advantages:**

1. Speed is enhanced as there is no need to reload the page again.
2. AJAX make asynchronous calls to a web server, this means client browsers avoid waiting for all the data to arrive before starting of rendering.
3. Form validation can be done successfully through it.
4. Bandwidth utilization – It saves memory when the data is fetched from the same page.
5. More interactive.

#### **Disadvantages:**

1. Ajax is dependent on Javascript. If there is some Javascript problem with the browser or in the OS, Ajax will not support.

2. Ajax can be problematic in Search engines as it uses Javascript for most of its parts.
3. Source code written in AJAX is easily human readable. There will be some security issues in Ajax.
4. Debugging is difficult.
5. Problem with browser back button when using AJAX enabled pages.

### **AJAX GET Coding:-**

```
<!DOCTYPE html>
<html>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>
<script>
$(document).ready(function(){
    $("button").click(function(){
        $.get("demo_test.asp", function(data, status){
            alert("Data: " + data + "\nStatus: " + status);
        });
    });
});
</script>
</head>
<body>

<button>Send an HTTP GET request to a page and get the result back</button>

</body>
</html>
```

### **AJAX POST CODING:-**

```
<!DOCTYPE html>
<html>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>
<script>
$(document).ready(function(){
    $("button").click(function(){
        $.post("demo_test_post.asp",
        {
            name: "Donald Duck",
            city: "Duckburg"
        },
        function(data,status){
            alert("Data: " + data + "\nStatus: " + status);
        });
    });
});
</script>
```

```
</head>
<body>
<button>Send an HTTP POST request to a page and get the result back</button>
</body>
</html>
```

[[https://www.w3schools.com/jquery/tryit.asp?filename=tryjquery\\_ajax\\_post](https://www.w3schools.com/jquery/tryit.asp?filename=tryjquery_ajax_post)]



## UNIT II

## SERVER SIDE PROGRAMMING WITH NODE JS

9

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.

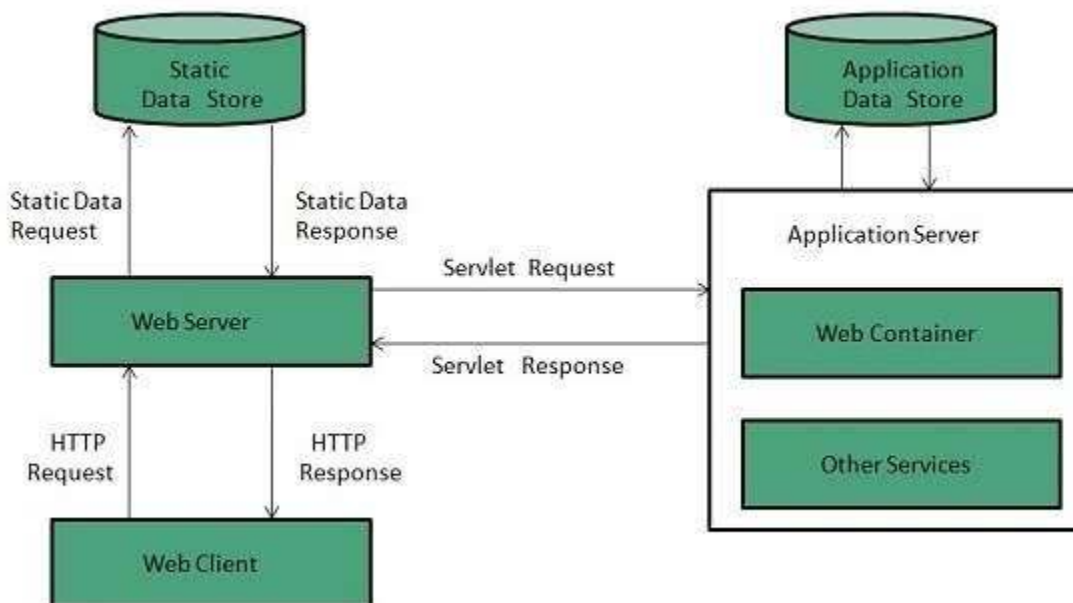
### Introduction to Web Servers :-

**Web server** is a computer where the web content is stored. Basically web server is used to host the web sites but there exists other web servers also such as gaming, storage, FTP, email etc. Web site is collection of web pages while web server is a software that respond to the request for web resources.

### Web Server Working

Web server respond to the client request in either of the following two ways:

- ✓ Sending the file to the client associated with the requested URL.
- ✓ Generating response by invoking a script and communicating with database



### Key Points

- ✓ When client sends request for a web page, the web server search for the requested page if requested page is found then it will send it to client with an HTTP response.
- ✓ If the requested web page is not found, web server will the send an **HTTP response:Error 404 Not found.**
- ✓ If client has requested for some other resources then the web server will contact to the application server and data store to construct the HTTP response.

## Architecture:-

Web Server Architecture follows the following two approaches:

1. Concurrent Approach
2. Single-Process-Event-Driven Approach.

## Concurrent Approach:-

Concurrent approach allows the web server to handle multiple client requests at the same time. It can be achieved by following methods:

- Multi-process
- Multi-threaded
- Hybrid method.

## Multi-processing:-

In this a single process (parent process) initiates several single-threaded child processes and distribute incoming requests to these child processes. Each of the child processes are responsible for handling single request. It is the responsibility of parent process to monitor the load and decide if processes should be killed or forked.

## Multi-threaded:-

Unlike Multi-process, it creates multiple single-threaded process.

## Hybrid

It is combination of above two approaches. In this approach multiple process are created and each process initiates multiple threads. Each of the threads handles one connection. Using multiple threads in single process results in less load on system resources.

## Examples:-

Following table describes the most leading web servers available today:

### S.N.

### Web Server Descriptino

#### Apache HTTP Server

- This is the most popular web server in the world developed by the Apache Software Foundation. Apache web server is an open source software and can be installed on almost all operating systems including Linux, UNIX, Windows, FreeBSD, Mac OS X and more. About 60% of the web server machines run the Apache Web Server.
- 1

#### Internet Information Services (IIS)

2. The Internet Information Server (IIS) is a high performance Web Server from Microsoft. This web server runs on Windows NT/2000 and 2003 platforms (and may be on upcoming

new Windows version also). IIS comes bundled with Windows NT/2000 and 2003; Because IIS is tightly integrated with the operating system so it is relatively easy to administer it.

### **Lighttpd**

- The lighttpd, pronounced lighty is also a free web server that is distributed with the
3. FreeBSD operating system. This open source web server is fast, secure and consumes much less CPU power. Lighttpd can also run on Windows, Mac OS X, Linux and Solaris operating systems.

### **Sun Java System Web Server**

- This web server from Sun Microsystems is suited for medium and large web sites. Though the server is free it is not open source. It however, runs on Windows, Linux and UNIX
4. platforms. The Sun Java System web server supports various languages, scripts and technologies required for Web 2.0 such as JSP, Java Servlets, PHP, Perl, Python, and Ruby on Rails, ASP and Coldfusion etc.

### **Jigsaw Server**

- Jigsaw (W3C's Server) comes from the World Wide Web Consortium. It is open source and free and can run on various platforms like Linux, UNIX, Windows, and Mac OS X Free BSD etc. Jigsaw has been written in Java and can run CGI scripts and PHP programs.
- 5.

## **Javascript in the Desktop with NodeJS:-**

### **Using Node.js instead of Java:-**

Using a Node.js-based application can mitigate many of the drawbacks mentioned above, while providing additional advantages in all three areas:

- **Developer velocity.** To create a new Node.js server-side application, you only need to write about 10 lines of code. Check out the web server example on the front page of the Node.js website for a simple, fully functional example. You can also create a REST-based server application using freely available modules, such as Express. Any time you need to make a change, you just restart the application, with no compilation required. In addition, the front end of the application is also in JavaScript. That's a good thing because having everything in the same language boosts velocity. The coding styles are the same, the developer doesn't need to switch programming methods or languages, and you can reuse some code, such as third-party libraries.
- **Runtime environment.** Node.js enables JavaScript programming for the server. That means that most of the bootstrapping work you need to do to develop a server application requires little overhead. However, you'll still need to choose which framework to use (that is, Express, Hapi, Koa, etc.). You can find many blog posts about which one is better based on performance, velocity, usability, and so on. But ultimately, the choice comes down to the developer's personal preference.
- **System resource utilization.** Node.js is lightweight, using around 20 MB of memory and very little processing power. Because JavaScript is asynchronous, it can hand off the low-level work to the underlying operating system, while the application itself can handle more requests. This means that many processes can happily run in parallel on the same machine without hogging too many resources. Nevertheless, you should

make sure that the Node.js application isn't used to run CPU-intensive tasks. Node.js lives and dies on maximizing its extremely fast I/O capabilities, but because it's single threaded, the moment the CPU utilization increases all other requests begin to queue up. For developers coming from a multi-threaded environment, this is an extremely sensitive point.

### Node.js for the enterprise:-

Many case studies have demonstrated the value of Node.js in the enterprise. However, almost all these stories have one thing in common: the applications are kept under tight control in a cloud-based or SaaS environment, where DevOps teams can constantly monitor, debug, refactor, and update them to ensure they always run smoothly. Companies such as PayPal, Walmart, and Dow Jones serve millions of requests per day through their Node-based applications. If something goes wrong, issues can be mitigated smoothly, with little impact on their customers.

For an on-premise solution, however, developers have none of these luxuries. After installing software on customers' machines, developers need in-depth, precise information in order to understand how the application is performing. Performance can manifest itself in several different ways:

- **Application crashes.** When your application goes down, the customer will contact support and expect a prompt solution with minimal disruption. To understand the issues, developers normally need several files, including log files, heap dumps, and audits.
- **Performance monitoring.** A bug that affects performance can be even worse than sporadic crashes. You need a way for developers to determine why the application is performing as it is so they can provide a solution.
- **Bugs.** When customers encounter a critical bug, they report it to support, which passes it on to the developer, along with an expectation of a quick fix by way of a patch or new release. Unlike SaaS-based applications, however, there's no easy way to roll back to a previous release.
- **Security and licensing.** This one is more of an issue for software providers than for the customer, but JavaScript doesn't let you protect intellectual property in the code. Minification and obfuscation can only do so much, and they can be reverse-engineered. Developers must ensure that the software is legally protected so intellectual property isn't easily compromised.

When developing in Node.js, these and other problems need to be carefully thought through. There's no compilation with JavaScript, so full testing is necessary. For example, something as simple as misspelling a function name won't be caught until you run the code. Extensive automated testing can mitigate this, however. Also, logging needs to be extremely thorough so developers can determine the true problem. You should run performance tests that accurately mimic the types of environments on which the application will run, and licensing should be watertight.

Despite all these caveats and potential pitfalls, Node.js was the right technology for what we wanted to achieve. Developer velocity is one of the most important factors in the era of agile programming and continuous delivery, and with Node.js our velocity increased significantly. We developed a complex state machine in just four weeks, about half of what we estimated it would take to develop the equivalent code in Java, and we saw similar gains after rewriting large sections of the product's back end.

Another huge benefit of coding in Node.js, and in JavaScript in general, is that developers spend all their time actually writing and testing code, rather than having to wait for the codebase to compile first. Most of the work we needed to do was ultimately suited for asynchronous, I/O-based, single-threaded processes. Our developers are now able to switch from client to server development smoothly.

### **Example:-**

Node.js is an open source server environment.

Node.js allows you to run JavaScript on the server.

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

http.createServer(function (req, res) {

    res.writeHead(200, {'Content-Type': 'text/plain'});

    res.end('Hello World!');

}).listen(8080);
```

### **Output:-**

```
http://localhost:8080
Hello World!
```

## **NPM – Serving files with the http module:-**

### **What is npm?**

- ♣ **npm** is the world's largest **Software Library** (Registry)
- ♣ **npm** is also a software **Package Manager** and **Installer**

### **The World's Largest Software Registry (Library)**

- ♣ **npm** is the world's largest **Software Registry**.

The registry contains over 800,000 **code packages**.

♣ **Open-source** developers use **npm** to **share** software.

Many organizations also use npm to manage private development.

### Using npm is Free

**npm** is free to use.

You can download all npm public software packages without any registration or logon.

### Command Line Client

**npm** includes a **CLI** (Command Line Client) that can be used to download and install software:

### Windows Example

```
C:\>npm install <package>
```

### Mac OS Example

```
>npm install <package>
```

### ♣ Installing npm

**npm** is installed with **Node.js**

This means that you have to install Node.js to get npm installed on your computer.

Download Node.js from the official Node.js web site: <https://nodejs.org>

### ♣ Software Package Manager:-

The name **npm** (Node Package Manager) stems from when npm first was created as a package manager for Node.js.

All **npm** packages are defined in files called **package.json**.

The content of package.json must be written in **JSON**.

At least two fields must be present in the definition file: **name** and **version**.

### Example

```
{  
"name" : "foo",  
"version" : "1.2.3",  
}
```

```
"description" : "A package for fooing things",
"main" : "foo.js",
"keywords" : ["foo", "fool", "foolish"],
"author" : "John Doe",
"licence" : "ISC"
}
```

## ♣ Managing Dependencies

**npm** can manage **dependencies**.

**npm** can (in one command line) install all the dependencies of a project.

Dependencies are also defined in **package.json**.

## Sharing Your Software

If you want to share your own software in the **npm registry**, you can sign in at:

<https://www.npmjs.com>

## Publishing a Package

You can publish **any directory** from your computer as long as the directory has a **package.json** file.

Check if npm is installed:

```
C:\>npm
```

Check if you are logged in:

```
C:\>npm whoami
```

If not, log in:

```
C:\>npm login
```

```
Username: <your username>
```

```
Password: <your password>
```

Navigate to your project and publish your project:

```
C:\Users\myuser>cd myproject
```

```
C:\Users\myuser\myproject>npm publish
```

```
const _ = require("lodash");
```

```
let colors = ['blue', 'green', 'yellow', 'red'];
```

```
let firstElement = _.first(colors);
let lastElement = _.last(colors);
console.log(`First element: ${firstElement}`);
console.log(`Last element: ${lastElement}`);
```

### Output:-

First element: blue  
Last element: red

```
// HTTP Module for Creating Server and Serving Static Files Using Node.js
// Static Files: HTML, CSS, JS, Images
// Get Complete Source Code from Pabbly.com
```

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

```
http.createServer(function(req, res){

  if(req.url === "/"){
    fs.readFile("./public/index.html", "UTF-8", function(err, html){
      res.writeHead(200, {"Content-Type": "text/html"});
      res.end(html);
    });
  } else if(req.url.match("\.css$")){
    var cssPath = path.join(__dirname, 'public', req.url);
    var fileStream = fs.createReadStream(cssPath, "UTF-8");
    res.writeHead(200, {"Content-Type": "text/css"});
    fileStream.pipe(res);

  } else if(req.url.match("\.png$")){
    var imagePath = path.join(__dirname, 'public', req.url);
    var fileStream = fs.createReadStream(imagePath);
    res.writeHead(200, {"Content-Type": "image/png"});
    fileStream.pipe(res);
  } else {
    res.writeHead(404, {"Content-Type": "text/html"});
    res.end("No Page Found");
  }

}).listen(3000);
Copy
```

### Introduction to the Express framework:-



Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications. It facilitates the rapid development of Node based Web applications. Following are some of the core features of Express framework –

- Allows to set up middlewares to respond to HTTP Requests.
- Defines a routing table which is used to perform different actions based on HTTP Method and URL.
- Allows to dynamically render HTML Pages based on passing arguments to templates.

## Installing Express

Firstly, install the Express framework globally using NPM so that it can be used to create a web application using node terminal.

```
$ npm install express --save
```

The above command saves the installation locally in the **node\_modules** directory and creates a directory express inside node\_modules. You should install the following important modules along with express –

- **body-parser** – This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.
- **cookie-parser** – Parse Cookie header and populate req.cookies with an object keyed by the cookie names.
- **multer** – This is a node.js middleware for handling multipart/form-data.

```
$ npm install body-parser --save
```

```
$ npm install cookie-parser --save
```

```
$ npm install multer --save
```

## Hello world Example

Following is a very basic Express app which starts a server and listens on port 8081 for connection. This app responds with **Hello World!** for requests to the homepage. For every other path, it will respond with a **404 Not Found**.

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

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

```
var server = app.listen(8081, function () {  
  var host = server.address().address  
  var port = server.address().port
```

```
    console.log("Example app listening at http://%s:%s", host, port)
  })
```

Save the above code in a file named `server.js` and run it with the following command.

```
$ node server.js
```

You will see the following output –

```
Example app listening at http://0.0.0.0:8081
```

Open `http://127.0.0.1:8081/` in any browser to see the following result.



## Request & Response

Express application uses a callback function whose parameters are **request** and **response** objects.

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

- Request Object – The request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.
- Response Object – The response object represents the HTTP response that an Express app sends when it gets an HTTP request.

You can print **req** and **res** objects which provide a lot of information related to HTTP request and response including cookies, sessions, URL, etc.

## Basic Routing

We have seen a basic application which serves HTTP request for the homepage. Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).

We will extend our Hello World program to handle more types of HTTP requests.

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

// This responds with "Hello World" on the homepage
app.get('/', function (req, res) {
  console.log("Got a GET request for the homepage");
  res.send('Hello GET');
})

// This responds a POST request for the homepage
app.post('/', function (req, res) {
  console.log("Got a POST request for the homepage");
  res.send('Hello POST');
})

// This responds a DELETE request for the /del_user page.
app.delete('/del_user', function (req, res) {
  console.log("Got a DELETE request for /del_user");
  res.send('Hello DELETE');
})

// This responds a GET request for the /list_user page.
app.get('/list_user', function (req, res) {
  console.log("Got a GET request for /list_user");
  res.send('Page Listing');
})

// This responds a GET request for abcd, abxcd, ab123cd, and so on
app.get('/ab*cd', function(req, res) {
  console.log("Got a GET request for /ab*cd");
  res.send('Page Pattern Match');
})

var server = app.listen(8081, function () {
  var host = server.address().address
```

```
var port = server.address().port

console.log("Example app listening at http://%s:%s", host, port)
})
```

Save the above code in a file named server.js and run it with the following command.

```
$ node server.js
```

You will see the following output –

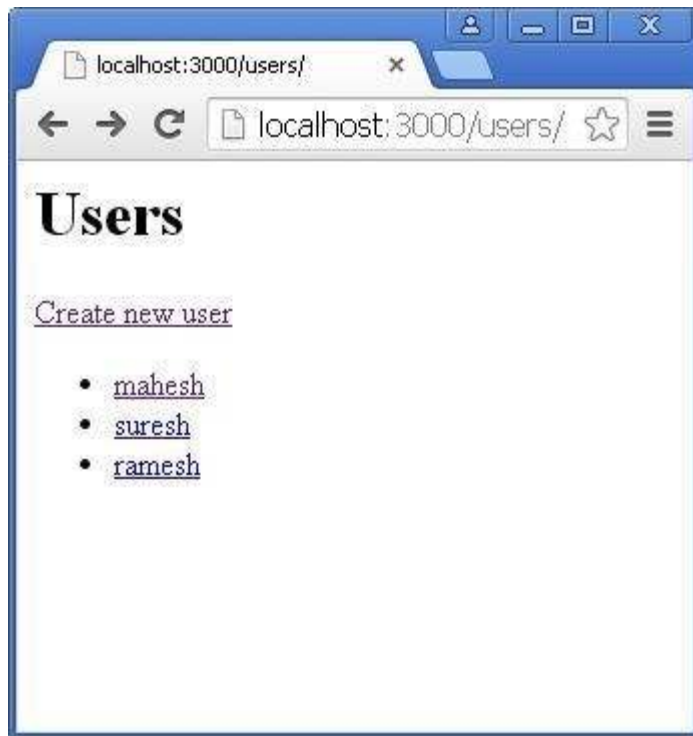
Example app listening at http://0.0.0.0:8081

Now you can try different requests at <http://127.0.0.1:8081> to see the output generated by server.js. Following are a few screens shots showing different responses for different URLs.

Screen showing again [http://127.0.0.1:8081/list\\_user](http://127.0.0.1:8081/list_user)



Screen showing again <http://127.0.0.1:8081/abcd>



Screen showing again <http://127.0.0.1:8081/abcdefg>



## Serving Static Files

Express provides a built-in middleware **express.static** to serve static files, such as images, CSS, JavaScript, etc.

You simply need to pass the name of the directory where you keep your static assets, to the **express.static** middleware to start serving the files directly. For example, if you keep your images, CSS, and JavaScript files in a directory named public, you can do this –

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

We will keep a few images in **public/images** sub-directory as follows –

```
node_modules
server.js
public/
public/images
public/images/logo.png
```

Let's modify "Hello Word" app to add the functionality to handle static files.

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

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

app.get('/', function (req, res) {
  res.send('Hello World');
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)
})
```

Save the above code in a file named server.js and run it with the following command.

```
$ node server.js
```

Now open <http://127.0.0.1:8081/images/logo.png> in any browser and see observe following result.



## GET Method

Here is a simple example which passes two values using HTML FORM GET method. We are going to use **process\_get** router inside server.js to handle this input.

```
<html>
  <body>

    <form action = "http://127.0.0.1:8081/process_get" method = "GET">
      First Name: <input type = "text" name = "first_name"> <br>
      Last Name: <input type = "text" name = "last_name">
      <input type = "submit" value = "Submit">
    </form>

  </body>
</html>
```

Let's save above code in index.htm and modify server.js to handle home page requests as well as the input sent by the HTML form.

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

app.use(express.static('public'));
app.get('/index.htm', function (req, res) {
  res.sendFile(__dirname + "/" + "index.htm" );
})

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

```
// Prepare output in JSON format
response = {
  first_name:req.query.first_name,
  last_name:req.query.last_name
};
console.log(response);
res.end(JSON.stringify(response));
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)
})
```

Accessing the HTML document using *http://127.0.0.1:8081/index.htm* will generate the following form –

First Name:

Last Name:

Now you can enter the First and Last Name and then click submit button to see the result and it should return the following result –

```
{"first_name":"John","last_name":"Paul"}
```

## POST Method

Here is a simple example which passes two values using HTML FORM POST method. We are going to use **process\_get** router inside server.js to handle this input.

```
<html>
<body>

  <form action = "http://127.0.0.1:8081/process_post" method = "POST">
    First Name: <input type = "text" name = "first_name"> <br>
    Last Name: <input type = "text" name = "last_name">
    <input type = "submit" value = "Submit">
  </form>

</body>
```



</html>

Let's save the above code in index.htm and modify server.js to handle home page requests as well as the input sent by the HTML form.

```
var express = require('express');
var app = express();
var bodyParser = require('body-parser');

// Create application/x-www-form-urlencoded parser
var urlencodedParser = bodyParser.urlencoded({ extended: false })

app.use(express.static('public'));
app.get('/index.htm', function (req, res) {
  res.sendFile(__dirname + "/" + "index.htm" );
})

app.post('/process_post', urlencodedParser, function (req, res) {
  // Prepare output in JSON format
  response = {
    first_name:req.body.first_name,
    last_name:req.body.last_name
  };
  console.log(response);
  res.end(JSON.stringify(response));
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)
})
```

Accessing the HTML document using *http://127.0.0.1:8081/index.htm* will generate the following form –

First Name:

Last Name:

Now you can enter the First and Last Name and then click the submit button to see the following result –

```
{"first_name":"John","last_name":"Paul"}
```

## File Upload

The following HTML code creates a file uploader form. This form has method attribute set to **POST** and enctype attribute is set to **multipart/form-data**

```
<html>
  <head>
    <title>File Uploading Form</title>
  </head>

  <body>
    <h3>File Upload:</h3>
    Select a file to upload: <br />

    <form action = "http://127.0.0.1:8081/file_upload" method = "POST"
      enctype = "multipart/form-data">
      <input type="file" name="file" size="50" />
      <br />
      <input type = "submit" value = "Upload File" />
    </form>

  </body>
</html>
```

Let's save above code in index.htm and modify server.js to handle home page requests as well as file upload.

```
var express = require('express');
var app = express();
var fs = require("fs");

var bodyParser = require('body-parser');
var multer = require('multer');

app.use(express.static('public'));
app.use(bodyParser.urlencoded({ extended: false }));
app.use(multer({ dest: '/tmp/'}));

app.get('/index.htm', function (req, res) {
  res.sendFile(__dirname + "/" + "index.htm" );
})

app.post('/file_upload', function (req, res) {
  console.log(req.files.file.name);
})
```

```

console.log(req.files.file.path);
console.log(req.files.file.type);
var file = __dirname + "/" + req.files.file.name;

fs.readFile( req.files.file.path, function (err, data) {
  fs.writeFile(file, data, function (err) {
    if( err ) {
      console.log( err );
    } else {
      response = {
        message:'File uploaded successfully',
        filename:req.files.file.name
      };
    }

    console.log( response );
    res.end( JSON.stringify( response ) );
  });
});
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)
})

```

Accessing the HTML document using *http://127.0.0.1:8081/index.htm* will generate the following form –

### **File Upload:**

Select  a  file  to  upload:

NOTE: This is just dummy form and would not work, but it must work at your server.

### **Server-side rendering with Templating Engines :-**

Most front-end developers have needed to work with server-side templates at some point. They may not have been called that—there was a time when these templates were simply called “PHP pages”, “JSPs”, or similar, before the push to apply separation of concerns on the web. These

days it's more common to see pages and views rendered by any back-end framework trimmed of as much business logic as possible.

Node is no different. Just as those other application frameworks need a way to separate the HTML produced from the data that populates it, so does ours. We want to be able to create a set of views loosely coupled to our application logic and have the application decide when to render them, and with what data.

## **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.

Some things you might look at when selecting a templating engine are:

Does it require any language besides JavaScript?

If so, you won't be able to use it on the client.

Does it require the existence of a DOM?

If so, you'll need to fake a DOM in Node to use it on the server—this can be done, but it adds another step, of course.

Does it allow templates to be compiled once and cached, before they're first rendered?

This may be a concern if you want to do all your template parsing up front, or if you'll render the same template many times.

Are there any restrictions on where or how templates are read into the template engine functions?

You may need to have templates come from a script element in the DOM, a module in Node, or from a string literal. Wherever your templates will be stored, you want a template engine that doesn't expect them to be elsewhere.

How much logic is allowed in the template?

Some template engines aim to be logic-less, with a subset going as far as only allowing insertion of strings. Others will allow you to write blocks of JavaScript as long as you like within the template. Logic-less templates are often managed by an additional layer

that prepares data for rendering. This aspect of the template engine can affect your architecture, so it's worth doing some research.

To keep things simple, we'll use Mustache templates for these examples. If you've never used a JavaScript template engine, Mustache is a good place to start because it's right in the middle of the logic/no-logic spectrum, offering a subset of functions but not access to the whole of JavaScript. Additionally, the same syntax is used in lots of other template engines, which is a bonus in case your site eventually requires something more powerful or more minimal.

We already set up our application to accept values from a form collecting our user's first and last name, but we never discussed the form itself. Moreover, we haven't done a thing to help our user out should they want to change their submitted name. Our first server-side template will be an edit page for our user's (somewhat limited) information:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Edit your user information</title>
  </head>
  <body>
    <h1>Edit your user information</h1>
    <form action="/" method="POST">
      <label>First name:
        <input type="text" name="firstName" value="{{firstName}}" />
      </label>
      <label>Last name:
        <input type="text" name="lastName" value="{{lastName}}" />
      </label>
      <input type="submit" value="Save" />
    </form>
  </body>
</html>
```

The double curly braces around `firstName` and `lastName` are the delimiters that tell Mustache where to substitute in the data we'll pass it when it's time to render the page. We're using the same property names for the existing values as we use for the name of the input elements to make things easy to keep track of, but the name, ID, and CSS class in the HTML are irrelevant to Mustache (which is one way it differs from some other template engines). To use our template, we need to install Mustache with npm, then modify our existing code to render the template instead of building the response HTML out of concatenated strings:

```
var connect = require("connect"),
    fs = require("fs"),
    mustache = require("mustache");
```

```

connect(
  connect.bodyParser(),
  function(req, res) {
    var userName = {
      firstName: req.body.firstName,
      lastName: req.body.lastName
    },
    // create and open the stream
    tplFile = fs.createReadStream(
      __dirname + "/public/edit.html",
      {encoding: "utf8"}
    ),
    template = "",
    html;

    tplFile.on("data", function(data) {
      template += data;
    });
    tplFile.on("end", function() {

      // render the template with the userName object as data
      html = mustache.to_html(template, userName);

      res.end(html);

    });
  }
).listen(8000);

```

Assuming we stored our template as edit.html in the public directory, this code will stream the contents into a variable in our application and, once the template is fully loaded, pass it the submitted first and last name to render as HTML. Then we send the HTML back like we normally would.

We've switched from fs.readFile() to reading the template content from a stream above, which would normally make things more efficient, except that we don't need to load these templates over and over again or wait for them to be requested to know we'll need them. Going forward, we'll treat our templates as dependencies, and load them accordingly. It's good to note that there's more than one way to read a file, though.

## Partial Templates:-

There will be certain pieces of the pages in our site that are repeated. Things like headers, footers, and certain application-specific widgets will appear on multiple pages, or every page.

We could copy the markup for those pieces into the page's template, but it will be easier to manage a single shared template for each of these common elements. Copying and pasting a reference to a child template is more maintainable than copying and pasting the markup itself. If we don't want to have to repeat the code that imports our CSS and client-side JavaScript, we can templatize a default version of the pages in our site, storing all that information there, in addition to site-wide headers and footers. We can leave a space for the content of each page and reuse the rest. Let's create a file with this boilerplate markup called `parent.html`:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>{{pageTitle}}</title>
    <link rel="stylesheet" src="css/style.css" />
    {{>stylesheets}}
  </head>
  <body>
    {{>content}}
    <footer>&copy; 2011 Node for Front-End Developers</footer>
  </body>
  {{>scripts}}
</html>
```

Every template engine handles nested templates differently. While Mustache uses the `{{>...}}` syntax you see above, templating engines that are otherwise very similar to Mustache may use something different. Try not to get too caught up in the particular syntax of this engine, just know that the greater-than sign here indicates a child template, and we'll use that to do composition.

To allow this parent template to load individual child templates and create full pages, we'll have to add some robustness into our application code. Instead of loading and rendering templates inline, we'll load another third-party module to manage loading, and move rendering to its own function.

The `requirejs` module works like the client-side `Require.js` utility, providing a slightly different approach to dependency management than Node's. The difference we're taking advantage of below is the `text.js` extension. We'll need to download this from the `Require.js` site and save it in our Node application's root directory. This will allow `Require` to load and manage text files, guaranteeing that they're loaded efficiently and available when they're needed. Adding the prefix `text!` to the path of the dependency (our template, in this case) tells `Require` to load the file as text instead of trying to evaluate it:

```
var connect = require("connect"),
    fs = require("fs"),
    mustache = require("mustache"),
    requirejs = require("requirejs"),
```

```

parentTmpl;

// configure requirejs to fall back to Node's require if a module is not found
requirejs.config({ nodeRequire: require });

connect(
  connect.static(__dirname + "/public"),
  connect.router(function(app) {
    app.get("/show/:tmpl/:firstName/:lastName", function(req, res) {
      var userName = {
        firstName: req.params.firstName,
        lastName: req.params.lastName
      };
      // once the parent template is loaded, render the page
      requirejs(["text!public/parent.html"], function(_parentTmpl) {
        parentTmpl = _parentTmpl;
        render(res, req.params.tmpl + ".html", userName);
      });
    });
  })
).listen(8000);

function render(res, filename, data, style, script, callback) {
  // load the template and return control to another function or send the response
  requirejs(["text!public/" + filename], function(tmpl) {
    if(callback) {
      callback(res, tmpl, data, style, script);
    } else {
      // render parent template with page template as a child
      var html = mustache.to_html(
        parentTmpl,
        {content: data},
        {content: tmpl, stylesheets: style || "", scripts: script || ""}
      );
      res.end(html);
    }
  });
}

```

While we could make our system for rendering templates within a wrapper far more robust, this smaller amount of code will let us do quite a bit (within certain constraints). In addition to the content for each page, our parent template and its rendering function will accept additional CSS or JavaScript imports as string literals. We can allow the render function to send back the rendered HTML without modification, or we can override that behavior by passing in an additional callback to call once the child template loads.



Now that we have files being loaded and requests being made in order, you can begin to see the issues with nesting callbacks that frighten some people who are new to Node. We've abstracted out some functionality, but there are still anonymous functions and nested logic. If you're used to DOM scripting, this probably isn't as scary, but you've probably also recognized it as something that needs to be managed. Although the code above is all in a single file for clarity, in a real application we'd put all of our template rendering code into its own module and move the logic into named functions private to that module.

### **Parsing Other File Types:-**

If we wanted to dynamically generate CSS (maybe for use in a theme builder of some sort) or JavaScript (perhaps cherry-picking pieces of a more comprehensive JS library), all we have to do is change the file type we're reading in. Because we read in the files as strings, it makes no difference if they're something Node could technically try to execute. Here again, Node's lack of out-of-the-box handling for common file types works to our advantage. Any file can be an application, a library, a template, or a static resource—it's entirely up to us and how we treat it.

Let's say that we do indeed have a site offering some kind of customizable client-side widget library. For example, assume we have a customizable CSS file using Mustache to populate user-defined values:

```
h1 { color: #{{main}}; }
h2 { color: #{{secondary}}; }
input { border-style: {{border}}; border-radius: {{corners}}px; }
```

To provide our end users with CSS and JavaScript matching their site's needs, we only really need to change our routing. We'll assume the values we're receiving are coming from a form post, and that the rest of our application file remains the same:

```
connect(
  connect.static(_dirname + "/public"),
  connect.bodyParser(),
  connect.router(function(app) {
    app.post("/theme", function(req, res) {
      var theme = {
        main: req.body.mainColor,
        secondary: req.body.secondaryColor,
        border: req.body.borderStyle,
        corners: req.body.borderRadius
      };
      // load and render the CSS template
      requirejs(["text!public/css/theme.css"], function(tmpl) {
        var css = mustache.to_html(tmpl, theme);
        res.writeHead(200, {
          "Content-Type": "text/css",
```

```

        "Content-Length": css.length
    });
    res.end(css);
});
};
app.post("/builder", function(req, res) {
    var options = {
        shim: req.body.html5shim,
        flash: req.body.useFlash,
        sockets: req.body.useWebSockets,
        jsonp: req.body.useJsonp
    };
    // load and render the JS template
    requirejs(["text!public/js/builder.js"], function(tmpl) {
        var js = mustache.to_html(tmpl, options);
        res.writeHead(200, {
            "Content-Type": "application/javascript",
            "Content-Length": js.length
        });
        res.end(js);
    });
});
}).listen(8000);

```

Since we don't want our CSS or JavaScript composed into our parent template, we can skip the render function and recreate its functionality for both MIME types we'll want to return. You can probably see how the logic to render and return a response could be abstracted out into a function of its own, but since our example application only handles these two scenarios, the less efficient code above will get the job done.

### Creating Files on the Fly:-

An in-depth exploration of any of the necessary modules is out of scope for this guide, but there are numerous libraries to help you generate other types of files from within Node. While these aren't templates in the same sense as the templates we've looked at so far, they can map directly to a filename in a URL and be built and served in a way that's invisible to the user, just like templates. For instance, you might have cause to dynamically generate image files to do resizing or add watermarks. Modules to do that will have their own APIs, and can be loaded and used like any template engine or other module. To keep your application code clean, you'd probably choose to keep the code you write to work with those APIs in a separate file, the same way an external template stores information specific to rendering HTML or other text-based file types.

### Static Files - async/await :-

Java Asynchronous await is defined as performing I/O bound operations and doesn't need any application responsiveness. These functions are normally used in file and network operations as they require callbacks executed on operation completion; also that this function always returns a value. With the help of the `await` keyword, asynchronous calls are used inside regular control flow statements, and it is a non-blocking code. In this topic, we are going to learn about Java `async await`.

## Syntax

The general signature of `async / await` is given as

```
async Void test() {  
    print('Welcome to EDUCBA');  
}
```

## The await goes like

```
const test=async() =>  
{  
    Await test ();  
    Print ("completed");  
}
```

## How does the `async-await` function work in Java?

`Async await` function helps to write synchronous code meanwhile performing `async` tasks behind the code. And we need to have the `async` keyword. And next is the `awaited` part that says run the asynchronous code normally and proceed on to the next line of code. `Await` is a new operator that automatically waits for a promise to resolve the running process, and it is used inside the `async` function. It leads to syntax error if it is used in any other case.

In the case of error handling, if the function throws an error, the promise made by `async` will be rejected. If the respective function happens to return a value, the promise will be solved. This non-blocking code runs on a separate thread and notifies the main thread about its completion or failure of a task. `Try-catch` is used in a function to handle the errors synchronously. Let's take a sample beginning like

```
async function hello() {  
    //process waiting  
    await new Promise(res => setTimeout(res, 2000));  
    // Rejection with 20 %  
    if(Math.random() > 0.2) {  
        throw new Error('Check the number.')    }  
    return 'number';  
}
```

The above code says that the function hello() is async, and it resolves it by returning a number and throwing an error by checking the number.

Next, using await and return together to suspend a process

```
async function miss() {  
  try {  
    return await hello();  
  } catch (e) {  
    return 'error caught';  
  }  
}
```

Better promising chaining with this function is given as

```
Async function promise1( req,res)  
{  
  Try  
  {  
    Let a=await a.get(req,uid);  
    Let b=await cart.get (yser,uid);  
    Res.send(await dosome(a,card));  
  }  
  Catch (err)  
  {  
    Res.send(err);  
  }  
}
```

So here await keyword instructs the function get () to complete before catching an error.

With this Completable future, it returns a future object. This Completable future is a reference to asynchronous computation and implements the future.

```
private static CompletableFuture<Void> hello{  
  try {  
    String intermediate = await(doA());  
    String res = await(doB(intermediate));  
    reportSuccess(res);  
  } catch (Throwable th) {  
    reportFailure(th);  
  }  
  return completedFuture(null);  
}
```

**Examples of Java async await:-**

```

import java.util.*;
import java.util.concurrent.*;
public class Async {
static List<Task> tasks = new ArrayList<>();
static ExecutorService executor = Executors.newScheduledThreadPool(3);
public static void main(String[] args) {
createTasks();
executeTasks();
}
private static void createTasks() {
for (int k= 0; k < 10; k++) {
tasks.add(new Task(k));
}
}
private static void executeTasks() {
for (Task task : tasks) {
executor.submit(task);
}
}
static class Task extends Thread {
int n;
public void run() {
try {
Thread.sleep(new Random (). nextInt (1000));
} catch (InterruptedException e) {
e.printStackTrace();
}
printNum();
}
private void printNum() {
System.out.print(n + " ");
}
public Task(int n) {
this.n = n;
}
}
}

```

### Output:-

1 0 3 4 2 5 8 9 6 7

### Fetching JSON from Express:-

The **express.json()** function is a built-in middleware function in Express. It parses incoming requests with JSON payloads and is based on **body-parser**.

**Syntax:**

`express.json( [options] )`

**Parameters:** The options parameter have various property like inflate, limit, type, etc.

**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.

```
node index.js
```

**Example 1: Filename: index.js**

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




app.use(express.json());

app.post('/', function (req, res) {
  console.log(req.body.name)
  res.end();
})

app.listen(PORT, function(err){
  if (err) console.log(err);
  console.log("Server listening on PORT", PORT);
});
```

**Steps to run the program:**

1. The project structure will look like this:

Name	Date modified	Type	Size
 node_modules	06-06-2020 04:55 PM	File folder	
 index.js	06-06-2020 04:54 PM	JS File	0 KB
 package-lock.json	06-06-2020 04:55 PM	JSON File	14 KB

2. Make sure you have installed **express** module using the following command:

```
npm install express
```

3. Run index.js file using below command:

```
node index.js
```

### Output:

Server listening on PORT 3000

4. Now make a POST request to *http://localhost:3000/* with header set to '**content-type: application/json**' and body **{“name”:”GeeksforGeeks”}**, then you will see the following output on your console:
5. Server listening on PORT 3000

### Example 2: Filename: index.js

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

// Without this middleware
// app.use(express.json());
app.post('/', function (req, res) {
  console.log(req.body.name)
  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 make a POST request to *http://localhost:3000/* with header set to **'content-type: application/json'** and body **{“name”:”GeeksforGeeks”}**, then you will see the following output on your console:

Server listening on PORT 3000

TypeError: Cannot read property 'name' of undefined



## UNIT III

## ADVANCED NODE JS AND DATABASE

### 9

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 NodeJS.

### Introduction to NoSQL databases:-

A **NoSQL** originally referring to non SQL or non relational is a database that provides a mechanism for storage and retrieval of data. This data is modeled in means other than the tabular relations used in relational databases. Such databases came into existence in the late 1960s, but did not obtain the NoSQL moniker until a surge of popularity in the early twenty-first century. NoSQL databases are used in real-time web applications and big data and their use are increasing over time. NoSQL systems are also sometimes called Not only SQL to emphasize the fact that they may support SQL-like query languages.

A NoSQL database includes simplicity of design, simpler horizontal scaling to clusters of machines and finer control over availability. The data structures used by NoSQL databases are different from those used by default in relational databases which makes some operations faster in NoSQL. The suitability of a given NoSQL database depends on the problem it should solve. Data structures used by NoSQL databases are sometimes also viewed as more flexible than relational database tables.

Many NoSQL stores compromise consistency in favor of availability, speed and partition tolerance. Barriers to the greater adoption of NoSQL stores include the use of low-level query languages, lack of standardized interfaces, and huge previous investments in existing relational databases. Most NoSQL stores lack true ACID(Atomicity, Consistency, Isolation, Durability) transactions but a few databases, such as MarkLogic, Aerospike, FairCom c-treeACE, Google Spanner (though technically a NewSQL database), Symas LMDB, and OrientDB have made them central to their designs.

Most NoSQL databases offer a concept of eventual consistency in which database changes are propagated to all nodes so queries for data might not return updated data immediately or might result in reading data that is not accurate which is a problem known as stale reads. Also some NoSQL systems may exhibit lost writes and other forms of data loss. Some NoSQL systems provide concepts such as write-ahead logging to avoid data loss. For distributed transaction processing across multiple databases, data consistency is an even bigger challenge. This is difficult for both NoSQL and relational databases. Even current relational databases do not allow referential integrity constraints to span databases. There are few systems that maintain both X/Open XA standards and ACID transactions for distributed transaction processing.

### Advantages of NoSQL:

There are many advantages of working with NoSQL databases such as MongoDB and Cassandra. The main advantages are high scalability and high availability.

1. **High scalability** – NoSQL database use sharding for horizontal scaling. Partitioning of data and placing it on multiple machines in such a way that the order of the data is preserved is sharding. Vertical scaling means adding more resources to the existing machine whereas horizontal scaling means adding more machines to handle the data. Vertical scaling is not that easy to implement but horizontal scaling is easy to implement. Examples of horizontal scaling databases are MongoDB, Cassandra etc. NoSQL can handle huge amount of data because of scalability, as the data grows NoSQL scale itself to handle that data in efficient manner.
2. **High availability** – Auto replication feature in NoSQL databases makes it highly available because in case of any failure data replicates itself to the previous consistent state.

### **Disadvantages of NoSQL:**

NoSQL has the following disadvantages.

1. **Narrow focus** – NoSQL databases have very narrow focus as it is mainly designed for storage but it provides very little functionality. Relational databases are a better choice in the field of Transaction Management than NoSQL.
2. **Open-source** – NoSQL is open-source database. There is no reliable standard for NoSQL yet. In other words two database systems are likely to be unequal.
3. **Management challenge** – The purpose of big data tools is to make management of a large amount of data as simple as possible. But it is not so easy. Data management in NoSQL is much more complex than a relational database. NoSQL, in particular, has a reputation for being challenging to install and even more hectic to manage on a daily basis.
4. **GUI is not available** – GUI mode tools to access the database is not flexibly available in the market.
5. **Backup** – Backup is a great weak point for some NoSQL databases like MongoDB. MongoDB has no approach for the backup of data in a consistent manner.
6. **Large document size** – Some database systems like MongoDB and CouchDB store data in JSON format. Which means that documents are quite large (BigData, network bandwidth, speed), and having descriptive key names actually hurts, since they increase the document size.

### **Types of NoSQL database:**

Types of NoSQL databases and the name of the databases system that falls in that category are:

1. MongoDB falls in the category of NoSQL document based database.
2. **Key value store:** Memcached, Redis, Coherence
3. **Tabular:** Hbase, Big Table, Accumulo
4. **Document based:** MongoDB, CouchDB, Cloudant

### **MongoDB system overview:-**

MongoDB is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document.

**Database** - Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

**Collection** - Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

**Document** - A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

The following table shows the relationship of RDBMS terminology with MongoDB.

<b>RDBMS</b>	<b>MongoDB</b>
Database	Database
Table	Collection
Tuple/Row	Document
Column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by MongoDB itself)
<b>Database Server and Client</b>	
mysql/Oracle	Mongod
mysql/sqlplus	Mongo

## Sample Document

Following example shows the document structure of a blog site, which is simply a comma separated key value pair.

```
{
  _id: ObjectId(7df78ad8902c)
  title: 'MongoDB Overview',
  description: 'MongoDB is no sql database',
  by: 'tutorials point',
  url: 'http://www.tutorialspoint.com',
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 100,
  comments: [
```

```

{
  user:'user1',
  message: 'My first comment',
  dateCreated: new Date(2011,1,20,2,15),
  like: 0
},
{
  user:'user2',
  message: 'My second comments',
  dateCreated: new Date(2011,1,25,7,45),
  like: 5
}
]
}

```

**\_id** is a 12 bytes hexadecimal number which assures the uniqueness of every document. You can provide **\_id** while inserting the document. If you don't provide then MongoDB provides a unique id for every document. These 12 bytes first 4 bytes for the current timestamp, next 3 bytes for machine id, next 2 bytes for process id of MongoDB server and remaining 3 bytes are simple incremental VALUE.

### **Mysqld / Oracle mongod**

Here is a list of some popular and multinational companies and organizations that are using MongoDB as their official database to perform and manage different business applications.

- Adobe
- McAfee
- LinkedIn
- FourSquare
- MetLife
- eBay
- SAP

Beginners need to know the purpose and requirement of why to use MongoDB or what is the need of it in contrast to SQL and other database systems. In simple words, it can be said that every modern-day application involves the concept of big data, analyzing different forms of data, fast features improvement in handling data, deployment flexibility, which old database systems are not competent enough to handle. Hence, MongoDB is the next choice.

Some basic requirements are supported by this NoSQL database, which is lacking in other database systems. These collective reasons make MongoDB popular among other database systems:

- Document-Oriented data storage, i.e., data, is stored in a JSON style format, which increases the readability of data as well.

- Replication and high availability of data.
- MongoDB provides Auto-sharding.
- Ad hoc queries are supported by MongoDB, which helps in searching by range queries, field, or using regex terms.
- Indexing of values can be used to create and improve the overall search performance in MongoDB. MongoDB allows any field to be indexed within a document.
- MongoDB has a rich collection of queries.
- Updating of data can be done at a faster pace.
- It can be integrated with other popular programming languages also to handle structured as well as unstructured data within various types of applications.
- It is easy to set up, i.e., install the MongoDB.
- Since MongoDB is a schema-less database, so there is no hassle of schema migration.
- Since it is a document-oriented language, document queries are used, which plays a vital role in supporting dynamic queries.
- Easily scalable.
- It is easy to have a performance tuning as compared to other relational databases.
- It helps in providing fast accessing of data because of its nature of implementing the internal memory to store the data.
- MongoDB is also used as a file system that can help in easy management of load balancing.
- MongoDB also supports the searching using the concept of regex (regular expression) as well as fields.
- Users can run MongoDB as a windows service also.
- It does not require any VM to run on different platforms.
- It also supports sharding of data.

### Basic **querying with MongoDB shell**:-

Populate the inventory collection, run the following:

```
db.inventory.insertMany([
  { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
  { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "A" },
  { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" },
  { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
  { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" }
]);
```

### Select All Documents in a Collection

To select all documents in the collection, pass an empty document as the query filter parameter to the find method. The query filter parameter determines the select criteria:

```
db.inventory.find( {} )
```

This operation corresponds to the following SQL statement:

```
SELECT * FROM inventory
```

For more information on the syntax of the method, see `find()`.

### Specify Equality Condition

To specify equality conditions, use `<field>:<value>` expressions in the query filter document:

#### MongoDB Shell

```
{ <field1>: <value1>, ... }
```

The following example selects from the `inventory` collection all documents where the `status` equals "D":

#### MongoDB Shell

```
db.inventory.find( { status: "D" } )
```

This operation corresponds to the following SQL statement:

This operation corresponds to the following SQL statement:  

```
SELECT * FROM inventory WHERE status = "D"
```

### Specify Conditions Using Query Operators

A query filter document can use the query operators to specify conditions in the following form:

#### MongoDB Shell

```
{ <field1>: { <operator1>: <value1> }, ... }
```

The following example retrieves all documents from the `inventory` collection where `status` equals either "A" or "D":

## MongoDB Shell

```
db.inventory.find( { status: { $in: [ "A", "D" ] } } )
```

The operation corresponds to the following SQL statement:

```
SELECT * FROM inventory WHERE status in ("A", "D")
```

### Specify & Conditions:-

A compound query can specify conditions for more than one field in the collection's documents. Implicitly, a logical AND conjunction connects the clauses of a compound query so that the query selects the documents in the collection that match all the conditions.

The following example retrieves all documents in the `inventory` collection where the `status` equals "A" **and** `qty` is less than ([\\$lt](#)) 30:

## MongoDB Shell

```
db.inventory.find( { status: "A", qty: { $lt: 30 } } )
```

The operation corresponds to the following SQL statement:

```
SELECT * FROM inventory WHERE status = "A" AND qty < 30
```

### Specify OR Conditions

Using the [\\$or](#) operator, you can specify a compound query that joins each clause with a logical OR conjunction so that the query selects the documents in the collection that match at least one condition.

The following example retrieves all documents in the collection where the `status` equals "A" **or** `qty` is less than ([\\$lt](#)) 30:

## MongoDB Shell

```
db.inventory.find( { $or: [ { status: "A" }, { qty: { $lt: 30 } } ] } )
```

The operation corresponds to the following SQL statement:

```
SELECT * FROM inventory WHERE status = "A" OR qty < 30
```

## Specify **AND** as well as **OR** Conditions

In the following example, the compound query document selects all documents in the collection where the `status` equals "A" and *either* `qty` is less than (\$lt) 30 *or* `item` starts with the character `p`:

MongoDB Shell

```
db.inventory.find( {  
  status: "A",  
  $or: [ { qty: { $lt: 30 } }, { item: /^p/ } ]  
} )
```

The operation corresponds to the following SQL statement:

```
SELECT * FROM inventory WHERE status = "A" AND ( qty < 30 OR item LIKE "p%" )
```

### **Request body parsing in Express:-**

Node.js body parsing middleware.

Parse incoming request bodies in a middleware before your handlers, available under the `req.body` property.

**Note** As `req.body`'s shape is based on user-controlled input, all properties and values in this object are untrusted and should be validated before trusting. For example, `req.body.foo.toString()` may fail in multiple ways, for example the `foo` property may not be there or may not be a string, and `toString` may not be a function and instead a string or other user input.

This does not handle multipart bodies, *due to their complex and typically large nature. For multipart bodies, you may be interested in the following modules:*

- busboy and connect-busboy
- multipart and connect-multipart
- formidable
- multer

This module provides the following parsers:

- JSON body parser
- Raw body parser
- Text body parser
- URL-encoded form body parser



Other body parsers you might be interested in:

- `body`
- `co-body`

## Installation

```
$ npm install body-parser
```

## API

```
var bodyParser = require('body-parser')
```

The `bodyParser` object exposes various factories to create middlewares. All middlewares will populate the `req.body` property with the parsed body when the `Content-Type` request header matches the `type` option, or an empty object (`{}`) if there was no body to parse, the `Content-Type` was not matched, or an error occurred.

The various errors returned by this module are described in the errors section.

### **`bodyParser.json([options])`**

Returns middleware that only parses `json` and only looks at requests where the `Content-Type` header matches the `type` option. This parser accepts any Unicode encoding of the body and supports automatic inflation of `gzip` and `deflate` encodings.

A new `body` object containing the parsed data is populated on the `request` object after the middleware (i.e. `req.body`).

## Options

The `json` function takes an optional `options` object that may contain any of the following keys:

- **Inflate** - When set to true, then deflated (compressed) bodies will be inflated; when false, deflated bodies are rejected. Defaults to true.
- **Limit** - Controls the maximum request body size. If this is a number, then the value specifies the number of bytes; if it is a string, the value is passed to the bytes library for parsing. Defaults to '100kb'.
- **Reviver** - The reviver option is passed directly to JSON.parse as the second argument. You can find more information on this argument in the MDN documentation about JSON.parse.
- **Strict** - When set to true, will only accept arrays and objects; when false will accept anything JSON.parse accepts. Defaults to true.
- **Type** - The type option is used to determine what media type the middleware will parse. This option can be a string, array of strings, or a function. If not a function, type option is passed directly to the type-is library and this can be an extension name (like json), a mime type (like application/json), or a mime type with a wildcard (like \*/\* or \*/json). If a function, the type option is called as fn(req) and the request is parsed if it returns a truthy value. Defaults to application/json.
- **Verify** - The verify option, if supplied, is called as verify(req, res, buf, encoding), where buf is a Buffer of the raw request body and encoding is the encoding of the request. The parsing can be aborted by throwing an error.

### **bodyParser.raw([options])**

Returns middleware that parses all bodies as a `Buffer` and only looks at requests where the `Content-Type` header matches the `type` option. This parser supports automatic inflation of `gzip` and `deflate` encodings.

A new `body` object containing the parsed data is populated on the `request` object after the middleware (i.e. `req.body`). This will be a `Buffer` object of the body.

**Options** - The `raw` function takes an optional options object that may contain any of the following keys:

**Inflate** - When set to `true`, then deflated (compressed) bodies will be inflated; when `false`, deflated bodies are rejected. Defaults to `true`.

**Limit** - Controls the maximum request body size. If this is a number, then the value specifies the number of bytes; if it is a string, the value is passed to the bytes library for parsing. Defaults to `'100kb'`.

**Type** - The `type` option is used to determine what media type the middleware will parse. This option can be a string, array of strings, or a function. If not a function, `type` option is passed directly to the `type-is` library and this can be an extension name (like `bin`), a mime type (like `application/octet-stream`), or a mime type with a wildcard (like `*/*` or `application/*`). If a function, the `type` option is called as `fn(req)` and the request is parsed if it returns a truthy value. Defaults to `application/octet-stream`.

**Verify** - The `verify` option, if supplied, is called as `verify(req, res, buf, encoding)`, where `buf` is a Buffer of the raw request body and `encoding` is the encoding of the request. The parsing can be aborted by throwing an error.

### **`bodyParser.text([options])`**

Returns middleware that parses all bodies as a string and only looks at requests where the `Content-Type` header matches the `type` option. This parser supports automatic inflation of `gzip` and `deflate` encodings.

A new `body` string containing the parsed data is populated on the `request` object after the middleware (i.e. `req.body`). This will be a string of the body.

**Options** - The `text` function takes an optional `options` object that may contain any of the following keys:

**DefaultCharset** - Specify the default character set for the text content if the charset is not specified in the Content-Type header of the request. Defaults to `utf-8`.

**Inflate** - When set to `true`, then deflated (compressed) bodies will be inflated; when `false`, deflated bodies are rejected. Defaults to `true`.

**Limit** - Controls the maximum request body size. If this is a number, then the value specifies the number of bytes; if it is a string, the value is passed to the bytes library for parsing. Defaults to `'100kb'`.

**Type** - The type option is used to determine what media type the middleware will parse. This option can be a string, array of strings, or a function. If not a function, type option is passed directly to the `type-is` library and this can be an extension name (like `txt`), a mime type (like `text/plain`), or a mime type with a wildcard (like `*/*` or `text/*`). If a function, the type option is called as `fn(req)` and the request is parsed if it returns a truthy value. Defaults to `text/plain`.

**Verify** - The verify option, if supplied, is called as `verify(req, res, buf, encoding)`, where `buf` is a Buffer of the raw request body and `encoding` is the encoding of the request. The parsing can be aborted by throwing an error.

### **`bodyParser.urlencoded([options])`**

Returns middleware that only parses `urlencoded` bodies and only looks at requests where the Content-Type header matches the `type` option. This parser accepts only UTF-8 encoding of the body and supports automatic inflation of `gzip` and `deflate` encodings.

A new `body` object containing the parsed data is populated on the `request` object after the middleware (i.e. `req.body`). This object will contain key-value pairs, where the value can be a string or array (when `extended` is `false`), or any type (when `extended` is `true`).

**Options** - The `urlencoded` function takes an optional `options` object that may contain any of the following keys:

**Extended** - The `extended` option allows to choose between parsing the URL-encoded data with the `querystring` library (when `false`) or the `qs` library (when `true`). The “extended” syntax allows for rich objects and arrays to be encoded into the URL-encoded format, allowing for a JSON-like experience with URL-encoded. For more information, please see the `qs` library.

Defaults to `true`, but using the default has been deprecated. Please research into the difference between `qs` and `querystring` and choose the appropriate setting.

**Inflate** - When set to true, then deflated (compressed) bodies will be inflated; when false, deflated bodies are rejected. Defaults to true.

**Limit** - Controls the maximum request body size. If this is a number, then the value specifies the number of bytes; if it is a string, the value is passed to the bytes library for parsing. Defaults to '100kb'.

**ParameterLimit** - The parameterLimit option controls the maximum number of parameters that are allowed in the URL-encoded data. If a request contains more parameters than this value, a 413 will be returned to the client. Defaults to 1000.

**Type** - The type option is used to determine what media type the middleware will parse. This option can be a string, array of strings, or a function. If not a function, type option is passed directly to the type-is library and this can be an extension name (like urlencoded), a mime type (like application/x-www-form-urlencoded), or a mime type with a wildcard (like \*/x-www-form-urlencoded). If a function, the type option is called as fn(req) and the request is parsed if it returns a truthy value. Defaults to application/x-www-form-urlencoded.

**Verify** - The verify option, if supplied, is called as verify(req, res, buf, encoding), where buf is a Buffer of the raw request body and encoding is the encoding of the request. The parsing can be aborted by throwing an error.

## Errors

The middlewares provided by this module create errors using the `http-errors` module. The errors will typically have a `status/statusCode` property that contains the suggested HTTP response code, an `expose` property to determine if the `message` property should be displayed to the client, a `type` property to determine the type of error without matching against the `message`, and a `body` property containing the read body, if available.

The following are the common errors created, though any error can come through for various reasons.

### Content encoding unsupported

This error will occur when the request had a `Content-Encoding` header that contained an encoding but the “inflation” option was set to `false`. The `status` property is set to 415, the `type` property is set to `'encoding.unsupported'`, and the `charset` property will be set to the encoding that is unsupported.

### Entity parse failed

This error will occur when the request contained an entity that could not be parsed by the middleware. The `status` property is set to 400, the `type` property is set to `'entity.parse.failed'`, and the `body` property is set to the entity value that failed parsing.

### Entity verify failed

This error will occur when the request contained an entity that could not be failed verification by the defined verify option. The status property is set to 403, the type property is set to 'entity.verify.failed', and the body property is set to the entity value that failed verification.

### **request aborted**

This error will occur when the request is aborted by the client before reading the body has finished. The received property will be set to the number of bytes received before the request was aborted and the expected property is set to the number of expected bytes. The status property is set to 400 and type property is set to 'request.aborted'.

### **request entity too large**

This error will occur when the request body's size is larger than the "limit" option. The limit property will be set to the byte limit and the length property will be set to the request body's length. The status property is set to 413 and the type property is set to 'entity.too.large'.

### **request size did not match content length**

This error will occur when the request's length did not match the length from the Content-Length header. This typically occurs when the request is malformed, typically when the Content-Length header was calculated based on characters instead of bytes. The status property is set to 400 and the type property is set to 'request.size.invalid'.

### **stream encoding should not be set**

This error will occur when something called the req.setEncoding method prior to this middleware. This module operates directly on bytes only and you cannot call req.setEncoding when using this module. The status property is set to 500 and the type property is set to 'stream.encoding.set'.

### **stream is not readable**

This error will occur when the request is no longer readable when this middleware attempts to read it. This typically means something other than a middleware from this module read the request body already and the middleware was also configured to read the same request. The status property is set to 500 and the type property is set to 'stream.not.readable'.

### **too many parameters -**

This error will occur when the content of the request exceeds the configured `parameterLimit` for the `urlencoded` parser. The status property is set to 413 and the type property is set to 'parameters.too.many'.

### **unsupported charset "BOGUS"**

This error will occur when the request had a charset parameter in the `Content-Type` header, but the `iconv-lite` module does not support it OR the parser does not support it. The charset is contained in the message as well as in the `charset` property. The `status` property is set to 415, the `type` property is set to `'charset.unsupported'`, and the `charset` property is set to the charset that is unsupported.

### **unsupported content encoding “bogus”**

This error will occur when the request had a `Content-Encoding` header that contained an unsupported encoding. The encoding is contained in the message as well as in the `encoding` property. The `status` property is set to 415, the `type` property is set to `'encoding.unsupported'`, and the `encoding` property is set to the encoding that is unsupported.

## **Examples**

### **Express/Connect top-level generic**

This example demonstrates adding a generic JSON and URL-encoded parser as a top-level middleware, which will parse the bodies of all incoming requests. This is the simplest setup.

```
var express = require('express')
var bodyParser = require('body-parser')

var app = express()

// parse application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({ extended: false }))

// parse application/json
app.use(bodyParser.json())

app.use(function (req, res) {
  res.setHeader('Content-Type', 'text/plain')
  res.write('you posted:\n')
  res.end(JSON.stringify(req.body, null, 2))
})
```

### **Express route-specific**

This example demonstrates adding body parsers specifically to the routes that need them. In general, this is the most recommended way to use `body-parser` with Express.

```
var express = require('express')
var bodyParser = require('body-parser')
```

```

var app = express()

// create application/json parser
var bodyParser = bodyParser.json()

// create application/x-www-form-urlencoded parser
var urlencodedParser = bodyParser.urlencoded({ extended: false })

// POST /login gets urlencoded bodies
app.post('/login', urlencodedParser, function (req, res) {
  res.send('welcome, ' + req.body.username)
})

// POST /api/users gets JSON bodies
app.post('/api/users', bodyParser.json, function (req, res) {
  // create user in req.body
})

```

### Change accepted type for parsers

All the parsers accept a `type` option which allows you to change the Content-Type that the middleware will parse.

```

var express = require('express')
var bodyParser = require('body-parser')

var app = express()

// parse various different custom JSON types as JSON
app.use(bodyParser.json({ type: 'application/*+json' }))

// parse some custom thing into a Buffer
app.use(bodyParser.raw({ type: 'application/vnd.custom-type' }))

// parse an HTML body into a string
app.use(bodyParser.text({ type: 'text/html' }))

```

### NodeJS MongoDB connection:-

To create a database in MongoDB, start by creating a `MongoClient` object, then specify a connection URL with the correct ip address and the name of the database you want to create. MongoDB will create the database if it does not exist, and make a connection to it.

### Install Node.js

First, make sure you have a supported version of Node.js installed. The current version of MongoDB Node.js Driver requires Node 4.x or greater. For these examples, I've used Node.js



14.15.4. See the MongoDB Compatibility docs for more information on which version of Node.js is required for each version of the Node.js driver.

## Install the MongoDB Node.js Driver

The MongoDB Node.js Driver allows you to easily interact with MongoDB databases from within Node.js applications. You'll need the driver in order to connect to your database and execute the queries described in this Quick Start series.

If you don't have the MongoDB Node.js Driver installed, you can install it with the following command.

```
npm install mongodb
```

At the time of writing, this installed version 3.6.4 of the driver. Running `npm list mongodb` will display the currently installed driver version number. For more details on the driver and installation, see the official documentation.

## Create a free MongoDB Atlas cluster and load the sample data

Next, you'll need a MongoDB database. The easiest way to get started with MongoDB is to use Atlas, MongoDB's fully-managed database-as-a-service.

Head over to Atlas and create a new cluster in the free tier. At a high level, a cluster is a set of nodes where copies of your database will be stored. Once your tier is created, load the sample data. If you're not familiar with how to create a new cluster and load the sample data, check out this video tutorial from MongoDB Developer Advocate Maxime Beugnet.

Get started with an M0 cluster on Atlas today. It's free forever, and it's the easiest way to try out the steps in this blog series.

## Get your cluster's connection info

The final step is to prep your cluster for connection.

In Atlas, navigate to your cluster and click **CONNECT**. The Cluster Connection Wizard will appear.

The Wizard will prompt you to add your current IP address to the IP Access List and create a MongoDB user if you haven't already done so. Be sure to note the username and password you use for the new MongoDB user as you'll need them in a later step.

Next, the Wizard will prompt you to choose a connection method. Select **Connect Your Application**. When the Wizard prompts you to select your driver version, select **Node.js** and **3.6 or later**. Copy the provided connection string.

For more details on how to access the Connection Wizard and complete the steps described above, see the official documentation.

## Connect to your database from a Node.js application

Now that everything is set up, it's time to code! Let's write a Node.js script that connects to your database and lists the databases in your cluster.

### Import MongoClient

The MongoDB module exports `MongoClient`, and that's what we'll use to connect to a MongoDB database. We can use an instance of `MongoClient` to connect to a cluster, access the database in that cluster, and close the connection to that cluster.

```
const {MongoClient} = require('mongodb');
```

### Create our main function

Let's create an asynchronous function named `main()` where we will connect to our MongoDB cluster, call functions that query our database, and disconnect from our cluster.

```
async function main() {  
    // we'll add code here soon  
}
```

The first thing we need to do inside of `main()` is create a constant for our connection URI. The connection URI is the connection string you copied in Atlas in the previous section. When you paste the connection string, don't forget to update `<username>` and `<password>` to be the credentials for the user you created in the previous section. The connection string includes a `<dbname>` placeholder. For these examples, we'll be using the `sample_airbnb` database, so replace `<dbname>` with `sample_airbnb`.

**Note:** the username and password you provide in the connection string are NOT the same as your Atlas credentials.

```
/**  
 * Connection URI. Update <username>, <password>, and <your-cluster-url> to reflect your  
 cluster.  
 * See https://docs.mongodb.com/ecosystem/drivers/node/ for more details  
 */  
const uri = "mongodb+srv://<username>:<password>@<your-cluster-  
url>/test?retryWrites=true&w=majority";
```

Now that we have our URI, we can create an instance of `MongoClient`.

```
const client = new MongoClient(uri);
```

**Note:** When you run this code, you may see DeprecationWarnings around the URL string parser and the Server Discover and Monitoring engine. If you see these warnings, you can remove them by passing options to the MongoClient. For example, you could instantiate MongoClient by calling `new MongoClient(uri, { useNewUrlParser: true, useUnifiedTopology: true })`. See the Node.js MongoDB Driver API documentation for more information on these options.

Now we're ready to use MongoClient to connect to our cluster. `client.connect()` will return a promise. We will use the `await` keyword when we call `client.connect()` to indicate that we should block further execution until that operation has completed.

```
await client.connect();
```

Now we are ready to interact with our database. Let's build a function that prints the names of the databases in this cluster. It's often useful to contain this logic in well named functions in order to improve the readability of your codebase. Throughout this series, we'll create new functions similar to the function we're creating here as we learn how to write different types of queries. For now, let's call a function named `listDatabases()`.

```
await listDatabases(client);
```

Let's wrap our calls to functions that interact with the database in a `try/catch` statement so that we handle any unexpected errors.

```
try {  
    await client.connect();  
  
    await listDatabases(client);  
  
} catch (e) {  
    console.error(e);  
}
```

We want to be sure we close the connection to our cluster, so we'll end our `try/catch` with a `finally` statement.

```
finally {  
    await client.close();  
}
```

Once we have our `main()` function written, we need to call it. Let's send the errors to the console.

```
main().catch(console.error);
```

Putting it all together, our `main()` function and our call to it will look something like the following.

```

async function main(){
  /**
   * Connection URI. Update <username>, <password>, and <your-cluster-url>
   to reflect your cluster.
   * See https://docs.mongodb.com/ecosystem/drivers/node/ for more details
   */
  const uri = "mongodb+srv://<username>:<password>@<your-cluster-
url>/test?retryWrites=true&w=majority";

  const client = new MongoClient(uri);

  try {
    // Connect to the MongoDB cluster
    await client.connect();

    // Make the appropriate DB calls
    await listDatabases(client);

  } catch (e) {
    console.error(e);
  } finally {
    await client.close();
  }
}

main().catch(console.error);

```

## List the databases in our cluster

In the previous section, we referenced the `listDatabases()` function. Let's implement it!

This function will retrieve a list of databases in our cluster and print the results in the console.

```

async function listDatabases(client){
  databasesList = await client.db().admin().listDatabases();

  console.log("Databases:");
  databasesList.databases.forEach(db => console.log(` - ${db.name}`));
};

```

## Save Your File

You've been implementing a lot of code. Save your changes, and name your file something like `connection.js`. To see a copy of the complete file, visit the [nodejs-quickstart GitHub repo](#).

## Execute Your Node.js Script

Now you're ready to test your code! Execute your script by running a command like the following in your terminal: `node connection.js`

You will see output like the following:

Databases: ♣ sample\_airbnb  
♣ sample\_geospatial  
♣ sample\_mflix  
♣ sample\_supplies  
♣ sample\_training  
♣ sample\_weatherdata  
♣ admin  
♣ local

## Node.js MongoDB Create Database

### Creating a Database

To create a database in MongoDB, start by creating a MongoClient object, then specify a connection URL with the correct ip address and the name of the database you want to create.

MongoDB will create the database if it does not exist, and make a connection to it.

### Example

Create a database called "mydb":

```
var MongoClient = require('mongodb').MongoClient;  
var url = "mongodb://localhost:27017/mydb";
```

```
MongoClient.connect(url, function(err, db) {  
  if (err) throw err;  
  console.log("Database created!");  
  db.close();  
});
```

Save the code above in a file called "demo\_create\_mongo\_db.js" and run the file:

Run "demo\_create\_mongo\_db.js"

C:\Users\Your Name>node demo\_create\_mongo\_db.js

Which will give you this result:

Database created!

([https://www.w3schools.com/nodejs/shownodejs\\_cmd.asp?filename=demo\\_create\\_mongo\\_db](https://www.w3schools.com/nodejs/shownodejs_cmd.asp?filename=demo_create_mongo_db))

**Important:** In MongoDB, a database is not created until it gets content!

MongoDB waits until you have created a collection (table), with at least one document (record) before it actually creates the database (and collection).

## Node.js MongoDB Create Collection

A **collection** in MongoDB is the same as a **table** in MySQL

### Creating a Collection

To create a collection in MongoDB, use the `createCollection()` method:

#### Example

Create a collection called "customers":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
```

```
MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.createCollection("customers", function(err, res) {
    if (err) throw err;
    console.log("Collection created!");
    db.close();
  });
});
```

Save the code above in a file called "demo\_mongodb\_createcollection.js" and run the file:

Run "demo\_mongodb\_createcollection.js"

```
C:\Users\Your Name>node demo_mongodb_createcollection.js
```

Which will give you this result:

Collection created!

**Important:** In MongoDB, a collection is not created until it gets content!

MongoDB waits until you have inserted a document before it actually creates the collection.

([https://www.w3schools.com/nodejs/shownodejs\\_cmd.asp?filename=demo\\_mongodb\\_createcollection](https://www.w3schools.com/nodejs/shownodejs_cmd.asp?filename=demo_mongodb_createcollection))

## Adding and retrieving data to MongoDB from NodeJS:-

You can use read operations to retrieve data from your MongoDB database. There are multiple types of read operations that access the data in different ways. If you want to request results based on a set of criteria from the existing set of data, you can use a find operation such as the `find()` or `findOne()` methods.

You can also further specify the information you are requesting by including additional parameters or by chaining other methods such as:

- ❖ Sort Results
- ❖ Skip Returned Results
- ❖ Limit the Number of Returned Results
- ❖ Specify Which Fields to Return

You can also use an aggregation operation to retrieve data. This type of operation allows you to apply an ordered pipeline of transformations to the matched data.

If you want to monitor the database for incoming data that matches a set of criteria, you can use the watch operation to be notified in real-time when matching data is inserted.

Note:-

Your query operation may return a reference to a cursor that contains matching documents. To learn how to examine data stored in the cursor, see the Cursor Fundamentals page.

### Find:

The `find()` method is called on the `Collection` object that references the collection you want to query. The method accepts a query document that describes the documents you want to retrieve. For more information on how to specify your query document, see our guide on how to Specify a Query.

To access the results, you can optionally pass a callback in the method call or resolve the returned `Promise` object. See our guide on Promises and Callbacks for more information.

If you resolve the `Promise` returned by `find()`, you receive a reference to a `Cursor` with which you can navigate matched documents. If you resolve the `Promise` returned by `findOne()`, you receive the matching document or `null` if there are no matches.

### Example:-

A pizza restaurant wants to find all pizzas ordered by Lemony Snicket yesterday. They run the following `find()` query on the `orders` collection:

```
const findResult = await orders.find({
  name: "Lemony Snicket",
  date: {
    $gte: new Date(new Date().setHours(00, 00, 00)),
    $lt: new Date(new Date().setHours(23, 59, 59)),
  },
});
```

Once the operation returns, the `findResult` variable references a `Cursor`. You can print the documents retrieved using the `forEach()` method as shown below:

```
await cursor.forEach(console.dir);
```

The output might resemble the following:

```
[
  { name: "Lemony Snicket", type: "horseradish pizza", qty: 1, status: "delivered", date: ... },
  { name: "Lemony Snicket", type: "coal-fired oven pizza", qty: 3, status: "canceled", date: ... },
  ...
]
```

See the `find()` and `findOne()` for fully-runnable examples.

### **Aggregate:-**

If you want to run a custom processing pipeline to retrieve data from your database, you can use the `aggregate()` method. This method accepts aggregation expressions to run in sequence. These expressions let you filter, group, and arrange the result data from a collection.

### **Example**

A pizza restaurant wants to run a status report on-demand to summarize pizza orders over the past week. They run the following `aggregate()` query on the `orders` collection to fetch the totals for each distinct "status" field:

```
const aggregateResult = await orders.aggregate([
  {
    $match: {
```



```

date: {
  $gte: new Date(new Date().getTime() - 1000 * 3600 * 24 * 7),
  $lt: new Date(),
},
},
},
{
  $group: {
    _id: "$status",
    count: {
      $sum: 1,
    },
  },
},
]);

```

Once the operation returns, the `aggregateResult` variable references a `Cursor`. You can print the documents retrieved using the `forEach()` method as shown below:

```
await cursor.forEach(console.dir);
```

The output might resemble the following:

```

[
  { _id: 'delivering', count: 5 },
  { _id: 'delivered', count: 37 },
  { _id: 'created', count: 9 }
]

```

See the MongoDB server manual pages on aggregation for more information on how to construct an aggregation pipeline.

## Watch / Subscribe

You can use the `watch()` method to monitor a collection for changes to a collection that match certain criteria. These changes include inserted, updated, replaced, and deleted documents. You can pass this method a pipeline of aggregation commands that sequentially runs on the changed data whenever write operations are executed on the collection.

## Example

A pizza restaurant wants to receive a notification whenever a new pizza order comes in. To accomplish this, they create an aggregation pipeline to filter on insert operations and return specific fields. They pass this pipeline to the `watch()` method called on the `orders` collection as shown below:

```
const changeStream = orders.watch([
  { $match: { operationType: "insert" } },
  {
    $project: {
      "fullDocument.name": 1,
      "fullDocument.address": 1,
    },
  },
]);
changeStream.on("change", change => {
  const { name, address } = change.fullDocument;
  console.log(`New order for ${name} at ${address}.`);
});
```

For a runnable example of the `watch()` method using the NodeJS driver, see the change streams usage example.

## Handling SQL databases from NodeJS:-

Node.js can be used in database applications.

One of the most popular databases is MySQL.

## MySQL Database:-

To be able to experiment with the code examples, you should have MySQL installed on your computer.

You can download a free MySQL database at <https://www.mysql.com/downloads/>.

## Install MySQL Driver:-

Once you have MySQL up and running on your computer, you can access it by using Node.js.

To access a MySQL database with Node.js, you need a MySQL driver. This tutorial will use the "mysql" module, downloaded from NPM.

To download and install the "mysql" module, open the Command Terminal and execute the following:

```
C:\Users\Your Name>npm install mysql
```

Now you have downloaded and installed a mysql database driver.

Node.js can use this module to manipulate the MySQL database:

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

### **Create Connection:-**

Start by creating a connection to the database.

Use the username and password from your MySQL database.

```
demo_db_connection.js
```

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

```
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword"  
});
```

```
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
});
```

([https://www.w3schools.com/nodejs/shownodejs\\_cmd.asp?filename=demo\\_create\\_db](https://www.w3schools.com/nodejs/shownodejs_cmd.asp?filename=demo_create_db))

Save the code above in a file called "demo\_db\_connection.js" and run the file:

Run "demo\_db\_connection.js"

```
C:\Users\Your Name>node demo_db_connection.js
```

Which will give you this result:

Connected!

Now you can start querying the database using SQL statements.

## Query a Database:-

Use SQL statements to read from (or write to) a MySQL database. This is also called "to query" the database.

The connection object created in the example above, has a method for querying the database:

```
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
  con.query(sql, function (err, result) {  
    if (err) throw err;  
    console.log("Result: " + result);  
  });  
});
```

The query method takes an sql statements as a parameter and returns the result.

Learn how to read, write, delete, and update a database in the next chapters.

## Creating a Table:-

To create a table in MySQL, use the "CREATE TABLE" statement.

Make sure you define the name of the database when you create the connection:

### Example:-

Create a table named "customers":

```
var mysql = require('mysql');  
  
var con = mysql.createConnection({  
  host: "localhost",  
  user: "yourusername",  
  password: "yourpassword",  
  database: "mydb"  
});  
  
con.connect(function(err) {  
  if (err) throw err;  
  console.log("Connected!");  
  var sql = "CREATE TABLE customers (name VARCHAR(255), address  
VARCHAR(255))";
```

```
con.query(sql, function (err, result) {  
  if (err) throw err;  
  console.log("Table created");  
});  
});
```

Save the code above in a file called "demo\_create\_table.js" and run the file:

Run "demo\_create\_table.js"

C:\Users\Your Name>node demo\_create\_table.js

Which will give you this result:

Connected!  
Table created

([https://www.w3schools.com/nodejs/shownodejs\\_cmd.asp?filename=demo\\_create\\_table](https://www.w3schools.com/nodejs/shownodejs_cmd.asp?filename=demo_create_table))

### Handling Cookies in NodeJS:-

Cookies are simple, small files/data that are sent to client with a server request and stored on the client side. Every time the user loads the website back, this cookie is sent with the request. This helps us keep track of the user's actions.

The following are the numerous uses of the HTTP Cookies –

- ✓ Session management
- ✓ Personalization(Recommendation systems)
- ✓ User tracking

To use cookies with Express, we need the cookie-parser middleware. To install it, use the following code –

```
npm install --save cookie-parser
```

Now to use cookies with Express, we will require the **cookie-parser**. cookie-parser is a middleware which parses cookies attached to the client request object. To use it, we will require it in our **index.js** file; this can be used the same way as we use other middleware. Here, we will use the following code.

```
var cookieParser = require('cookie-parser');  
app.use(cookieParser());
```

cookie-parser parses Cookie header and populates **req.cookies** with an object keyed by the cookie names. To set a new cookie, let us define a new route in your Express app like –

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

app.get('/', function(req, res){
  res.cookie('name', 'express').send('cookie set'); //Sets name = express
});

app.listen(3000);
```

To check if your cookie is set or not, just go to your browser, fire up the console, and enter –

```
console.log(document.cookie);
```

You will get the output like (you may have more cookies set maybe due to extensions in your browser) –

```
"name = express"
```

The browser also sends back cookies every time it queries the server. To view cookies from your server, on the server console in a route, add the following code to that route.

```
console.log('Cookies: ', req.cookies);
```

Next time you send a request to this route, you will receive the following output.

```
Cookies: { name: 'express' }
```

## Adding Cookies with Expiration Time

You can add cookies that expire. To add a cookie that expires, just pass an object with property 'expire' set to the time when you want it to expire. For example,

```
//Expires after 360000 ms from the time it is set.
res.cookie(name, 'value', {expire: 360000 + Date.now()});
```

Another way to set expiration time is using '**maxAge**' property. Using this property, we can provide relative time instead of absolute time. Following is an example of this method.

```
//This cookie also expires after 360000 ms from the time it is set.
res.cookie(name, 'value', {maxAge: 360000});
```

## Deleting Existing Cookies

To delete a cookie, use the clearCookie function. For example, if you need to clear a cookie named **foo**, use the following code.

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

```
app.get('/clear_cookie_foo', function(req, res){
  res.clearCookie('foo');
  res.send('cookie foo cleared');
});

app.listen(3000);
```

## What are cookies?

A cookie is usually a tiny text file stored in your web browser. A cookie was initially used to store information about the websites that you visit. But with the advances in technology, a cookie can track your web activities and retrieve your content preferences.

The different types of cookies include:

- ✓ **Session cookies** - store user's information for a short period. When the current session ends, that session cookie is deleted from the user's computer.
- ✓ **Persistent cookies** - a persistent cookie lacks expiration date. It is saved as long as the webserver administrator sets it.
- ✓ **Secure cookies** - are used by encrypted websites to offer protection from any possible threats from a hacker.
- ✓ **Third-party cookies** - are used by websites that show ads on their pages or track website traffic. They grant access to external parties to decide the types of ads to show depending on the user's previous preferences.

## The main difference between a session and a cookie

The major difference between sessions and cookies is that sessions live on the server-side (the webserver), and cookies live on the client-side (the user browser). Sessions have sensitive information such as usernames and passwords. This is why they are stored on the server. Sessions can be used to identify and validate which user is making a request.

## Setting up cookies with Node.js

Let's dive in and see how we can implement cookies using Node.js. We will create and save a cookie in the browser, update and delete a cookie.

We will use the following NPM packages:

- ✓ **Express** - this is an opinionated server-side framework for Node.js that helps you create and manage HTTP server REST endpoints.
- ✓ **cookie-parser** - cookie-parser looks at the headers in between the client and the server transactions, reads these headers, parses out the cookies being sent, and saves them in a browser. In other words, cookie-parser will help us create and manage cookies depending on the request a user makes to the server.

**Run the following command to install these NPM packages:**

```
npm install express cookie-parser
```

We will create a simple example to demonstrate how cookies work.

### Step 1 - Import the installed packages

To set up a server and save cookies, import the cookie parser and express modules to your project. This will make the necessary functions and objects accessible.

```
const express = require('express')
const cookieParser = require('cookie-parser')
```

Step  
- 2

### Get your application to use the packages

You need to use the above modules as middleware inside your application, as shown below.

```
//setup express app
const app = express()

// let's you use the cookieParser in your application
app.use(cookieParser());
```

This  
will  
make  
your

application use the cookie parser and Express modules.

### Step - 3 Set a simple route to start the server

We use the following code to set up a route for the homepage:

```
//set a simple for homepage route
app.get('/', (req, res) => {
  res.send('welcome to a simple HTTP cookie server');
});
```

Step  
4 -  
Set a

### port number

This is the port number that the server should listen to when it is running. This will help us access our server locally. In this example, the server will listen to port 8000, as shown below.

```
//server listening to port 8000
app.listen(8000, () => console.log('The server is running port 8000...'));
```

Now

we have a simple server set. Run `node app.js` to test if it is working.

```
$ Node app.js
The server is running port 8000...
█
```



And if you access the localhost on port 8000 (<http://localhost:8000/>), you should get an HTTP response sent by the server. Now we're ready to start implementing cookies.

## Setting cookies

Let's add routes and endpoints that will help us create, update and delete a cookie.

### Step 1 - Set a cookie

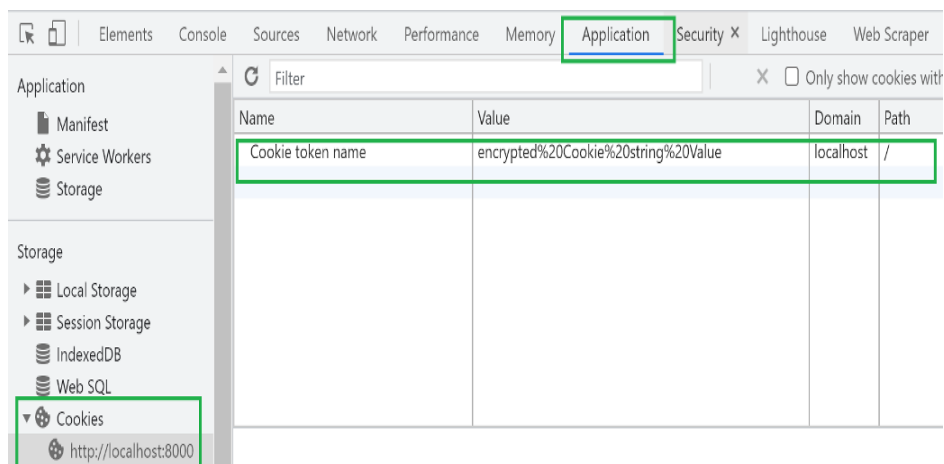
We will set a route that will save a cookie in the browser. In this case, the cookies will be coming from the server to the client browser. To do this, use the `res` object and pass `cookie` as the method, i.e. `res.cookie()` as shown below.

```
//a get route for adding a cookie
app.get('/setcookie', (req, res) => {
  res.cookie('Cookie token name', 'encrypted cookie string Value');
  res.send('Cookie have been saved successfully');
});
```

When the above route is executed from a browser, the client sends a get request to the server. But in this case, the server will respond with a cookie and save it in the browser.

Go ahead and run `node app.js` to serve the above endpoint. Open <http://localhost:8000/getcookie> your browser and access the route.

To confirm that the cookie was saved, go to your browser's inspector tool select the application tab cookies select your domain URL.



## Step 2 - Using the req.cookies method to check the saved cookies

If the server sends this cookie to the browser, this means we can iterate the incoming requests through `req.cookies` and check the existence of a saved cookie. You can log this cookie to the console or send the cookie request as a response to the browser. Let's do that.

```
// get the cookie incoming request
app.get('/getcookie', (req, res) => {
  //show the saved cookies
  console.log(req.cookies)
  res.send(req.cookies);
});
```

Again run the server using `node app.js` to expose the above route (`http://localhost:8000/getcookie`) and you can see the response on the browser.

---

```
{"Cookie token name":"encrypted Cookie string Value"}
```

As well as on your console logs.

```
$ node app.js
The server is running port 8000...
{ 'Cookie token name': 'encrypted Cookie string Value' }
█
```

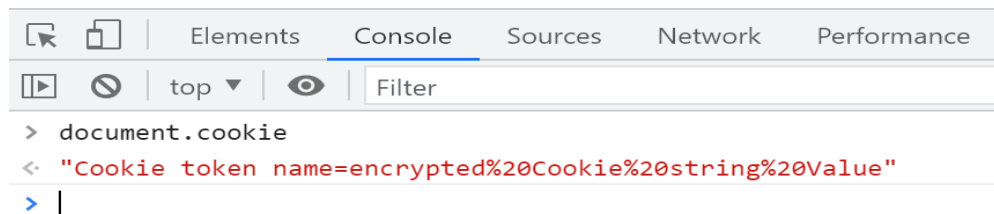
## Step 3 - Secure cookies

One precaution that you should always take when setting cookies is security. In the above example, the cookie can be deemed insecure.

For example, you can access this cookie on a browser console using JavaScript (`document.cookie`). This means that this cookie is exposed and can be exploited through cross-site scripting. You can see the cookie when you open the browser inspector tool and execute the following in the console.

```
document.cookie
```

The saved cookie values can be seen through the browser console.



As a precaution, you should always try to make your cookies inaccessible on the client-side using JavaScript.

We can add several attributes to make this cookie more secure.

- ❖ HTTPOnly ensures that a cookie is not accessible using the JavaScript code. This is the most crucial form of protection against cross-scripting attacks.
- ❖ A secure attribute ensures that the browser will reject cookies unless the connection happens over HTTPS.
- ❖ sameSite attribute improves cookie security and avoids privacy leaks.

By default, `sameSite` was initially set to `none` (`sameSite = None`). This allowed third parties to track users across sites. Currently, it is set to `Lax` (`sameSite = Lax`) meaning a cookie is only set when the domain in the URL of the browser matches the domain of the cookie, thus eliminating third party's domains. `sameSite` can also be set to `Strict` (`sameSite = Strict`). This will restrict cross-site sharing even between different domains that the same publisher owns.

## Handling User Authentication with NodeJS:-

Creating a user registration form employs the management of the registered user. This is where user role authentication comes into play. Role authentication ensures that non-admin users cannot make changes or access exclusive information. It grants administrative privileges to admin users and basic privileges to basic users.

You can build your own authentication functionality with web tokens like JSON Web Token (JWT) or use a trusted third-party customer identity and access management (CIAM) software like LoginRadius.

## Goal:-

This tutorial helps you:

- ✓ understand the differences between the Admin role and the Basic user role;
- ✓ use JWT to authenticate users; and
- ✓ learn role-based authentication using JWT in a simple Node.js app.

## Prerequisites

You have installed the following:

- Node
- MongoDB
- Text Editor

You already understand JavaScript E56 Syntax.

Now that everything is in place, let's set up your database.

## Set Up a Mongo Database

You'll store all your user data — which includes username, password, and role — in MongoDB.

Install a node package called Mongoose that will connect to MongoDB. Then create a user schema for your application.

```
npm init
npm install mongoose
```

`npm init` sets up your new project and creates a `package.json` file with the credentials.

After installing mongoose, create a new file `db.js` in the project's directory and require mongoose.

```
const Mongoose = require("mongoose")
```

With the help of mongoose, you can connect your application to MongoDB:

```
// db.js
const Mongoose = require("mongoose")
const localDB = `mongodb://localhost:27017/role_auth`
const connectDB = async () => {
  await Mongoose.connect(localDB, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
  })
  console.log("MongoDB Connected")
}
module.exports = connectDB
```

The code snippet here connects to `mongodb://localhost:27017` and then specifies the name of the database `/role_auth`.

The function `connectDB` awaits for the connection, which contains the `URI` and `options` as a second parameter. If it connects without errors, it will log out `MongoDB Connected`. Error issues will be fixed while connecting to the database. After this, it exported the function for use in the server.

## Set Up the Server

You need to install some dependencies that you'll use in this tutorial.

```
npm i express nodemon
```

Express.js is a Node.js framework for building web applications quickly and easily.

Nodemon is a tool that watches the file system and automatically restarts the server when there is a change.

You require `express` in your application to listen for a connection on port `5000`. Create a new file `server.js` in the root directory and create the listening event:

```
const express = require("express")
const app = express()
const PORT = 5000
app.listen(PORT, () => console.log(`Server Connected to port ${PORT}`))
```

The next step is to test your application. Open up your `package.json` file and add the following to `scripts`:

```
"scripts": {
  "start": "node server.js",
  "dev": "nodemon server.js"
}
```

Open your terminal and run `npm run dev` to start the server.

## Connect to the Database

Earlier, you've created a function that connects to MongoDB and exported that function. Now import that function into your `server.js`:

```
const connectDB = require("../db");
...
//Connecting the Database
connectDB();
```

You also need to create an error handler that catches every `unhandledRejection` error.

```
const server = app.listen(PORT, () =>
  console.log(`Server Connected to port ${PORT}`)
)
```

```
// Handling Error
process.on("unhandledRejection", err => {
  console.log(`An error occurred: ${err.message}`)
  server.close(() => process.exit(1))
})
```

The listening event is assigned to a constant `server`. If an `unhandledRejection` error occurs, it logs out the error and closes the `server` with an exit code of 1.

## Create User Schema:-

Schema is like a blueprint that shows how the database will be constructed.

You'll structure a user schema that contains username, password, and role.

Create a new folder `model` in the project's directory, and create a file called `User.js`. Now open `User.js` and create the user schema:

```
// user.js
const Mongoose = require("mongoose")
const UserSchema = new Mongoose.Schema({
  username: {
    type: String,
    unique: true,
    required: true,
  },
  password: {
    type: String,
    minlength: 6,
    required: true,
  },
  role: {
    type: String,
    default: "Basic",
    required: true,
  },
})
```

In the schema, the `username` will be unique, required, and will accept strings.

You've specified the minimum characters(6) the `password` field will accept. The `role` field grants a default value (basic) that you can change if needed.

Now, you need to create a user model and export it:

```
const User = Mongoose.model("user", UserSchema)
module.exports = User
```

You've created the user model by passing the `UserSchema` as the second argument while the first argument is the name of the model `user`.

## Perform CRUD Operations:-

You'll create functions that handle:

- ✓ adding users;
- ✓ getting all users;
- ✓ updating the role of users; and,
- ✓ deleting users.

## Register Function:-

As the name implies, this function will handle the registrations of users.

Let's create a new folder named `Auth`. It will contain the Authentication file and the Route set-up file.

After creating the `Auth` folder, add two files — `Auth.js` and `Route.js`.

Now open up our `Auth.js` file and import that `User` model:

```
const User = require("../model/User")
```

The next step is to create an `async express` function that will take the user's data and register it in the database.

You need to use an Express middleware function that will grant access to the user's data from the body. You'll use this function in the `server.js` file:

```
const app = express()  
app.use(express.json())
```

Let's go back to your `Auth.js` file and create the register function:

```
// auth.js  
exports.register = async (req, res, next) => {  
  const { username, password } = req.body  
  if (password.length < 6) {  
    return res.status(400).json({ message: "Password less than 6 characters"  
  })  
  }  
  try {  
    await User.create({  
      username,  
      password,  
    }).then(user =>  
      res.status(200).json({  
        message: "User successfully created",  
        user,  
      })  
    )  
  }  
}
```

```

    } catch (err) {
      res.status(401).json({
        message: "User not successful created",
        error: error.mesage,
      })
    }
  }
}

```

The exported `register` function will be used to set up the routes. You got the username and password from the `req.body` and created a `tryCatch` block that will create the user if successful; else, it returns status code 401 with the error message.

## Set Up Register Route

You'll create a route to `/register` using `express.Router`. Import the `register` function into your `route.js` file, and use it as the route's function:

```

const express = require("express")
const router = express.Router()
const { register } = require("../auth")
router.route("/register").post(register)
module.exports = router

```

The last step is to import your `route.js` file as middleware in `server.js`:

```
app.use("/api/auth", require("../Auth/route"))
```

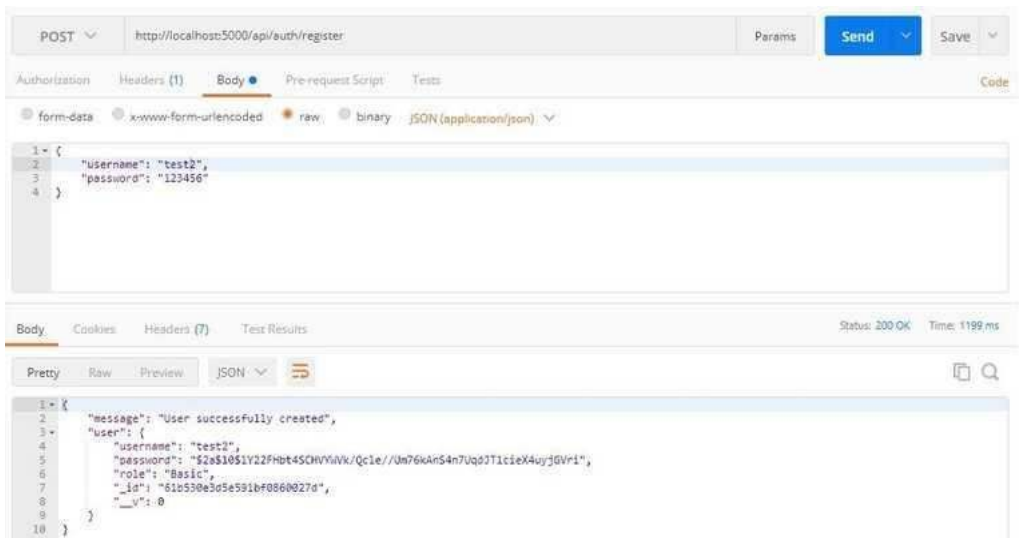
The server will use the `router` middleware function if there is a request to `/api/auth`.

## Test the Register Route

You'll use Postman to test all the routes.

Open up Postman to send a `POST` request to `http://localhost:5000/api/auth/register` and pass the username and password to the body:





## Login Function

You've created a function that adds registered users to the database. You have to create another function that will authenticate user credentials and check if the user is registered.

Open the `Auth.js` file and create the Login function, as follows:

```
// auth.js
exports.login = async (req, res, next) => {
  const { username, password } = req.body
  // Check if username and password is provided
  if (!username || !password) {
    return res.status(400).json({
      message: "Username or Password not present",
    })
  }
}
```

The `login` function returns status code 400 if the username and password were not provided. You need to find a user with the provided username and password:

```
exports.login = async (req, res, next) => {
  try {
    const user = await User.findOne({ username, password })
    if (!user) {
      res.status(401).json({
        message: "Login not successful",
        error: "User not found",
      })
    } else {
      res.status(200).json({
        message: "Login successful",
        user,
      })
    }
  }
}
```

```

    } catch (error) {
      res.status(400).json({
        message: "An error occurred",
        error: error.message,
      })
    }
  }
}

```

Here, it returns status code 401 when a user isn't found and 200 when a user is found. The code snippet wrapped all this in a `tryCatch` block to detect and output errors, if any.

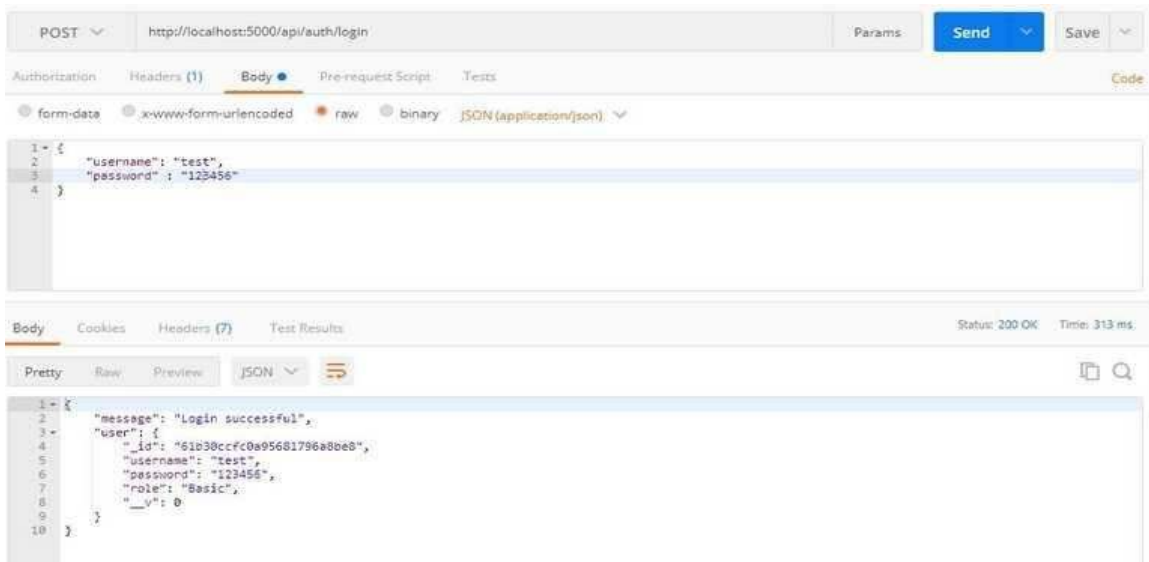
### Set Up Login Route:-

To set up the login route, import the `login` function into your `route.js`:

```
const express = require("express");
const router = express.Router();
const { register, login } = require("../auth");
...
router.route("/login").post(login);
module.exports = router;
```

### Test the Login Route:-

Make a POST request at `http://localhost:5000/api/auth/login` and pass a valid username and password to the body:



### Update Function:-

This function will be responsible for updating the role of a basic user to an admin user. Open the `auth.js` file and create the `update` function, as follows:

```
//auth.js
exports.update = async (req, res, next) => {
  const { role, id } = req.body
  // Verifying if role and id is presnt
  if (role && id) {
    // Verifying if the value of role is admin
    if (role === "admin") {
      await User.findById(id)
    } else {
      res.status(400).json({
        message: "Role is not admin",
      })
    }
  } else {
    res.status(400).json({ message: "Role or Id not present" })
  }
}
```

The first `if` statement verifies if `role` and `id` are present in the request body.

The second `if` statement checks if the value of `role` is `admin`. You should do this to avoid having over two roles.

After finding a user with that ID, you'll create a third `if` block that will check for the role of the user:

```
exports.update = async (req, res, next) => {
  const { role, id } = req.body;
  // First - Verifying if role and id is presnt
  if (role && id) {
    // Second - Verifying if the value of role is admin
    if (role === "admin") {
      // Finds the user with the id
      await User.findById(id)
        .then((user) => {
          // Third - Verifies the user is not an admin
          if (user.role !== "admin") {
            user.role = role;
            user.save((err) => {
              //Monogodb error checker
              if (err) {
                res
                  .status("400")
                  .json({ message: "An error occurred", error: err.message });
              }
            });
            process.exit(1);
          }

          res.status("201").json({ message: "Update successful", user });
        })
        .catch((error) => {
          res
            .status(400)
            .json({ message: "User is already an Admin" });
        });
    } else {
      res.status(400).json({ message: "User is already an Admin" });
    }
  }
}
```

```

        .status(400)
        .json({ message: "An error occurred", error: error.message });
    });

    ...

```

The third `if` block prevents assigning an admin role to an admin user, while the last `if` block checks if an error occurred when saving the role in the database.

The numerous `if` statements might be a little bit tricky but understandable. Please read the comments in the above code block for better understanding.

## Set Up Update Route

Import the update function in your `route.js`, as follows:

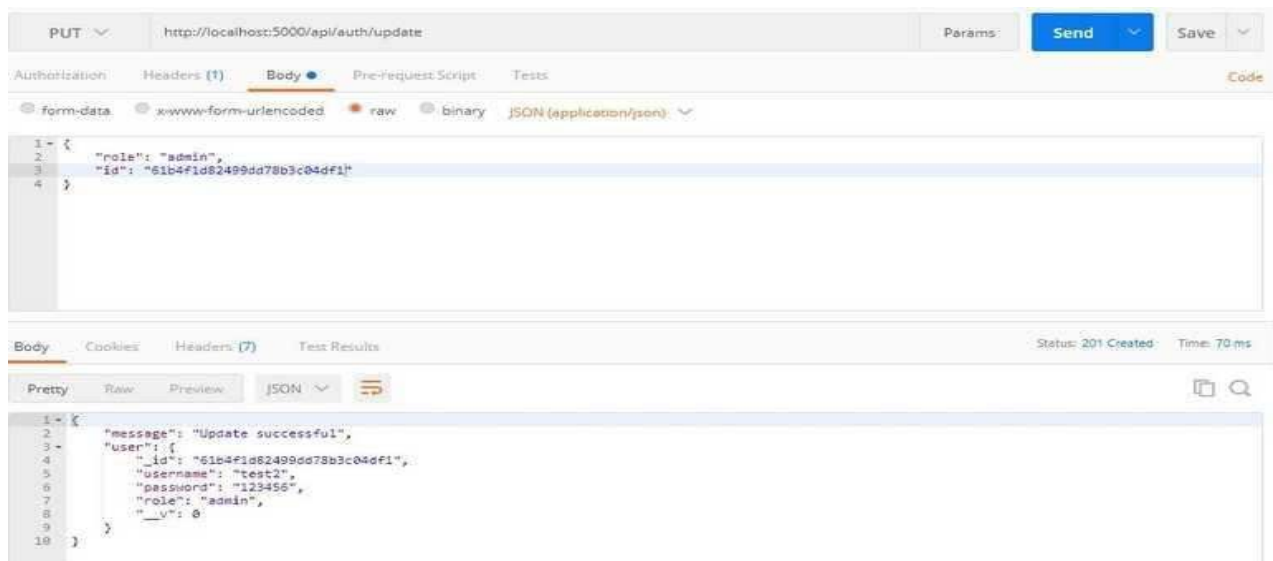
```

const { register, login, update } = require("./auth");
...
router.route("/update").put(update);

```

## Testing the Update Route

Send a `put` request to `http://localhost:5000/api/auth/update`:



## Delete Function

The `deleteUser` function will remove a specific user from the database. Let's create this function in our `auth.js` file:

```

exports.deleteUser = async (req, res, next) => {
  const { id } = req.body
  await User.findById(id)

```

```

    .then(user => user.remove())
    .then(user =>
      res.status(201).json({ message: "User successfully deleted", user })
    )
    .catch(error =>
      res
        .status(400)
        .json({ message: "An error occurred", error: error.message })
    )
  }
}

```

You remove the user based on the `id` you get from `req.body`.

## Set up the deleteUser Route

Open your `route.js` file to create a delete request to `/deleteUser`, using the `deleteUser` as its function:

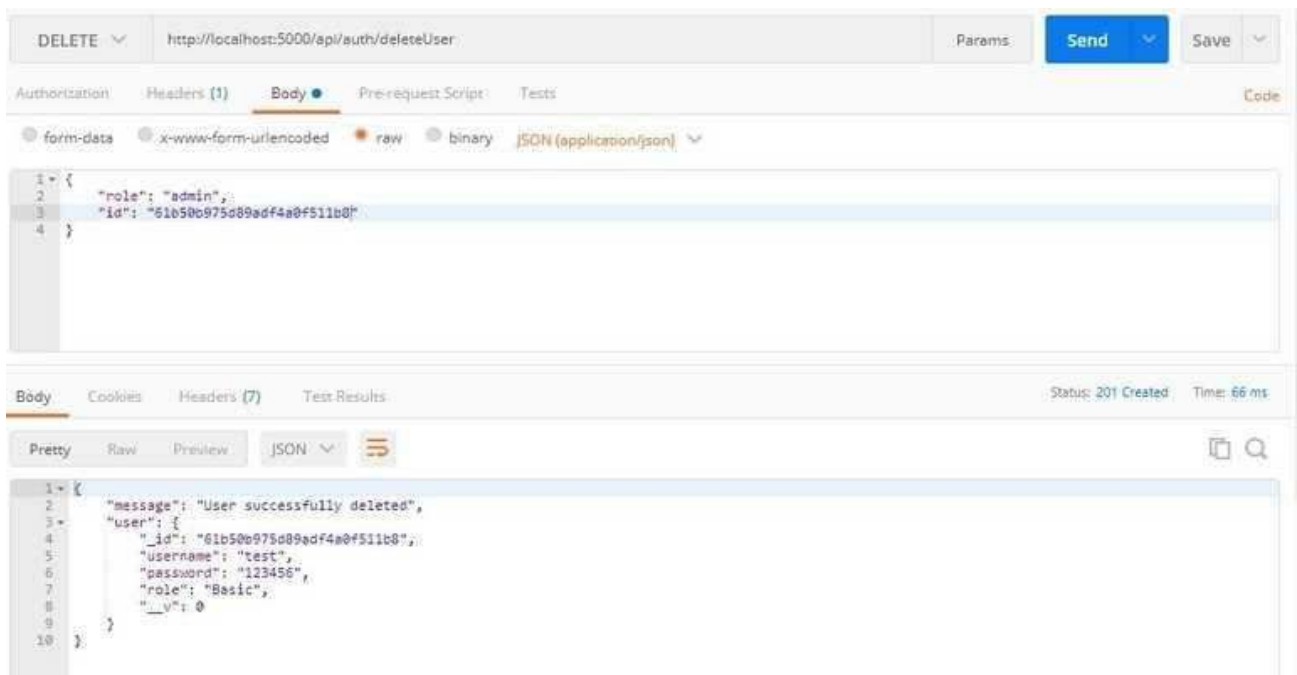
```

const { register, login, update, deleteUser } = require("../auth");
...
router.route("/deleteUser").delete(deleteUser);

```

## Test the deleteUser Route

Send a delete request to `http://localhost:5000/api/auth/deleteUser` by passing a valid `id` to the body:



## Hash User Passwords:-

Saving user passwords in the database in plain text format is reckless. It is preferable to hash your password before storing it.

For instance, it will be tough to decipher the passwords in your database if they are leaked. Hashing passwords is a cautious and reliable practice.

Let's use `bcryptjs` to hash your user passwords.

Lets install `bcryptjs`:

```
npm i bcryptjs
```

After installing `bcryptjs`, import it into your `auth.js`

```
const bcrypt = require("bcryptjs")
```

## Refactor Register Function

Instead of sending a plain text format to your database, lets hash the password using `bcrypt`:

```
exports.register = async (req, res, next) => {
  const { username, password } = req.body;

  ...

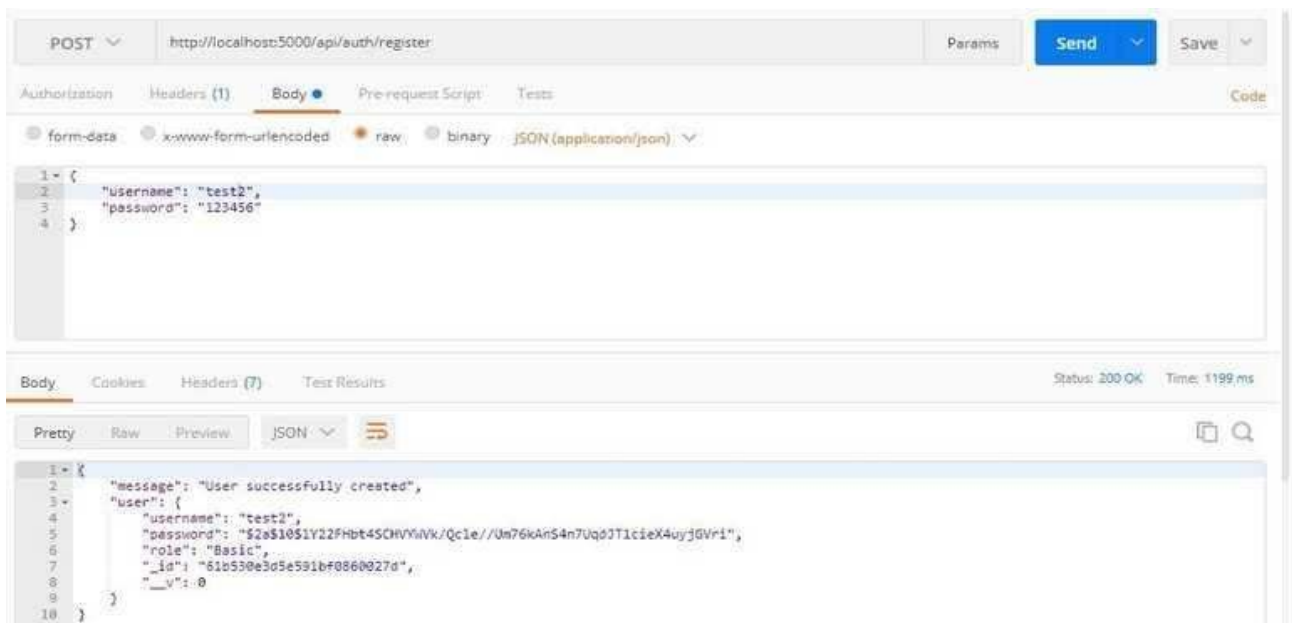
  bcrypt.hash(password, 10).then(async (hash) => {
    await User.create({
      username,
      password: hash,
    })
    .then((user) =>
      res.status(200).json({
        message: "User successfully created",
        user,
      })
    )
    .catch((error) =>
      res.status(400).json({
        message: "User not successful created",
        error: error.message,
      })
    );
  });
};
```

`bcrypt` takes in your password as the first argument and the number of times it will hash the password as the second argument. Passing a large number will take `bcrypt` a long time to hash the password, so give a moderate number like 10.

bcrypt will return a promise with the hashed password; then, send that hashed password to the database.

## Test the Register Function

Send a POST request to `http://localhost:5000/api/auth/register` and pass the username and password to the body:



## Refactor the Login Function

```
exports.login = async (req, res, next) => {
  const { username, password } = req.body
  // Check if username and password is provided
  if (!username || !password) {
    return res.status(400).json({
      message: "Username or Password not present",
    })
  }
  try {
    const user = await User.findOne({ username })
    if (!user) {
      res.status(400).json({
        message: "Login not successful",
        error: "User not found",
      })
    } else {
      // comparing given password with hashed password
      bcrypt.compare(password, user.password).then(function (result) {
        result
          ? res.status(200).json({
              message: "Login successful",
              user,
            })
          :
      })
    }
  }
}
```

```

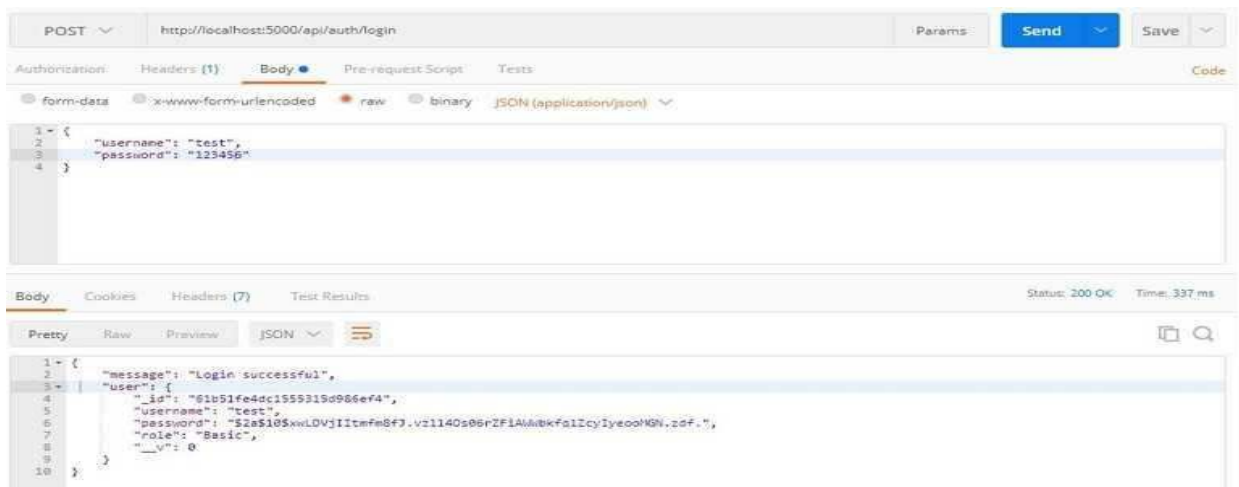
        : res.status(400).json({ message: "Login not succesful" })
    })
}
} catch (error) {
    res.status(400).json({
        message: "An error occurred",
        error: error.message,
    })
}
}
}

```

`bcrypt.compare` checks if the given password and the hashed password in the database are the same.

## Test the Login Function

Send a POST request to `http://localhost:5000/api/auth/login` and pass a valid username and password to the body:





## UNIT IV

## ADVANCED CLIENT SIDE PROGRAMMING

### 9

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

#### React JS:

- ❖ React is a JavaScript library for building user interfaces.
- ❖ React is used to build single-page applications.
- ❖ React allows us to create reusable UI components.

#### What is React?

React, sometimes referred to as a frontend JavaScript framework, is a JavaScript library created by Facebook.

React is a tool for building UI components.

#### How does React Work?

- ✓ React creates a VIRTUAL DOM in memory.
- ✓ Instead of manipulating the browser's DOM directly, React creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM.
- ✓ React only changes what needs to be changed!
- ✓ React finds out what changes have been made, and changes **only** what needs to be changed.
- ✓ You will learn the various aspects of how React does this in the rest of this tutorial.

#### React.JS History:-

- ✓ Current version of React.JS is V18.0.0 (April 2022).
- ✓ Initial Release to the Public (V0.3.0) was in July 2013.
- ✓ React.JS was first used in 2011 for Facebook's Newsfeed feature.
- ✓ Facebook Software Engineer, Jordan Walke, created it.
- ✓ Current version of create-react-app is v5.0.1 (April 2022).
- ✓ create-react-app includes built tools such as webpack, Babel, and ESLint.
- ✓ To use React in production, you need npm which is included with Node.js.
- ✓ To get an overview of what React is, you can write React code directly in HTML.
- ✓ But in order to use React in production, you need npm and Node.js installed.

#### React Directly in HTML:-

The quickest way start learning React is to write React directly in your HTML files.

Start by including three scripts, the first two let us write React code in our JavaScripts, and the third, Babel, allows us to write JSX syntax and ES6 in older browsers.

You will learn more about JSX in the React JSX chapter.

### Example:-

Include three CDN's in your HTML file:

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://unpkg.com/react@18/umd/react.development.js"
crossorigin></script>
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
crossorigin></script>
    <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>
  <body>

    <div id="mydiv"></div>

    <script type="text/babel">
      function Hello() {
        return <h1>Hello World!</h1>;
      }

      ReactDOM.render(<Hello />, document.getElementById('mydiv'))
    </script>

  </body>
</html>
```

This way of using React can be OK for testing purposes, but for production you will need to set up a

**React [w3schools CERTIFIED . 2022](https://www.w3schools.com/REACT/tryit.asp?filename=tryreact_getstarted_class)**

([https://www.w3schools.com/REACT/tryit.asp?filename=tryreact\\_getstarted\\_class](https://www.w3schools.com/REACT/tryit.asp?filename=tryreact_getstarted_class))

### Get Certified!

Complete the React modules, do the exercises, take the exam and become w3schools certified!!

### Setting up a React Environment

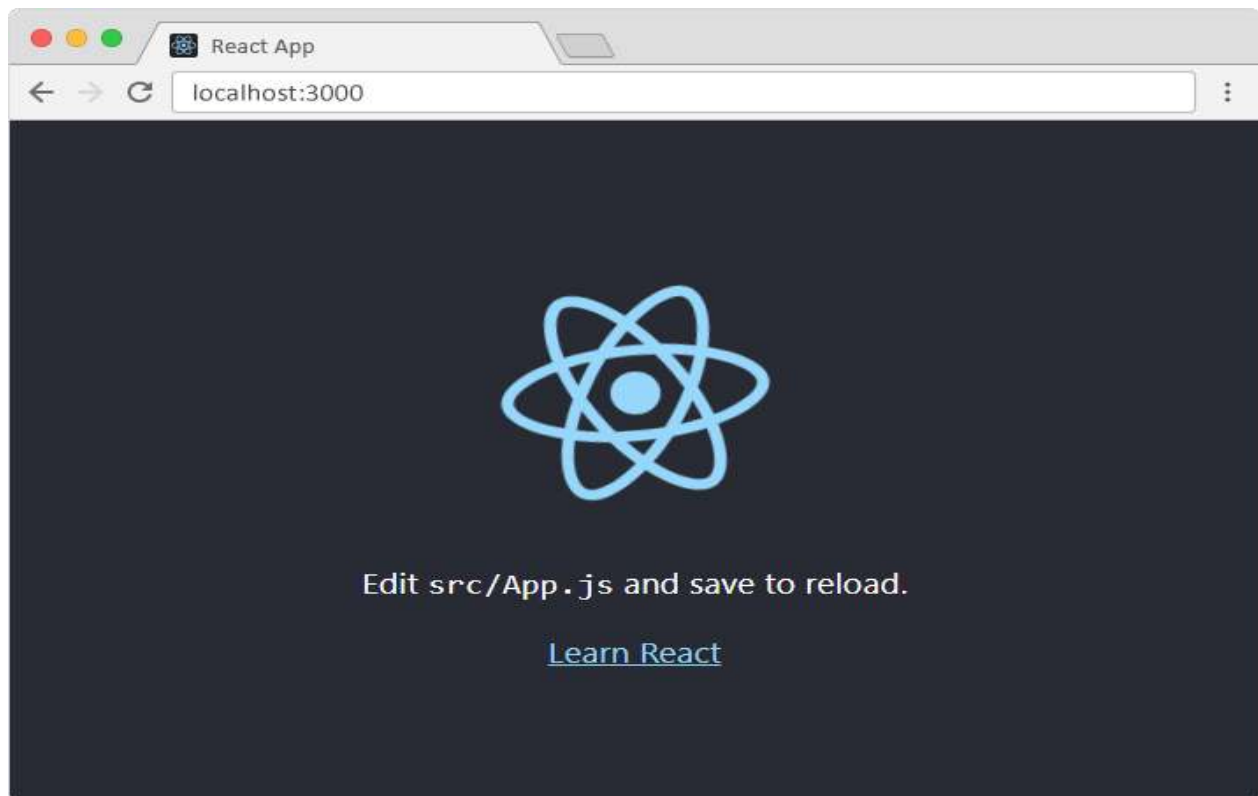
- ✓ If you have npx and Node.js installed, you can create a React application by using create-react-app.
- ✓ If you've previously installed create-react-app globally, it is recommended that you uninstall the package to ensure npx always uses the latest version of create-react-app.
- ✓ To uninstall, run this command: `npm uninstall -g create-react-app`.
- ✓ Run this command to create a React application named my-react-app:

- ✓ `npx create-react-app my-react-app`
- ✓ The create-react-app will set up everything you need to run a React application.

### Run the React Application:-

- ✓ Now you are ready to run your first *real* React application!
- ✓ Run this command to move to the my-react-app directory:
- ✓ `cd my-react-app`
- ✓ Run this command to run the React application my-react-app:
- ✓ `npm start`
- ✓ A new browser window will pop up with your newly created React App! If not, open your browser and type `localhost:3000` in the address bar.

### The result:-



### Modify the React Application:-

So far so good, but how do I change the content?

Look in the `my-react-app` directory, and you will find a `src` folder. Inside the `src` folder there is a file called `App.js`, open it and it will look like this:

`/myReactApp/src/App.js:`

```

import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;

```

Try changing the HTML content and save the file.

Notice that the changes are visible immediately after you save the file, you do not have to reload the browser!

### Example:-

Replace all the content inside the `<div className="App">` with a `<h1>` element.

See the changes in the browser when you click Save.

```

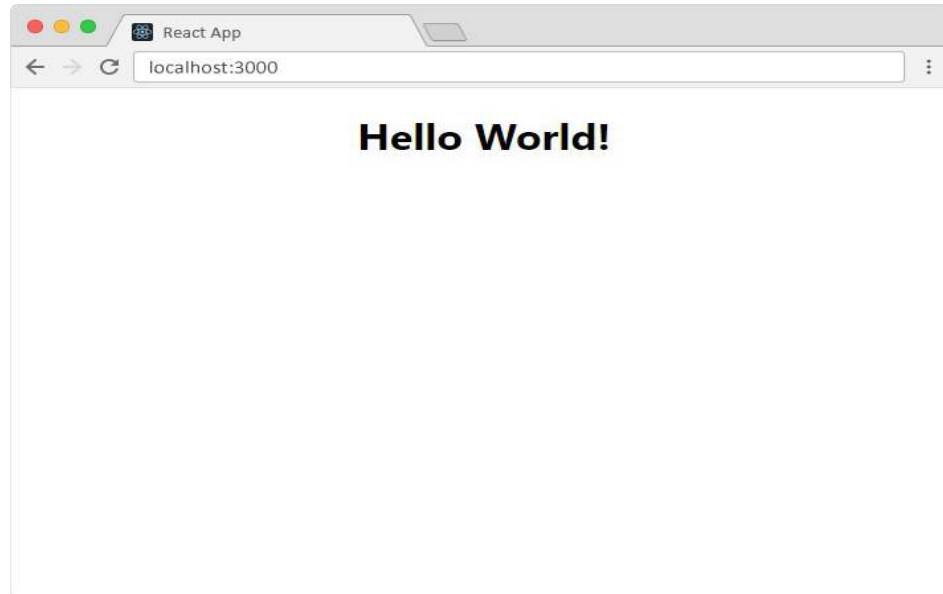
function App() {
  return (
    <div className="App">
      <h1>Hello World!</h1>
    </div>
  );
}

export default App;

```

Notice that we have removed the imports we do not need (logo.svg and App.css).

## The result:-



Now you have a React Environment on your computer, and you are ready to learn more about React.

In the rest of this tutorial we will use our "Show React" tool to explain the various aspects of React, and how they are displayed in the browser.

If you want to follow the same steps on your computer, start by stripping down the `src` folder to only contain one file: `index.js`. You should also remove any unnecessary lines of code inside the `index.js` file to make them look like the example in the "Show React" tool below:

## Example:-

Click the "Run Example" button to see the result.

`index.js`:

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

const myFirstElement = <h1>Hello React!</h1>

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myFirstElement);
```

([https://www.w3schools.com/REACT/showreact.asp?filename=demo2\\_react1](https://www.w3schools.com/REACT/showreact.asp?filename=demo2_react1))

## ReactDOM :-

The `react-dom` package provides DOM-specific methods that can be used at the top level of your app and as an escape hatch to get outside the React model if you need to.

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

If you use ES5 with npm, you can write:

```
var ReactDOM = require('react-dom');
```

The `react-dom` package also provides modules specific to client and server apps:

- ✓ `react-dom/client`
- ✓ `react-dom/server`

## Overview :-

The `react-dom` package exports these methods:

- ✓ `createPortal()`
- ✓ `flushSync()`

These `react-dom` methods are also exported, but are considered legacy:

- ✓ `render()`
- ✓ `hydrate()`
- ✓ `findDOMNode()`
- ✓ `unmountComponentAtNode()`

Note:

Both `render` and `hydrate` have been replaced with new client methods in React 18. These methods will warn that your app will behave as if it's running React 17 ([learn more here](#)).

## Browser Support:-

React supports all modern browsers, although some polyfills are required for older versions.

Note:-

We do not support older browsers that don't support ES5 methods or microtasks such as Internet Explorer. You may find that your apps do work in older browsers if polyfills such as `es5-shim` and `es5-sham` are included in the page, but you're on your own if you choose to take this path.

## Reference :-

## **createPortal()**

`createPortal(child, container)`

Creates a portal. Portals provide a way to render children into a DOM node that exists outside the hierarchy of the DOM component.

## **flushSync()**

`flushSync(callback)`

Force React to flush any updates inside the provided callback synchronously. This ensures that the DOM is updated immediately.

// Force this state update to be synchronous.

```
flushSync(() => {  
  setCount(count + 1);  
});
```

// By this point, DOM is updated.

### **Note:**

- ✓ `flushSync` can significantly hurt performance. Use sparingly.
- ✓ `flushSync` may force pending Suspense boundaries to show their `fallback` state.
- ✓ `flushSync` may also run pending effects and synchronously apply any updates they contain before returning.
- ✓ `flushSync` may also flush updates outside the callback when necessary to flush the updates inside the callback. For example, if there are pending updates from a click, React may flush those before flushing the updates inside the callback.

## **Legacy Reference :-**

### **render()**

`render(element, container[, callback])`

### **Note:-**

- ❖ Render has been replaced with `createRoot` in React 18. See `createRoot` for more info.
- ❖ Render a React element into the DOM in the supplied container and return a reference to the component (or returns null for stateless components).
- ❖ If the React element was previously rendered into container, this will perform an update on it and only mutate the DOM as necessary to reflect the latest React element.
- ❖ If the optional callback is provided, it will be executed after the component is rendered or updated.

### Note:-

`render()` controls the contents of the container node you pass in. Any existing DOM elements inside are replaced when first called. Later calls use React's DOM diffing algorithm for efficient updates.

`render()` does not modify the container node (only modifies the children of the container). It may be possible to insert a component to an existing DOM node without overwriting the existing children.

`render()` currently returns a reference to the root `ReactComponent` instance. However, using this return value is legacy and should be avoided because future versions of React may render components asynchronously in some cases. If you need a reference to the root `ReactComponent` instance, the preferred solution is to attach a callback ref to the root element.

Using `render()` to hydrate a server-rendered container is deprecated. Use `hydrateRoot()` instead.

### **hydrate()**

```
hydrate(element, container[, callback])
```

### Note:

`hydrate` has been replaced with `hydrateRoot` in React 18. See `hydrateRoot` for more info.

Same as `render()`, but is used to hydrate a container whose HTML contents were rendered by `ReactDOMServer`. React will attempt to attach event listeners to the existing markup.

React expects that the rendered content is identical between the server and the client. It can patch up differences in text content, but you should treat mismatches as bugs and fix them. In development mode, React warns about mismatches during hydration. There are no guarantees that attribute differences will be patched up in case of mismatches. This is important for performance reasons because in most apps, mismatches are rare, and so validating all markup would be prohibitively expensive.

If a single element's attribute or text content is unavoidably different between the server and the client (for example, a timestamp), you may silence the warning by adding `suppressHydrationWarning={true}` to the element. It only works one level deep, and is intended to be an escape hatch. Don't overuse it. Unless it's text content, React still won't attempt to patch it up, so it may remain inconsistent until future updates.

If you intentionally need to render something different on the server and the client, you can do a two-pass rendering. Components that render something different on the client can read a state variable like `this.state.isClient`, which you can set to `true` in `componentDidMount()`.



This way the initial render pass will render the same content as the server, avoiding mismatches, but an additional pass will happen synchronously right after hydration. Note that this approach will make your components slower because they have to render twice, so use it with caution.

Remember to be mindful of user experience on slow connections. The JavaScript code may load significantly later than the initial HTML render, so if you render something different in the client-only pass, the transition can be jarring. However, if executed well, it may be beneficial to render a “shell” of the application on the server, and only show some of the extra widgets on the client. To learn how to do this without getting the markup mismatch issues, refer to the explanation in the previous paragraph.

### **unmountComponentAtNode()**

### **unmountComponentAtNode(container)**

#### **Note:**

`unmountComponentAtNode` has been replaced with `root.unmount()` in React 18. See `createRoot` for more info.

Remove a mounted React component from the DOM and clean up its event handlers and state. If no component was mounted in the container, calling this function does nothing. Returns true if a component was unmounted and false if there was no component to unmount.

### **findDOMNode()**

#### **Note:**

`findDOMNode` is an escape hatch used to access the underlying DOM node. In most cases, use of this escape hatch is discouraged because it pierces the component abstraction. It has been deprecated in StrictMode.

### **findDOMNode(component)**

If this component has been mounted into the DOM, this returns the corresponding native browser DOM element. This method is useful for reading values out of the DOM, such as form field values and performing DOM measurements. **In most cases, you can attach a ref to the DOM node and avoid using `findDOMNode` at all.**

When a component renders to null or false, `findDOMNode` returns null. When a component renders to a string, `findDOMNode` returns a text DOM node containing that value. As of React 16, a component may return a fragment with multiple children, in which case `findDOMNode` will return the DOM node corresponding to the first non-empty child.

#### **Note:**

`findDOMNode` only works on mounted components (that is, components that have been placed in the DOM). If you try to call this on a component that has not been mounted yet (like calling `findDOMNode()` in `render()` on a component that has yet to be created) an exception will be thrown.

`findDOMNode` cannot be used on function components.

## **JSX - Components - Properties :-**

Instead of separating by technology React encourages to **combine Markup and UI Logic into encapsulated components**. JSX enables this by providing an easy way to write HTML Markup in JavaScript. If you are new to React or JSX you should read the Main concepts of React first.

## **JSX Components**

The portal engine is build with customization in mind and offers a very flexible way to customized the application. This chapter contains a small introduction to React / JSX components and describes how you can customize the appearance, markup and behaviour of JSX components.

## **Introduction to React / JSX components**

React embraces the fact that rendering logic is inherently coupled with other UI logic (like event handling and state updates). Instead of separating by technology React encourages to combine Markup and UI Logic into encapsulated components. JSX enables this by providing an easy way to write HTML Markup in JavaScript.

If you are new to React or JSX you should read the Main concepts of React first.

The portal engine only uses function components which means that components are just simple JavaScript functions that take a props object argument and return JSX. Each component file contains one main component which is exported as the default export.

## **Customizing JSX Components**

The process of customizing any component can be broken down into four steps:

1. Create the file `{PROJECT_ROOT}/public/portal-engine/{FRONTEND_BUILD_NAME}/{PATH_TO_FILE}` (for Pimcore 6 `{PROJECT_ROOT}/web/portal-engine/{FRONTEND_BUILD_NAME}/{PATH_TO_FILE}`)
2. Import everything you want to reused from the bundle source file with the portal-engine-bundle prefix
3. Add your customizations and export them
4. Re-export everything else from the bundle source files

## Override generic JSX components

The easiest way to start customizing the application is by overwriting a generic component completely. To overwrite a component we have to create the corresponding file `{PROJECT_ROOT}/public/portal-engine/{FRONTEND_BUILD_NAME}/{PATH_TO_FILE}` (for Pimcore 6 `{PROJECT_ROOT}/web/portal-engine/{FRONTEND_BUILD_NAME}/{PATH_TO_FILE}`). Make sure that `{PATH_TO_FILE}` matches the name and path of the bundle source file (relative to `assets` source folder). The next step is to create our customized component function and declare it as default export. The function is going to be called with the same props object argument as the original component function.

The following example shows how this could look like for the generic Teaser component `/scripts/components/Teaser.js`:

```
import React from 'react';
import Trans from "~/portal-engine/scripts/components/Trans";

// Create customized component function & declare it as default export
export default function ({
  image = {},
  title = '',
  href = '#',
  children
}) {
  return (
    <div className="card">
      <img className="card-img-top" {...image} alt={image.alt ||
title}/>
      <div className="card-body">
        <h5 className="card-title">{title}</h5>
        <div className="card-text">
          {children}
        </div>
        <a href={href} className="btn btn-primary">
          <Trans t={'teaser.cta'}/>
        </a>
      </div>
    </div>
  )
}

// Import all other functions from the bundle source file and re-export all
other exports export * from "portal-engine-bundle/scripts/components/Teaser";
```

## Customization through subcomponents:-

Some components allow you to customize parts of component through subcomponents. Subcomponents can be passed in through component props and usually end with the suffix `Component`.

If we look at the bundle source file of the Teaser component we find three of those subcomponents:

- ❖ ImageComponent
- ❖ BodyComponent
- ❖ DetailActionComponent

The following example shows how you can customize the Image subcomponent and keep the rest of the component unchanged:

```
import React from 'react';

// Import base component
import Teaser from "portal-engine-bundle/scripts/components/Teaser";

// Create custom subcomponent & export it
export function Image({
  image = {},
  title,
}) {
  return (
    <div className="embed-responsive embed-responsive-16by9">
      <img {...image} alt={image.alt || title} className="embed-responsive-item"/>
    </div>
  )
}

// Reuse the base Teaser component and pass the customized Image subcomponent
as ImageComponent prop
export default function (props) {
  return <Teaser {...props} ImageComponent={Image} />;
}

// Import all other functions from the bundle source file and re-export all
other exports
export * from "portal-engine-bundle/scripts/components/Teaser";
```

You can also use subcomponents to customize the overall structure of a component and reuse some parts of the original component. The example below shows how you could reuse the Image subcomponent while changing the rest of the Teaser completely:

```
import React from 'react';
import Trans from "~portal-engine/scripts/components/Trans";

// Import Image subcomponent from the bundle source file
import {Image} from "portal-engine-bundle/scripts/components/Teaser";

// Create custom Teaser component
export default function (props) {
  const {
    title = '',
    href = '#',
  }
```

```

    children
  } = props;

  return (
    <div className="card">
      <div className="card-img-top">
        /* Reuse the Image subcomponent from the bundle source file */
        <Image {...props}/>
      </div>

      <div className="card-body">
        <h5 className="card-title">{title}</h5>
        <div className="card-text">
          {children}
        </div>

        <a href={href} className="btn btn-primary">
          <Trans t="teaser.open"/>
        </a>
      </div>
    </div>

  )
}

// Import all other functions from the bundle source file and re-export all other exports
export * from "portal-engine-bundle/scripts/components/Teaser";

```

### Override connected components:-

A lot of non-generic components from the feature subfolder are connected to the global application store and have to be treated a bit differently if you want to customize them. While you can just overwrite the default export of a generic component you have to reconnect connected components to the store. A connected component file usually exports a `mapStateToProps` function, a `mapDispatchToProps` function/object and the unconnected JSX component as named exports. The default export is always the connected JSX component. You can connect your customized component by calling the `connect` function with `mapStateToProps` and `mapDispatchToProps` from the bundle source file.

The following example shows how this would look like for the connected `DataPoolPagination` component `/scripts/features/data-pool-list/components/DataPoolPagination.js`.

```

import React from 'react';
import {connect} from "react-redux";

/* Import the mapDispatchToProps, mapStateToProps from the bundle source file */
import {
  mapDispatchToProps,
  mapStateToProps

```

```

} from "portal-engine-bundle/scripts/features/data-pool-
list/components/DataPoolPagination";
import {noop} from "~portal-engine/scripts/utils/utils";

// Create custom pagination component
function Pagination({
  pageCount = 5,
  currentPage = 1,
  onPageClick = noop
}) {
  const pages = new Array(pageCount).fill(0).map((_, index) => index + 1);

  return (
    <nav>
      <ul className="pagination">
        {pages.map((current) => (
          <li key={current} className={`page-item ${current ===
currentPage ? 'active' : ''}`}>
            <button type="button" className="page-link"
onClick={() => onPageClick(current)}>
              {current}
            </button>
          </li>
        ))}
      </ul>
    </nav>
  )
}

// Connect customized component with the application store and declare it as
default export
export default connect(mapStateToProps, mapDispatchToProps)(Pagination)

```

### Adding new features:-

If you want to take it a step further you can customize `mapStateToProps` and `mapDispatchToProps` too. This allows you to add new features or change the behaviour of a component completely. The following example illustrates this:

```

import React from 'react';
import {connect} from "react-redux";

/* Import mapDispatchToProps & mapStateToProps from the bundle source file */
import {
  mapDispatchToProps,
  mapStateToProps
} from "portal-engine-bundle/scripts/features/data-pool-
list/components/DataPoolPagination";
import {noop} from "~portal-engine/scripts/utils/utils";

// Customize mapStateToProps
function customizedMapStateToProps(state) {
  return {
    // Reuse mapStateToProps from the bundle source file
    ...mapStateToProps(state),

```

```

        // Add new custom props
        additionalProp: getDataFromTheState(state),
        otherAdditionalProp: getOtherDataFromTheState(state),

        // Overwrite existing props
        pageCount: customizedPageCountSelector(state)
    }
}

// Customize mapDispatchToProps
const customizedMapDispatchToProps = {
    // Reuse mapDispatchToProps from the bundle source file
    ...mapDispatchToProps,

    // Add custom event handler function
    onSomeCustomEvent: someCustomActionCreator,

    // Overwrite existing behaviour
    onPageClick: goToRandomPage
};

// Customize component
function Pagination({
    pageCount = 5,
    currentPage = 1,
    additionalProp,
    someOtherAdditionalProp,
    onPageClick = noop,
    onSomeCustomEvent = noop
}) {
    const pages = new Array(pageCount).fill(0).map((_, index) => index + 1);

    return (
        <nav>
            {/* Use custom props*/}
            {additionalProp}

            {/* Use custom event handler function */}
            <button onClick={() => onSomeCustomEvent()}>Surprise!</button>

            {/* Additional customizations */}
        </nav>
    )
}

// Connect the component with your customized mapStateToProps &
mapDispatchToProps
export default connect(customizedMapStateToProps,
    customizedMapDispatchToProps)(Pagination)

```

## JSX Properties:-

Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen.

## Function and Class Components:-

The simplest way to define a component is to write a JavaScript function:

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}
```

This function is a valid React component because it accepts a single “props” (which stands for properties) object argument with data and returns a React element. We call such components “function components” because they are literally JavaScript functions.

You can also use an ES6 class to define a component:

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Hello, {this.props.name}</h1>;  
  }  
}
```

The above two components are equivalent from React’s point of view.

Function and Class components both have some additional features that we will discuss in the next sections.

## Rendering a Component:-

Previously, we only encountered React elements that represent DOM tags:

```
const element = <div />;
```

However, elements can also represent user-defined components:

```
const element = <Welcome name="Sara" />;
```

When React sees an element representing a user-defined component, it passes JSX attributes and children to this component as a single object. We call this object “props”.

For example, this code renders “Hello, Sara” on the page:

```
function Welcome(props) { return <h1>Hello, {props.name}</h1>;  
}  
  
const root = ReactDOM.createRoot(document.getElementById('root'));  
const element = <Welcome name="Sara" />; root.render(element);
```

## Try it on CodePen

Let’s recap what happens in this example:



1. We call `root.render()` with the `<Welcome name="Sara" />` element.
2. React calls the `Welcome` component with `{name: 'Sara'}` as the props.
3. Our `Welcome` component returns a `<h1>Hello, Sara</h1>` element as the result.
4. React DOM efficiently updates the DOM to match `<h1>Hello, Sara</h1>`.

**Note:** Always start component names with a capital letter.

React treats components starting with lowercase letters as DOM tags. For example, `<div />` represents an HTML `div` tag, but `<Welcome />` represents a component and requires `Welcome` to be in scope.

To learn more about the reasoning behind this convention, please read [JSX In Depth](#).

## Composing Components:-

Components can refer to other components in their output. This lets us use the same component abstraction for any level of detail. A button, a form, a dialog, a screen: in React apps, all those are commonly expressed as components.

For example, we can create an `App` component that renders `Welcome` many times:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />      <Welcome name="Cahal" />      <Welcome
name="Edite" />      </div>
    );
}
```

## Try it on CodePen

Typically, new React apps have a single `App` component at the very top. However, if you integrate React into an existing app, you might start bottom-up with a small component like `Button` and gradually work your way to the top of the view hierarchy.

## Extracting Components:-

Don't be afraid to split components into smaller components.

For example, consider this `Comment` component:

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
```

```

    <img className="Avatar"
      src={props.author.avatarUrl}
      alt={props.author.name}
    />
    <div className="UserInfo-name">
      {props.author.name}
    </div>
  </div>
  <div className="Comment-text">
    {props.text}
  </div>
  <div className="Comment-date">
    {formatDate(props.date)}
  </div>
</div>
);
}

```

## Try it on CodePen

It accepts `author` (an object), `text` (a string), and `date` (a date) as props, and describes a comment on a social media website.

This component can be tricky to change because of all the nesting, and it is also hard to reuse individual parts of it. Let's extract a few components from it.

First, we will extract `Avatar`:

```

function Avatar(props) {
  return (
    <img className="Avatar" src={props.user.avatarUrl}
    alt={props.user.name} />
  );
}

```

The `Avatar` doesn't need to know that it is being rendered inside a `Comment`. This is why we have given its prop a more generic name: `user` rather than `author`.

We recommend naming props from the component's own point of view rather than the context in which it is being used.

We can now simplify `Comment` a tiny bit:

```

function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <Avatar user={props.author} />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
    </div>
  );
}

```

```

    </div>
    <div className="Comment-date">
      {formatDate(props.date)}
    </div>
  </div>
);
}

```

Next, we will extract a `UserInfo` component that renders an `Avatar` next to the user's name:

```

function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user} />
      <div
        className="UserInfo-name">
          {props.user.name}
        </div>
      </div>
    );
}

```

This lets us simplify `Comment` even further:

```

function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}

```

## Try it on CodePen

Extracting components might seem like grunt work at first, but having a palette of reusable components pays off in larger apps. A good rule of thumb is that if a part of your UI is used several times (`Button`, `Panel`, `Avatar`), or is complex enough on its own (`App`, `FeedStory`, `Comment`), it is a good candidate to be extracted to a separate component.

## Props are Read-Only

Whether you declare a component as a function or a class, it must never modify its own props. Consider this `sum` function:

```

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

```

Such functions are called “pure” because they do not attempt to change their inputs, and always return the same result for the same inputs.

In contrast, this function is impure because it changes its own input:

```
function withdraw(account, amount) {  
  account.total -= amount;  
}
```

React is pretty flexible but it has a single strict rule:

**All React components must act like pure functions with respect to their props.**

Of course, application UIs are dynamic and change over time. In the next section, we will introduce a new concept of “state”. State allows React components to change their output over time in response to user actions, network responses, and anything else, without violating this rule.

## Fetch API – ReactJS

**ReactJS:** ReactJS is a declarative, efficient, and flexible JavaScript library for building user interfaces. It’s ‘V’ in MVC. ReactJS is an open-source, component-based front-end library responsible only for the view layer of the application. It is maintained by Facebook.

**API:** API is an abbreviation for Application Programming Interface which is a collection of communication protocols and subroutines used by various programs to communicate between them. A programmer can make use of various API tools to make its program easier and simpler. Also, an API facilitates the programmers with an efficient way to develop their software programs.

**Approach:** In this article, we will know how we fetch the data from API (Application Programming Interface). For the data, we have used the API endpoint from <http://jsonplaceholder.typicode.com/users> we have created the component in App.js and styling the component in App.css. From the API we have target “id”, “name”, “username”, “email” and fetch the data from API endpoints. Below is the stepwise implementation of how we fetch the data from an API in react. We will use the fetch function to get the data from the API.

Step by step implementation to fetch data from an api in react.

➤ **Step 1:** Create React Project

```
npm create-react-app MY-APP
```

➤ **Step 2:** Change your directory and enter your main folder charting as

```
cd MY-APP
```

➤ **Step 3:** API endpoint

```
https://jsonplaceholder.typicode.com/users
```

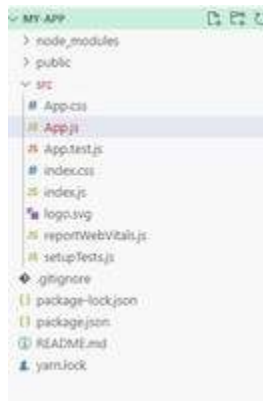
```
// 20210815205448
// https://jsonplaceholder.typicode.com/users

{
  "id": 1,
  "name": "Leanne Graham",
  "username": "Bret",
  "email": "Sincere@april.biz",
  "address": {
    "street": "Kulas Light",
    "suite": "Apt. 556",
    "city": "Wenborough",
    "zipcode": "92998-3874",
    "geo": {
      "lat": "-37.1559",
      "lng": "81.1496"
    }
  },
  "phone": "1-770-736-8011 x56442",
  "website": "hildegard.org",
  "company": {
    "name": "Romaguera-Crona",
    "catchPhrase": "Multi-layered client-server neural-net",
    "bs": "harness real-time e-markets"
  }
}
```

## API

- **Step 4:** Write code in App.js to fetch data from API and we are using fetch function.

**Project Structure:** It will look the following.



## Project Structure

### Example:

```
import React from "react";
import './App.css';
class App extends React.Component {

  // Constructor
  constructor(props) {
    super(props);
```

```

    this.state = {
      items: [],
      DataisLoaded: false
    };
  }

  // componentDidMount is used to
  // execute the code
  componentDidMount() {
    fetch(
      "https://jsonplaceholder.typicode.com/users")
      .then((res) => res.json())
      .then((json) => {
        this.setState({
          items: json,
          DataisLoaded: true
        });
      })
  }

  render() {
    const { DataisLoaded, items } = this.state;
    if (!DataisLoaded) return <div>
      <h1> Pleses wait some time </h1> </div> ;
    return (
      <div className = "App">
        <h1> Fetch data from an api in react </h1> {
          items.map((item) => (
            <ol key = { item.id } >
              User_Name: { item.username },
              Full_Name: { item.name },
              User_Email: { item.email }
            </ol>
          ))
        }
      </div>
    );
  }
}

```

export default App;

Write code in App.css for styling the app.js file.

```

.App {
  text-align: center;
  color: Green;
}

```

```

}
.App-header {
  background-color: #282c34;
  min-height: 100vh;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  font-size: calc(10px + 2vmin);
  color: white;
}
.App-link {
  color: #61dafb;
}

@keyframes App-logo-spin {
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(360deg);
  }
}

```

**Step to run the application:** Open the terminal and type the following command.

**npm start**

**Output:** Open the browser and our project is shown in the URL **http://localhost:3000/**

## Fetch data from an api in react

- User\_Name: Bret, Full\_Name: Leanne Graham, User\_Email: Sincere@april.biz
- User\_Name: Antonette, Full\_Name: Ervin Howell, User\_Email: Shanna@melissa.tv
- User\_Name: Samantha, Full\_Name: Clementine Bauch, User\_Email: Nathan@yesenia.net
- User\_Name: Karianne, Full\_Name: Patricia Lebsack, User\_Email: Julianne.OConner@kory.org
- User\_Name: Kamren, Full\_Name: Chelsey Dietrich, User\_Email: Lucio\_Hettinger@annie.ca
- User\_Name: Leopoldo, Full\_Name: Corkery, Full\_Name: Mrs. Dennis Schulist, User\_Email: Karley\_Dach@jasper.info
- User\_Name: Elwyn, Full\_Name: Skiles, Full\_Name: Kurtis Weissnat, User\_Email: Telly.Hoeger@billy.biz
- User\_Name: Maxime, Full\_Name: Nienow, Full\_Name: Nicholas Runolfsdottir V, User\_Email: Sherwood@rosamond.me
- User\_Name: Delphine, Full\_Name: Glenna Reichert, User\_Email: Chaim\_McDermott@dana.io
- User\_Name: Moriah, Full\_Name: Stanton, Full\_Name: Clementina DuBuque, User\_Email: Rey.Padberg@karina.biz

## React State and Lifecycle:-

Introduces the concept of state and lifecycle in a React component. You can find a detailed component API reference [here](#).

Consider the ticking clock example from one of the previous sections. In Rendering Elements, we have only learned one way to update the UI. We call `root.render()` to change the rendered output:

```
const root = ReactDOM.createRoot(document.getElementById('root'));
```

```
function tick() {  
  const element = (  
    <div>  
      <h1>Hello, world!</h1>  
      <h2>It is {new Date().toLocaleTimeString()}.</h2>  
    </div>  
  );  
  root.render(element);  
}
```

```
setInterval(tick, 1000);
```

### Try it on CodePen

In this section, we will learn how to make the Clock component truly reusable and encapsulated. It will set up its own timer and update itself every second.

We can start by encapsulating how the clock looks:

```
const root = ReactDOM.createRoot(document.getElementById('root'));
```

```
function Clock(props) {  
  return (  
    <div>      <h1>Hello, world!</h1>      <h2>It is {props.date.toLocaleTimeString()}.</h2>  
    </div> );  
}
```

```
function tick() {  
  root.render(<Clock date={new Date()} />);  
}
```

```
setInterval(tick, 1000);
```

### Try it on CodePen

However, it misses a crucial requirement: the fact that the Clock sets up a timer and updates the UI every second should be an implementation detail of the Clock.

Ideally we want to write this once and have the Clock update itself:

```
root.render(<Clock />);
```



To implement this, we need to add “state” to the Clock component.

State is similar to props, but it is private and fully controlled by the component.

## Converting a Function to a Class

You can convert a function component like Clock to a class in five steps:

1. Create an ES6 class, with the same name, that extends `React.Component`.
2. Add a single empty method to it called `render()`.
3. Move the body of the function into the `render()` method.
4. Replace props with `this.props` in the `render()` body.
5. Delete the remaining empty function declaration.

```
class Clock extends React.Component {  
  render() {  
    return (  
      <div>  
        <h1>Hello, world!</h1>  
        <h2>It is {this.props.date.toLocaleTimeString()}.</h2>  
      </div>  
    );  
  }  
}
```

## Try it on CodePen

Clock is now defined as a class rather than a function.

The render method will be called each time an update happens, but as long as we render `<Clock />` into the same DOM node, only a single instance of the Clock class will be used. This lets us use additional features such as local state and lifecycle methods.

## Adding Local State to a Class

We will move the date from props to state in three steps:

1. Replace `this.props.date` with `this.state.date` in the `render()` method:

```
class Clock extends React.Component {  
  render() {  
    return (  
      <div>  
        <h1>Hello, world!</h1>  
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>    </div>  
    );  
  }  
}
```

```
}
}
```

2. Add a class constructor that assigns the initial `this.state`:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()}; }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Note how we pass props to the base constructor:

```
constructor(props) {
  super(props);  this.state = {date: new Date()};
}
```

Class components should always call the base constructor with props.

3. Remove the date prop from the `<Clock />` element:

```
root.render(<Clock />);
```

We will later add the timer code back to the component itself.

The result looks like this:

```
class Clock extends React.Component {
  constructor(props) {  super(props);  this.state = {date: new Date()}; }
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>    </div>
    );
  }
}
```

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

## Try it on CodePen

Next, we'll make the Clock set up its own timer and update itself every second.

## Adding Lifecycle Methods to a Class

In applications with many components, it's very important to free up resources taken by the components when they are destroyed.

We want to set up a timer whenever the Clock is rendered to the DOM for the first time. This is called "mounting" in React.

We also want to clear that timer whenever the DOM produced by the Clock is removed. This is called "unmounting" in React.

We can declare special methods on the component class to run some code when a component mounts and unmounts:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() { }
  componentWillUnmount() { }
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

These methods are called "lifecycle methods".

The `componentDidMount()` method runs after the component output has been rendered to the DOM. This is a good place to set up a timer:

```
componentDidMount() {
```

```
this.timerID = setInterval(    () => this.tick(),    1000    ); }
```

Note how we save the timer ID right on this (this.timerID).

While this.props is set up by React itself and this.state has a special meaning, you are free to add additional fields to the class manually if you need to store something that doesn't participate in the data flow (like a timer ID).

We will tear down the timer in the componentWillUnmount() lifecycle method:

```
componentWillUnmount() {  
  clearInterval(this.timerID); }
```

Finally, we will implement a method called tick() that the Clock component will run every second.

It will use this.setState() to schedule updates to the component local state:

```
class Clock extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {date: new Date()};  
  }  
  
  componentDidMount() {  
    this.timerID = setInterval(  
      () => this.tick(),  
      1000  
    );  
  }  
  
  componentWillUnmount() {  
    clearInterval(this.timerID);  
  }  
  
  tick() { this.setState({    date: new Date()    }); }  
  render() {  
    return (  
      <div>  
        <h1>Hello, world!</h1>  
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>  
      </div>  
    );  
  }  
}
```

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

## Try it on CodePen

Now the clock ticks every second.

Let's quickly recap what's going on and the order in which the methods are called:

1. When `<Clock />` is passed to `root.render()`, React calls the constructor of the Clock component. Since Clock needs to display the current time, it initializes `this.state` with an object including the current time. We will later update this state.
2. React then calls the Clock component's `render()` method. This is how React learns what should be displayed on the screen. React then updates the DOM to match the Clock's render output.
3. When the Clock output is inserted in the DOM, React calls the `componentDidMount()` lifecycle method. Inside it, the Clock component asks the browser to set up a timer to call the component's `tick()` method once a second.
4. Every second the browser calls the `tick()` method. Inside it, the Clock component schedules a UI update by calling `setState()` with an object containing the current time. Thanks to the `setState()` call, React knows the state has changed, and calls the `render()` method again to learn what should be on the screen. This time, `this.state.date` in the `render()` method will be different, and so the render output will include the updated time. React updates the DOM accordingly.
5. If the Clock component is ever removed from the DOM, React calls the `componentWillUnmount()` lifecycle method so the timer is stopped.

## Using State Correctly

There are three things you should know about `setState()`.

### Do Not Modify State Directly

For example, this will not re-render a component:

```
// Wrong
this.state.comment = 'Hello';
```

Instead, use `setState()`:

```
// Correct
this.setState({comment: 'Hello'});
```

The only place where you can assign `this.state` is the constructor.

## State Updates May Be Asynchronous

React may batch multiple `setState()` calls into a single update for performance.

Because `this.props` and `this.state` may be updated asynchronously, you should not rely on their values for calculating the next state.

For example, this code may fail to update the counter:

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

To fix it, use a second form of `setState()` that accepts a function rather than an object. That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument:

```
// Correct
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

We used an arrow function above, but it also works with regular functions:

```
// Correct
this.setState(function(state, props) {
  return {
    counter: state.counter + props.increment
  };
});
```

## State Updates are Merged

When you call `setState()`, React merges the object you provide into the current state.

For example, your state may contain several independent variables:

```
constructor(props) {
  super(props);
  this.state = {
    posts: [],    comments: []  };
}
```

Then you can update them independently with separate `setState()` calls:

```
componentDidMount() {
  fetchPosts().then(response => {
```

```

    this.setState({
      posts: response.posts    });
  });

  fetchComments().then(response => {
    this.setState({
      comments: response.comments    });
  });
}

```

The merging is shallow, so `this.setState({comments})` leaves `this.state.posts` intact, but completely replaces `this.state.comments`.

### The Data Flows Down:-

Neither parent nor child components can know if a certain component is stateful or stateless, and they shouldn't care whether it is defined as a function or a class.

This is why state is often called local or encapsulated. It is not accessible to any component other than the one that owns and sets it.

A component may choose to pass its state down as props to its child components:

```
<FormattedDate date={this.state.date} />
```

The `FormattedDate` component would receive the date in its props and wouldn't know whether it came from the `Clock`'s state, from the `Clock`'s props, or was typed by hand:

```

function FormattedDate(props) {
  return <h2>It is {props.date.toLocaleTimeString()}.</h2>;
}

```

### Try it on CodePen

This is commonly called a “top-down” or “unidirectional” data flow. Any state is always owned by some specific component, and any data or UI derived from that state can only affect components “below” them in the tree.

If you imagine a component tree as a waterfall of props, each component's state is like an additional water source that joins it at an arbitrary point but also flows down.

To show that all components are truly isolated, we can create an `App` component that renders three `<Clock>`s:

```

function App() {
  return (

```

```

    <div>
      <Clock />    <Clock />    <Clock />    </div>
    );
  }

```

### Try it on CodePen

Each Clock sets up its own timer and updates independently.

In React apps, whether a component is stateful or stateless is considered an implementation detail of the component that may change over time. You can use stateless components inside stateful components, and vice versa.

### JS Local storage - Events - Lifting State Up:-

Local storage is a web storage object that allows JavaScript sites and apps to keep key-value pairs in a web browser with no expiration date.

This means the data survives page refreshes (session Storage) and even browser restarts. This indicates that the data stored in the browser will remain even when the browser window is closed.

In basic terms, local storage enables developers to store and retrieve data in the browser. It is critical to understand, though, that using local Storage as a database for your project is not a good practice, since data will be lost when the user clears the cache, among other things. Developers frequently use local Storage for adding a dark mode feature to an application, saving a to-do item, or persisting a user's form input values, among many other scenarios. In this post, we'll take a look at how to use localStorage with React hooks to set and get items easily.

### What are React Hooks?

React Hooks are JavaScript functions that you may import from the React package to add capabilities to your components. Hooks allow React developers to use state and lifecycle methods within functional components. They also operate with existing code, making them easily adoptable into a codebase.

We will need two hooks in order to use localStorage with react hooks:

- **useState()** – The state of your application is guaranteed to change at some time. The useState() hook is a function that accepts one parameter, the initial state (which might be the value of a variable, an object, or any other sort of data in your component), and returns two values: the current state and a function that can be used to update the state.
- **useEffect()** – The Effect Hook is activated by default after the first render and each time the state is changed. As the names suggest, it is used to perform an effect each time the state changes. This hook is great for configuring listeners, retrieving data from the API, and deleting listeners before the component is removed from the DOM.



## How to Implement local Storage in React

Local Storage provides us with access to a browser's storage object, which includes five methods:

- **setItem()**: This method is used to add a key and a value to localStorage.
- **getItem()**: This method is used to get an item from localStorage using the key.
- **removeItem()**: This technique is used to delete an item from localStorage based on its key.
- **clear()**: This technique is used to delete all instances of localStorage.
- **key()**: When you supply a number, it aids in the retrieval of a localStorage key.

In this post, we will only consider the most popular methods, which are the first two methods.

### How to Use the setItem() Method

By giving values to a key, this technique is used to store objects in localStorage. This value can be of any datatype, including text, integer, object, array, and so on.

It is vital to remember that in order to store data in localStorage, you must first stringify it with the `JSON.stringify()` function.

```
const [items, setItems] = useState([]);

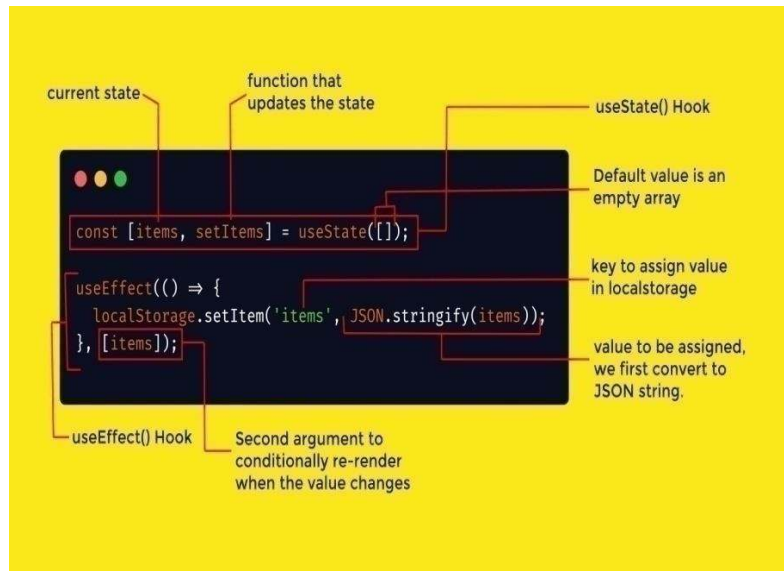
useEffect(() => {
  localStorage.setItem('items', JSON.stringify(items));
}, [items]);
```

In the above code, we first created a state and assigned it an empty array (yours could be any other datatype). Second, we used `useEffect()` to add objects to localStorage whenever the value of our state changed. We did this by passing the state as the second argument.

Basically, this is the major code responsible for adding key-value pairs to localStorage:

```
localStorage.setItem('items', JSON.stringify(items));
```

Simply put, the preceding code names the key (lists) and then assigns a value to it, but we had to first ensure that the data we were adding was a JSON string. We use `JSON.stringify()` to convert a JSON object to JSON text stored in a string, which can then be transmitted to the web server.



**Structure of how Hooks works with local storage to set items**

### How to Use the `getItem()` Method:-

This method retrieves objects from `localStorage`. There are other methods to accomplish this with React, but we will use the `useEffect()` hook because it is the best one.

The `useEffect()` hook helps us fetch all items on first render, which means that when the component mounts or re-renders, it obtains all of our data from `localStorage`.

Note that this is why we passed in an empty second argument.

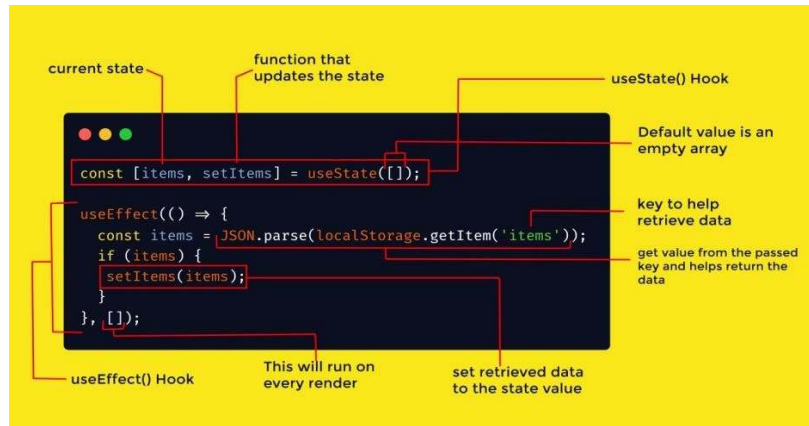
```

const [items, setItems] = useState([]);

useEffect(() => {
  const items = JSON.parse(localStorage.getItem('items'));
  if (items) {
    setItems(items);
  }
}, []);

```

It is important to remember that when we stored the data, we first converted it to a JSON string. This means that in order for us to now make use of it, we need to convert JSON string back to a JSON object. We do this with the `JSON.parse()` method.



Structure of how hooks works with local storage to get items

## JS - Events:-

Handling events with React elements is very similar to handling events on DOM elements. There are some syntax differences:

- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.

For example, the HTML:

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

is slightly different in React:

```
<button onClick={activateLasers}> Activate Lasers
</button>
```

Another difference is that you cannot return false to prevent default behavior in React. You must call `preventDefault` explicitly. For example, with plain HTML, to prevent the default form behavior of submitting, you can write:

```
<form onsubmit="console.log('You clicked submit.');" return false">
  <button type="submit">Submit</button>
</form>
```

In React, this could instead be:

```
function Form() {
  function handleSubmit(e) {
    e.preventDefault(); console.log('You clicked submit.');"
  }
}
```

```

return (
  <form onSubmit={handleSubmit}>
    <button type="submit">Submit</button>
  </form>
);
}

```

Here, `e` is a synthetic event. React defines these synthetic events according to the W3C spec, so you don't need to worry about cross-browser compatibility. React events do not work exactly the same as native events. See the [SyntheticEvent](#) reference guide to learn more.

When using React, you generally don't need to call `addEventListener` to add listeners to a DOM element after it is created. Instead, just provide a listener when the element is initially rendered.

When you define a component using an ES6 class, a common pattern is for an event handler to be a method on the class. For example, this `Toggle` component renders a button that lets the user toggle between "ON" and "OFF" states:

```

class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(prevState => ({
      isToggleOn: !prevState.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}> {this.state.isToggleOn ? 'ON' : 'OFF'}
    </button>
    );
  }
}

```

### Try it on CodePen

You have to be careful about the meaning of `this` in JSX callbacks. In JavaScript, class methods are not bound by default. If you forget to bind `this.handleClick` and pass it to `onClick`, `this` will be undefined when the function is actually called.

This is not React-specific behavior; it is a part of how functions work in JavaScript. Generally, if you refer to a method without `()` after it, such as `onClick={this.handleClick}`, you should bind that method.

If calling bind annoys you, there are two ways you can get around this. You can use public class fields syntax to correctly bind callbacks:

```
class LoggingButton extends React.Component {
  // This syntax ensures `this` is bound within handleClick. handleClick = () => {
  console.log('this is:', this); }; render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

This syntax is enabled by default in Create React App.

If you aren't using class fields syntax, you can use an arrow function in the callback:

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }

  render() {
    // This syntax ensures `this` is bound within handleClick return (
    <button onClick={() =>
    this.handleClick()}> Click me
    </button>
    );
  }
}
```

The problem with this syntax is that a different callback is created each time the LoggingButton renders. In most cases, this is fine. However, if this callback is passed as a prop to lower components, those components might do an extra re-rendering. We generally recommend binding in the constructor or using the class fields syntax, to avoid this sort of performance problem.

### Passing Arguments to Event Handlers:-

Inside a loop, it is common to want to pass an extra parameter to an event handler. For example, if id is the row ID, either of the following would work:

```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

The above two lines are equivalent, and use arrow functions and `Function.prototype.bind` respectively.

In both cases, the `e` argument representing the React event will be passed as a second argument after the ID. With an arrow function, we have to pass it explicitly, but with `bind` any further arguments are automatically forwarded.

### Lifting State up in ReactJS:-

**Lifting up the State:** As we know, every component in React has its own state. Because of this sometimes data can be redundant and inconsistent. So, by Lifting up the state we make the state of the parent component as a single source of truth and pass the data of the parent in its children.

**Time to use Lift up the State:** If the data in “parent and children components” or in “cousin components” is Not in Sync.

**Example 1:** If we have 2 components in our App. **A** -> **B** where, A is parent of B. keeping the same data in both Component A and B might cause inconsistency of data.

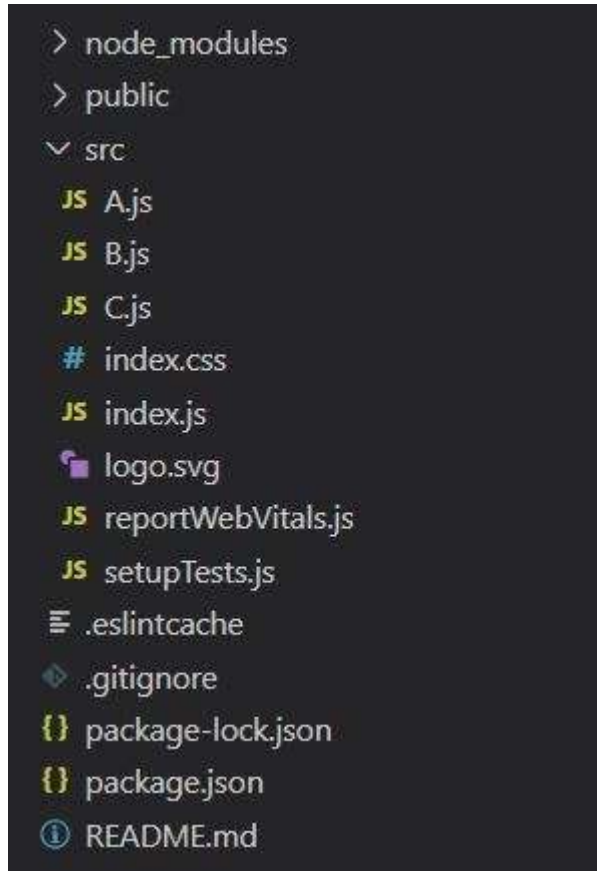
**Example 2:** If we have 3 components in our App.



Where A is the parent of B and C. In this case, If there is some Data only in component B but, component C also wants that data. We know Component C cannot access the data because a component can talk only to its parent or child (Not cousins).

**Problem:** Let's Implement this with a simple but general example. We are considering the second example.

## Complete File Structure:-



**Approach:** To solve this, we will Lift the state of component B and component C to component A. Make A.js as our Main Parent by changing the path of App in the index.js file

**Before:** `import App from './App';`

**After:** `import App from './A';`

### Filename- A.js:

```
import React, { Component } from 'react';
import B from './B'
import C from './C'

class A extends Component {

  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {text: ""};
  }
```

```

handleTextChange(newText) {
  this.setState({text: newText});
}

render() {
  return (
    <React.Fragment>
      <B text={this.state.text}
        handleTextChange={this.handleTextChange}/>
      <C text={this.state.text} />
    </React.Fragment>
  );
}
}

export default A;

```

### Filename- B.js:

```

import React,{ Component } from 'react';

class B extends Component {

  constructor(props) {
    super(props);
    this.handleTextChange = this.handleTextChange.bind(this);
  }

  handleTextChange(e){
    this.props.handleTextChange(e.target.value);
  }

  render() {
    return (
      <input value={this.props.text}
        onChange={this.handleTextChange} />
    );
  }
}

export default B;

```

### Filename- C.js:

```

import React,{ Component } from 'react';

class C extends Component {

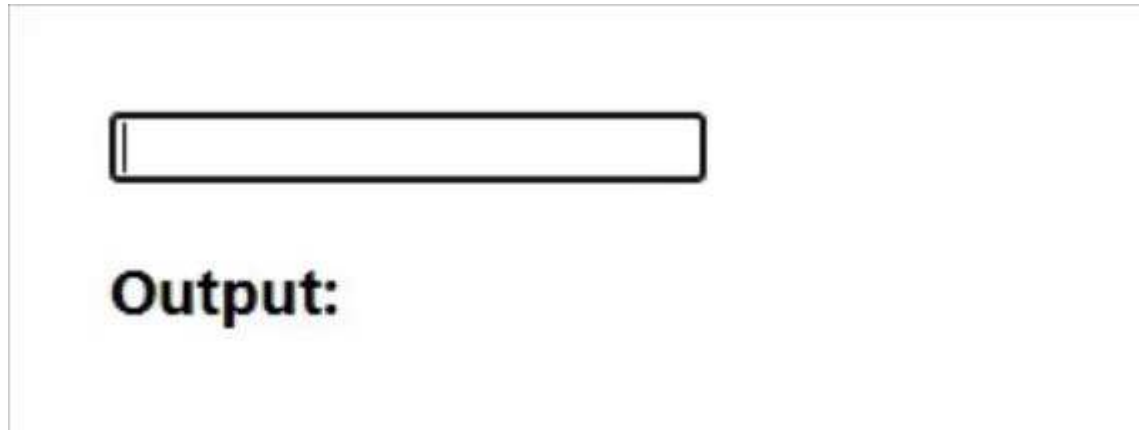
  render() {
    return (
      <h3>Output: {this.props.text}</h3>
    );
  }
}

```



```
}
export default C;
```

**Output:** Now, component C can Access text in component B through component A.



### Composition and Inheritance:-

React has a powerful composition model, and we recommend using composition instead of inheritance to reuse code between components.

Some components don't know their children ahead of time. This is especially common for components like `Sidebar` or `Dialog` that represent generic "boxes".

We recommend that such components use the special `children` prop to pass children elements directly into their output:

```
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children} </div>
    );
}
```

This lets other components pass arbitrary children to them by nesting the JSX:

```
function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">Welcome </h1> <p className="Dialog-
message"> Thank you for visiting our spacecraft! </p> </FancyBorder>
    );
}
```

## Try it on CodePen

Anything inside the `<FancyBorder>` JSX tag gets passed into the `FancyBorder` component as a `children` prop. Since `FancyBorder` renders `{props.children}` inside a `<div>`, the passed elements appear in the final output.

While this is less common, sometimes you might need multiple “holes” in a component. In such cases you may come up with your own convention instead of using `children`:

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}    </div>
      <div className="SplitPane-right">
        {props.right}   </div>
    </div>
  );
}
```

```
function App() {
  return (
    <SplitPane
      left={
        <Contacts />    }
      right={
        <Chat />      } />
  );
}
```

## Try it on CodePen

React elements like `<Contacts />` and `<Chat />` are just objects, so you can pass them as props like any other data. This approach may remind you of “slots” in other libraries but there are no limitations on what you can pass as props in React.

## Specialization

Sometimes we think about components as being “special cases” of other components. For example, we might say that a `WelcomeDialog` is a special case of `Dialog`.

In React, this is also achieved by composition, where a more “specific” component renders a more “generic” one and configures it with props:

```
function Dialog(props) {
  return (
```

```

    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}    </h1>
      <p className="Dialog-message">
        {props.message}    </p>
    </FancyBorder>
  );
}

function WelcomeDialog() {
  return (
    <Dialog    title="Welcome"    message="Thank you for visiting our spacecraft!" /> );
}

```

### Try it on CodePen

Composition works equally well for components defined as classes:

```

function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
      {props.children}    </FancyBorder>
    );
}

class SignUpDialog extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.handleSignUp = this.handleSignUp.bind(this);
    this.state = {login: ""};
  }

  render() {
    return (
      <Dialog title="Mars Exploration Program"
        message="How should we refer to you?">
        <input value={this.state.login}    onChange={this.handleChange} />    <button
onClick={this.handleSignUp}>    Sign Me Up!    </button>    </Dialog>
    );
  }
}

```

```
}

handleChange(e) {
  this.setState({login: e.target.value});
}

handleSignUp() {
  alert(`Welcome aboard, ${this.state.login}!`);
}
}
```

**Try it on CodePen**

### **So What About Inheritance?**

At Facebook, we use React in thousands of components, and we haven't found any use cases where we would recommend creating component inheritance hierarchies.

Props and composition give you all the flexibility you need to customize a component's look and behavior in an explicit and safe way. Remember that components may accept arbitrary props, including primitive values, React elements, or functions.

If you want to reuse non-UI functionality between components, we suggest extracting it into a separate JavaScript module. The components may import it and use that function, object, or a class, without extending it.

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

### Cloud providers Overview :-

Here is a list of my top 10 cloud service providers:

1. Amazon Web Services (AWS)
2. Microsoft Azure
3. Google Cloud
4. Alibaba Cloud
5. IBM Cloud
6. Oracle
7. Salesforce
8. SAP
9. Rackspace Cloud
10. VMWare

The following table summarizes the top 3 key players and their offerings in the cloud computing world:

	<b>AWS</b>	<b>Azure</b>	<b>Google Cloud</b>
Company	AWS Inc.	Microsoft	Google
Launch year	2006	2010	2008
Geographical Regions	25	54	21
Availability Zones	78	140 (countries)	61
Key offerings	Compute, storage, database, analytics, networking, machine learning, and AI, mobile, developer tools, IoT, security, enterprise applications, blockchain.	Compute, storage, mobile, data management, messaging, media services, CDN, machine learning and AI, developer tools, security, blockchain, functions, IoT.	Compute, storage, databases, networking, big data, cloud AI, management tools, Identity and security, IoT, API platform
Compliance Certificates	46	90	
Annual Revenue	\$33 billion	\$35 billion	\$8 billion

## 1. Amazon Web Services (AWS)



Amazon Web Services (AWS) is an Amazon company that was launched in the year 2002. AWS is the most popular cloud service provider in the world.

Amazon Web Services (AWS) is the world's most comprehensive and broadly adopted cloud platform, offering over 165 fully-featured services from data centers globally. This service is used by millions of customers.

AWS's revenue in the year 2018 was \$25.6 billion with a profit of \$7.2 billion. The revenue is expected to grow to \$33 billion in 2019.

### AWS Services

AWS offers hundreds of services. Some of these include Virtual Private Cloud, EC2, AWS Data Transfer, Simple Storage Service, DynamoDB, Elastic Compute Cloud, AWS Key Management Service, AmazonCloudWatch, Simple Notification Service, Relational Database Service, Route 53, Simple Queue Service, CloudTrail, and Simple Email Service.

The following graphic is a list of the various categories of services available in AWS. The right side of the list includes AWS's featured services.

Featured Services	Featured Services
Analytics	Amazon EC2 Virtual servers in the cloud
Application Integration	Amazon Simple Storage Service (S3) Scalable storage in the cloud
AR & VR	Amazon Aurora High performance managed relational database
AWS Cost Management	Amazon DynamoDB Managed NoSQL database
Blockchain	Amazon RDS Managed relational database service for MySQL, PostgreSQL, Oracle, SQL Server, and MariaDB
Business Applications	AWS Lambda Run code without thinking about servers
Compute	Amazon VPC Isolated cloud resources
Customer Engagement	Amazon Lightsail Launch and manage virtual private servers
Database	Amazon SageMaker Build, train, and deploy machine learning models at scale
Developer Tools	
End User Computing	
Game Tech	
Internet of Things	
Machine Learning	
Management & Governance	
Media Services	
Migration & Transfer	
Mobile	
Networking & Content Delivery	
Robotics	
Satellite	
Security, Identity, & Compliance	
Storage	

## AWS Security:-

Cloud security is the highest priority for AWS. As a customer, you will benefit from a data center and network architecture built to meet the requirements of the most security-sensitive organizations.

AWS security offers services such as infrastructure security, DDoS mitigation, data encryption, inventory and configuration, monitoring and logging, identity and access control, and penetration testing.

## Compliances:-

AWS provides 40+ compliance certifications for the global, US, and other countries. Here is the list of various supported compliance certifications:

### Global



**CSA**  
Cloud Security  
Alliance Controls



**ISO 9001**  
Global Quality  
Standard



**ISO 27001**  
Security Management  
Controls



**ISO 27017**  
Cloud Specific  
Controls



**ISO 27018**  
Personal Data  
Protection



**PCI DSS Level 1**  
Payment Card  
Standards



**SOC 1**  
Audit Controls Report



**SOC 2**  
Security, Availability,  
& Confidentiality  
Report



**SOC 3**  
General Controls  
Report

## United States



**CJIS**  
Criminal Justice  
Information Services



**DoD SRG**  
DoD Data Processing



**FedRAMP**  
Government Data  
Standards



**FERPA**  
Educational Privacy  
Act



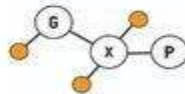
**FFIEC**  
Financial Institutions  
Regulation



**FIPS**  
Government Security  
Standards



**FISMA**  
Federal Information  
Security Management



**GxP**  
Quality Guidelines  
and Regulations



**HIPAA**  
Protected Health  
Information



**HITRUST CSF**  
Health Information  
Trust Alliance  
Common Security  
Framework



**ITAR**  
International Arms  
Regulations



**MPAA**  
Protected Media  
Content



**NIST**  
National Institute of  
Standards and  
Technology



**SEC Rule 17a-4(f)**  
Financial Data  
Standards



**VPAT / Section 508**  
Accessibility  
Standards

## AWS global availability:-

AWS offers the largest global footprint in the market. No other cloud provider offers as many regions or Availability Zones (AZs). This includes 78 AZs within 25 geographic regions around the world. Furthermore, AWS has announced plans for 9 more AZs and three more regions in Cape Town, Jakarta, and Milan.





## AWS Certifications:-

AWS certifications are divided into four major categories – Foundational, Associate, Professional, and Specialty.

### Professional

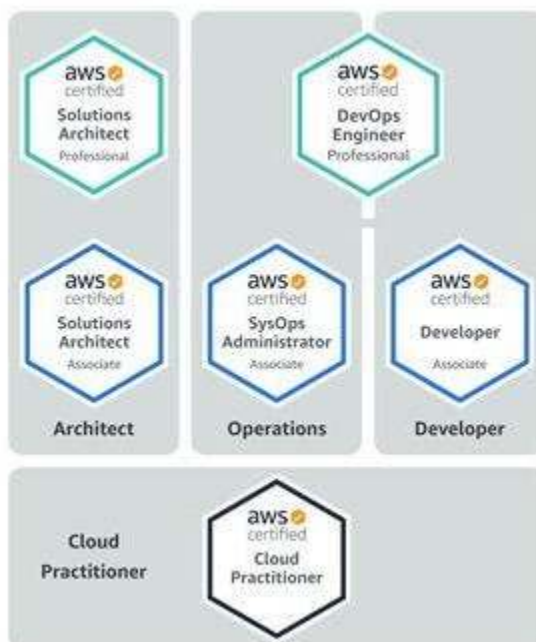
Two years of comprehensive experience designing, operating, and troubleshooting solutions using the AWS Cloud

### Associate

One year of experience solving problems and implementing solutions using the AWS Cloud

### Foundational

Six months of fundamental AWS Cloud and industry knowledge



### Specialty

Technical AWS Cloud experience in the Specialty domain as specified in the exam guide



## 2. Microsoft Azure

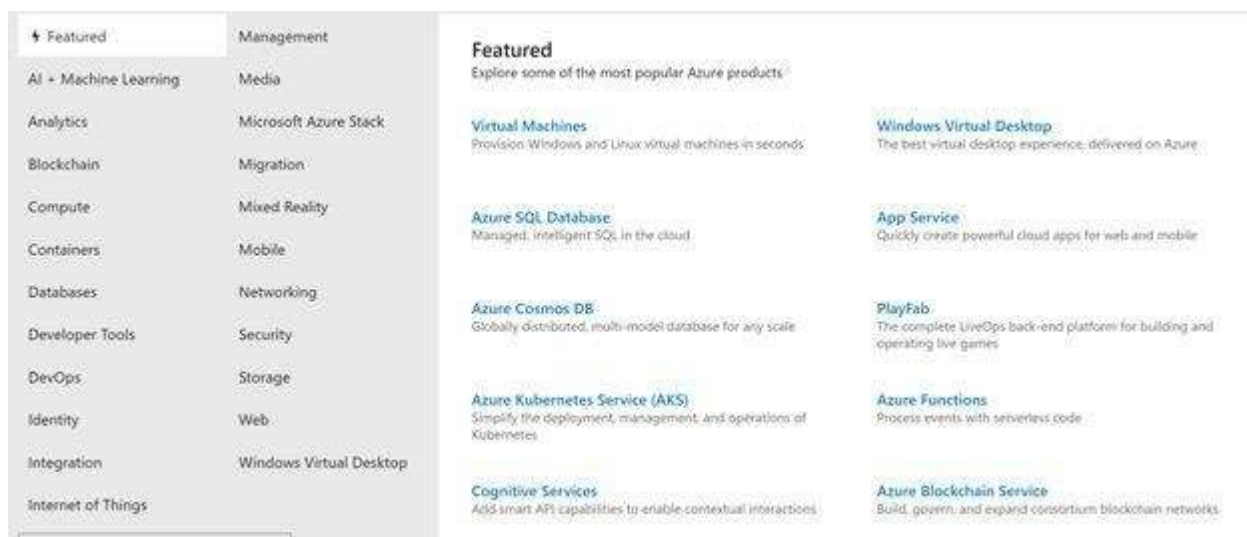


Microsoft Azure is one of the fastest-growing clouds among them all. Azure was launched years after the release of AWS and Google Cloud but is still knocking on the door to become the top cloud services provider. Microsoft Azure recently won a \$10 billion US government contract.

While Microsoft Azure revenue is difficult to predict, Microsoft broke down its revenue of the last quarter into three categories, Productivity and Business Processes, Intelligent Cloud, and Personal Computing. The respective revenue was \$11.0 billion, \$11.4 billion, and \$11.3 billion. Microsoft's Azure revenue is expected to grow between \$33 billion to \$35 billion. This makes Azure one of the most profitable cloud services in the world.

### Azure Services:-

Azure offers hundreds of services within various categories including AI + Machine Learning, Analytics, Blockchain, Compute, Containers, Databases, Developer Tools, DevOps, Identity, Integration, Internet of Things, Management, Media, Microsoft Azure Stack, Migration, Mixed Reality, Mobile, Networking, Security, Storage, Web, and Windows Virtual Desktop.



### Azure, the Intelligent Cloud:-

What makes Azure the most attractive and intelligent is its exclusive offering of Microsoft's previous products and services in the cloud. Azure's cloud supremacy revolves around its intelligence. Azure provides the most advanced and maximum number of intelligent products and services.

- Microsoft's Windows operating system Windows and database SQL Server are now available in the cloud via Windows Virtual Desktop.
- Microsoft's mixed reality technology (products for HoloLens) is also available in the Azure cloud.
- Microsoft's TFS and VSTS are now available in azure via Azure DevOps.
- Microsoft's popular Office suite, enterprise products such as Sharepoint, and Power BI are now available in the cloud as Office 365 and PowerXXX tools. Furthermore, some of the most popular and advanced developer tools and compilers are available in Azure via various UI, workflows, and interfaces.
- Microsoft is a leader in AI + Machine Learning and Microsoft Cognitive Services is one of the company's most advanced offerings.

### Azure Security:-

Azure offers the most advanced security products and services. The following table lists Azure security options:

<b>Azure Active Directory</b> Synchronize on-premises directories and enable single sign-on	<b>Azure Information Protection</b> Better protect your sensitive information—anytime, anywhere
<b>Azure Active Directory Domain Services</b> Join Azure virtual machines to a domain without domain controllers	<b>Key Vault</b> Safeguard and maintain control of keys and other secrets
<b>Security Center</b> Unify security management and enable advanced threat protection across hybrid cloud workloads	<b>Azure Dedicated HSM</b> Manage hardware security modules that you use in the cloud
<b>VPN Gateway</b> Establish secure, cross-premises connectivity	<b>Application Gateway</b> Build secure, scalable, and highly available web front ends in Azure
<b>Azure DDoS Protection</b> Protect your applications from Distributed Denial of Service (DDoS) attacks	<b>Azure Sentinel</b> Standing watch, by your side. Intelligent security analytics for your entire enterprise

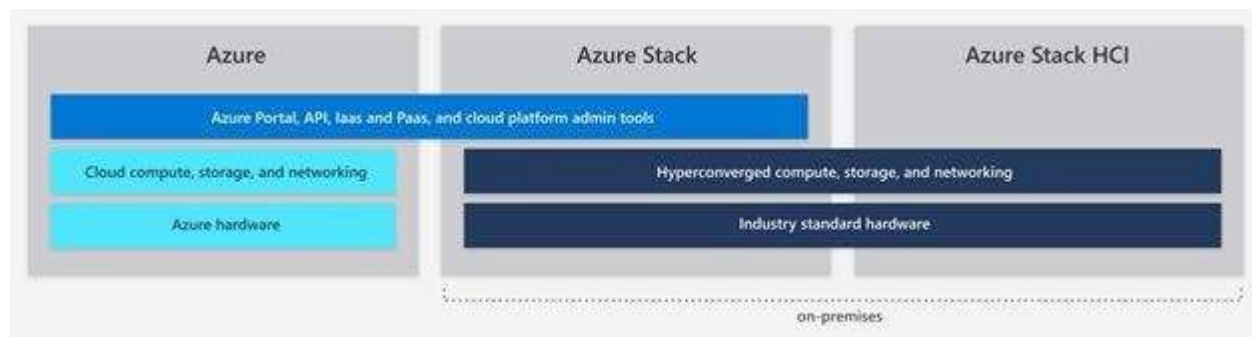
### Azure Compliance:-

Azure offers 90 compliance certifications for global, US government, region-specific, and industry-specific. The following is a list of Azure compliance certifications.

Global	US Government	Region / Country Specific		Industry Specific	
CIS Benchmark	CJIS	BIR 2012 (Netherlands)	IT-Grundschutz (Germany)	23 NVCRR 500 (US)	GLBA (US)
CSA STAR Attestation	CNSSI	CS (Germany)	LOPD (Spain)	AFM/DNS (Netherlands)	GoP (US)
CSA STAR Certification	DFARS	CS Mark Gold (Japan)	MeitY (India)	AMF/ACPR (France)	HIPAA (US)
CSA STAR Self-Assessment	DoD E 2.4.5	Cyber Essentials Plus (US)	MTCS Level 3 (Singapore)	APRA (Australia)	HITRUST (US)
ISO 20000	DoE 10	DJCP (China)	My Number Act (Japan)	CDSA	KMF (Poland)
ISO 22301	EAR	EN 301 549 (EU)	New Zealand CC Framework	CTTC 131 (US)	MARS-E (US)
ISO 27001	FOIA CFR Title 21	ENISA IAF (EU)	PASP (UK)	DPP (UK)	MAS/AES (Singapore)
ISO 27017	FedRAMP	ENS (Spain)	PIPEDA (Canada)	EBA (EU)	MPAA (US)
ISO 27018	FIPS 140-2	EU Model Clauses	PDPA (Argentina)	FACT (UK)	NBS/FSMA (Belgium)
ISO 9001	IRS 1075	EU-US Privacy Shield	TISAX (Germany)	FOIA/PRA (UK)	NEN 7510 (Netherlands)
SOC 1,2,3	ITAR	GB 18030 (China)	TRUCS (China)	FERPA (US)	NHS IG (UK)
WCAG 2.0	NIST CSF	G-Cloud OFFICIAL (UK)		FTIC (US)	OSFI (Canada)
	NIST 800-171	GDPR		FINMA (Switzerland)	PCI DSS
	Section 508 VPATs	HDS (France)		FINRA (US)	RBI/IRDAI (India)
		IRAP (Australia)		FISC (Japan)	SEC 17a-4 (US)
		ISMS (Korea)		FSA (Denmark)	Shared Assessments
					SOX (US)

## Azure Stack:-

Azure Stack is a service of Azure that allows enterprises to run apps in an on-premises environment and perform Azure services in your datacenter. Azure Stack syncs with global Azure and upgrades when new services and updates are available on Azure.



## Azure for Government:-

Azure for Government is an exclusive cloud designed for US government agencies including federal, state, and local.



Azure Government offers government exclusivity. Only US federal, state, local, and tribal governments and their partners have access to this dedicated instance with operations controlled by screened US citizens.

Azure Government offers the broadest and most compliance certifications. It runs on six government-only datacenter regions, all granted an Impacted Level 5 Provisional Authorization.

### Azure global availability:-

Azure offers more data centers around the world than any other cloud provider.



### Azure Certifications

Here is a list of Microsoft Azure certifications.

1. AZ-103: Microsoft Azure Administrator
2. AZ-203: Developing Solutions for Microsoft Azure
3. AZ-300: Microsoft Azure Architect Technologies
4. AZ-301: Microsoft Azure Architect Design
5. AZ-400: Microsoft Azure DevOps Solutions
6. AZ-500: Microsoft Azure Security Technologies
7. AZ-900: Microsoft Azure Fundamentals
8. 70-487: Developing Microsoft Azure and Web Services
9. 70-537: Configuring and Operating a Hybrid Cloud with Microsoft Azure Stack



### 3. IBM Cloud:-



IBM Cloud developed by IBM is a set of cloud computing services for businesses. Similar to other cloud service providers, the IBM cloud includes IaaS, SaaS, and PaaS services via public, private, and hybrid cloud models.

Compute, Network, Storage, Cloud Packs, Management, Security, Database, Analytics, AI, IoT, Mobile, Dev Tools, Blockchain, Integration, Migration, Private Cloud, and VMware.

Annual revenue: \$19.16 billion

### 4. Google Cloud:-



Google cloud platform is Google's cloud. Similar to AWS and Azure, Google Cloud also offers similar services in various categories, including compute, storage, identity, security, database, AI and machine learning, virtualization, DevOps and more.

Here is a list of complete products and services categories Google Cloud Platform services:

AI and Machine Learning, API Management, Compute, Containers, Data Analytics, Databases, Developer Tools, Healthcare and Life Sciences, Hybrid and Multi-cloud, Internet of Things, Management Tools, Media and Gaming, Migration, Networking, Security and Identity, Serverless Computing, and Storage. Google products are also offered in the cloud, including G Suite, Google Maps Platform, Google Hardware, Google Identity, Chrome Enterprise, Android Enterprise, Apigee, Firebase, and Orbitera.

Google Cloud Services are available in 20 regions, 61 zones, and 200+ countries.



Google Cloud's annual revenue is close to \$8 billion.

## Google Cloud Certifications

Here is a list of Google cloud certifications:

1. Associate Cloud Engineer
2. Professional Data Engineer
3. Professional Cloud Architect
4. Professional Cloud Developer
5. Professional Cloud Network Engineer
6. Professional Cloud Security Engineer
7. G Suite

## 5. Oracle Cloud



Oracle cloud platform is the cloud offering of Oracle corporation. Oracle cloud offers IaaS, PaaS, SaaS, and Data as a Service (DaaS).

Oracle offerings include the following:

- Oracle IaaS offerings are Compute, Storage, Networking, Governance, Database, Load Balancing, DNS Monitoring, Ravello, and FastConnect.

- Oracle PaaS offerings are Data Management, Application Development, Integration, Business Analytics, Security, Management, and Content and Enterprise.
- Oracle SaaS offerings are CX, HCM, ERP, SCM, EPM, IoT, Analytics, Data, and Blockchain Applications.
- Oracle DaaS is the Oracle Data Cloud.

## 6. Alibaba Cloud



Alibaba Cloud, founded in 2009, is registered and headquartered in Singapore. It was initially built to serve Alibaba's own e-commerce ecosystem and is now offered to the public. Alibaba Cloud is the largest cloud provider in China.

Alibaba offers various products and services in various categories, including Elastic Computing, Storage and CDN, Networking, Database Services, Security, Monitoring and Management, Domains and Websites, Analytics and Data Technology, Application Services, Media Services, Middleware, Cloud Communication, Apsara Stack, and Internet of Things.

Alibaba Cloud is available in 19 regions and 56 availability zones around the globe.

Alibaba Cloud's revenue is \$4.5 million annually.

### Which is the best cloud?

As of July 2019, according to Gartner Magic Quadrant for Cloud Infrastructure as a Service (IaaS), AWS leads all the way as a leader in its ability to execute, but Microsoft's Azure leads as a visionary. Google is third in the race followed by Oracle, Alibaba, and IBM.

While we can sit here and discuss which cloud is better, one size may not fit all. While there are so.



## **The Verdict**

Microsoft Azure and AWS are necks to neck in the race of cloud supremacy. Both offer similar products, services, and costs. However, Google, IBM, and other clouds are getting better each day. If I were to choose a cloud provider for my business, it would have to be Azure or AWS. The reason is obvious: both companies are way ahead in the race. You cannot go wrong with either of these.

If there are some specific tools and services offered by other clouds, you may end up deciding upon alternatives as well. For example, IBM Watson is a unique product used in IBM's cloud service. Most large enterprises use multiple clouds. One of my clients uses AWS for file storage and messaging, Azure for DevOps, database and developer tools, and Google's cloud for documents and emails. One size doesn't fit all. You need to figure out what your needs are, and which cloud fits you the best.

### **Virtual Private Cloud – Scaling (Horizontal and Vertical) :-**

#### **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 providers include AWS, Google Cloud Platform, and Microsoft Azure, among others.

The technical term for multiple separate customers accessing the same cloud infrastructure is "multitenancy" (see [What Is Multitenancy?](#) to learn more).

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.

## How does Cloudflare support virtual private clouds?

Cloudflare makes it easy to use any cloud service by providing a single plane of control for performance, security, and reliability services, including bot management, DNS, SSL, and

DDoS protection (even for layer 3 traffic). The full Cloudflare stack sits in front of any cloud deployment and accelerates good traffic while blocking bad traffic.

### **Horizontal and Vertical Scaling Strategies:-**

The cloud has dramatically simplified these scaling problems by making it easier to scale up or down and out or in. Primarily, there are two ways to scale in the cloud: horizontally or vertically. When you scale horizontally, you are scaling out or in, which refers to the number of provisioned resources. When you scale vertically, it's often called scaling up or down, which refers to the power and capacity of an individual resource.

### **What are the differences between horizontal and vertical scaling in the cloud?**

- **Horizontal scaling** refers to provisioning additional servers to meet your needs, often splitting workloads between servers to limit the number of requests any individual server is getting. Horizontal scaling in cloud computing means adding additional instances instead of moving to a larger instance size.
- **Vertical scaling** refers to adding more or faster CPUs, memory, or I/O resources to an existing server, or replacing one server with a more powerful server. In a data center, administrators traditionally achieved vertical scaling by purchasing a new, more powerful server and discarding or repurposing the old one. Today's cloud architects can accomplish AWS vertical scaling and Microsoft Azure vertical scaling by changing instance sizes. AWS and Azure cloud services have many different instance sizes, so vertical scaling in cloud computing is possible for everything from EC2 instances to RDS databases.

### **Horizontal vs. Vertical Scaling Pros and Cons**

#### **Pros and cons of horizontal scaling:**

- **Pros:** Horizontal scaling is much easier to accomplish without downtime. Horizontal scaling is also easier than vertical scaling to manage automatically. Limiting the number of requests any instance gets at one time is good for performance, no matter how large the instance. Provisioning additional instances also means having greater redundancy in the rare event of an outage.
- **Cons:** Depending on the number of instances you need, your costs may be higher. Additionally, without a load balancer in place, your machines run the risk of being over-utilized, which could lead to an outage. However, with public cloud platforms, you can pay attention to discounts for Reserved Instances (RIs) if you're able to predict when you require more compute power. Following cloud cost management best practices can help you efficiently scale in or out.

#### **Pros and cons of vertical scaling:**

- **Pros:** In the cloud, vertical scaling means changing the sizes of cloud resources, rather than purchasing more, to match them to the workload. This process is known as right

sizing. For example, right sizing in AWS can refer to the CPU, memory, storage, and networking capacity of instances and storage classes. Right sizing is one of the most effective ways to control cloud costs. When done correctly, right sizing can help lower costs of vertically scaled resources.

- **Cons:** In general, vertical scaling can cost more. Why is vertical scaling expensive? When resources aren't right sized correctly — or at all — costs can skyrocket. There's also downtime to consider. Even in a cloud environment, scaling vertically usually requires making an application unavailable for some amount of time. Therefore, environments or applications that can't have downtime would typically benefit more from horizontal scalability by provisioning additional resources instead of increasing capacity for existing resources.

### Which Is Better: Horizontal or Vertical Scaling?

The decision to scale horizontally or vertically in the cloud depends upon the requirements of your data. Remember that scaling continues to be a challenge, even in cloud environments. All parts of your application need to scale, from the compute resources to database and storage resources. Neglecting any pieces of the scaling puzzle can lead to unplanned downtime or worse. The best solution might be a combination of vertical scaling in order to find the ideal capacity of each instance and then horizontal scaling to handle spikes in demand, while ensuring uptime.

### Types of Cloud Scalability: Manual vs. Scheduled vs. Automatic Scaling

What also matters is *how* you scale. Three basic ways to scale in a cloud environment include manual scaling, scheduled scaling, and automatic scaling.

- **Manual Scaling** - Manual scaling is just as it sounds. It requires an engineer to manage scaling up and out or down and in. In the cloud, both vertical and horizontal scaling can be accomplished with the push of a button, so the actual scaling isn't terribly difficult when compared to managing a data center. However, because it requires a team member's attention, manual scaling cannot take into account all the minute-by-minute fluctuations in demand seen by a normal application. This also can lead to human error. An individual might forget to scale back down, leading to extra charges.
- **Scheduled Scaling** - Scheduled scaling solves some of the problems with manual scaling. This makes it easier to tailor your provisioning to your actual usage without requiring a team member to make the changes manually every day. If you know when peak activity occurs, you can schedule scaling based on your usual demand curve. For example, you can scale out to ten instances from 5 p.m. to 10 p.m., then back into two instances from 10 p.m. to 7 a.m., and then back out to five instances until 5 p.m. Look for a cloud management platform with Heat Maps that can visually identify such peaks and valleys of usage.

## **Automatic Scaling:-**

Automatic scaling (also known as Auto Scaling) is when your compute, database, and storage resources scale automatically based on predefined rules. For example, when metrics like vCPU, memory, and network utilization rates go above or below a certain threshold, you can scale out or in. Auto scaling makes it possible to ensure your application is always available — and always has enough resources provisioned to prevent performance problems or outages — without paying for far more resources than you are actually using.

## **Horizontal and Vertical Scaling for Cloud Cost Management:-**

Scaling is one of the most important components of cloud cost management. Organizations can follow a number of best practices around scaling instances. Namely, they need to know when to use horizontal and vertical scaling and where they can automate their cloud scalability. Horizontal and vertical scaling in AWS, for example, means paying attention to the *number* of EC2 instances provisioned as well as the *sizes* of those instances.

To scale horizontally in AWS, begin only with the resources you need and design your architecture to automatically respond to changes in demand. Amazon EC2 Auto Scaling can add or remove EC2 instances in response to changing demand. Two subsets of Auto Scaling in AWS include dynamic scaling, which can be configured based on policies you set, or predictive scaling, which can schedule the right number of instances based on predicted demand. Having a tool that can terminate instances automatically when they're not in use can also help organizations save money.

When it comes to vertical scalability, organizations should pay attention to their right sizing strategy. Right sizing instances, or choosing the correct instance sizes based on your actual application utilization, is one of the easiest ways to reduce cloud costs without affecting performance in any way. There are also some cost management strategies, like Reserved Instance purchases, that take away some of the ability to scale in or down, because you're committing to using certain amounts and types of resources for one to three years. When you're looking for ways to reduce costs, it's important to understand your current usage patterns and utilization rates to make the best decisions about how to strike a balance between total scaling flexibility and cost management strategies like RI purchases.

## **Managing Your Cloud Scaling Strategy:-**

Managing scaling correctly is the key to ensuring you always have enough resources without over-provisioning and wasting your cloud budget. The ability to automatically scale is one of the most attractive parts of moving to a cloud environment. When used correctly, auto scaling can ensure you're only paying for the resources you actually use. As you figure out the best strategy for managing scaling, it's important to understand your historical usage patterns, how RI purchases affect scaling, and whether manual, scheduled, or automatic scaling is best for your use case.

## Virtual Machines, Ethernet and Switches:-

### What is a virtual machine?

A virtual machine (VM) is a software-based computer that exists within another computer's operating system, often used for the purposes of testing, backing up data, or running SaaS applications. To grasp how VMs work, it's important to first understand how computer software and hardware are typically integrated by an operating system.

### What are virtual machines used for?

Common use cases for virtual machines on single computers include:

- **Testing** - Software developers often want to test their applications in different environments. They can use virtual machines to run their applications in various OSes on one computer. This is simpler and more cost-effective than testing on several different physical machines.
- **Running software designed for other OSes** - Although certain software applications are only available for a single platform, a VM can run software designed for a different OS. For example, a Mac user who wants to run software designed for Windows can run a Windows VM on their Mac host.
- **Running outdated software** - Some pieces of older software can't be run in modern OSes. Users who want to run these applications can run an old OS on a virtual machine.
- **Browser isolation** - Browser isolation is the practice of 'isolating' web browser activity away from the rest of a computer's operating system to keep malware from affecting the computer's other files and programs. Some browser isolation tools use VMs to establish this isolation — though this approach can slow down browsing activity.

### How does cloud computing use virtual machines?

Several cloud providers offer virtual machines to their customers. These virtual machines typically live on powerful servers that can act as a host to multiple VMs and can be used for a variety of reasons that wouldn't be practical with a locally-hosted VM. These include:

- **Running SaaS applications** - Software-as-a-Service, or SaaS for short, is a cloud-based method of providing software to users, in which an application is served to user over the Internet rather than running on their computers. Often, it is virtual machines in the cloud that do the computation for SaaS applications as well as delivering them to users. If the cloud provider has a geographically distributed network edge, then the application will run closer to the user, resulting in faster performance.
- **Backing up data** - Cloud-based VM services are popular for backing up data, because the data can be accessed from anywhere. Plus, cloud VMs provide better redundancy, require less maintenance, and generally scale better than physical data centers. (For example, it's relatively easy to buy an extra gigabyte of storage space from a cloud VM provider, but much more difficult to build a new local data server for that extra gigabyte of data.)

- **Hosting services like email and access management** - Hosting these services on cloud VMs is generally faster and more cost-effective, and helps minimize maintenance and offload security concerns as well.
- **Browser isolation** - Some browser isolation tools use cloud VMs to run web browsing activity and deliver safe content to users via a secure Internet connection.

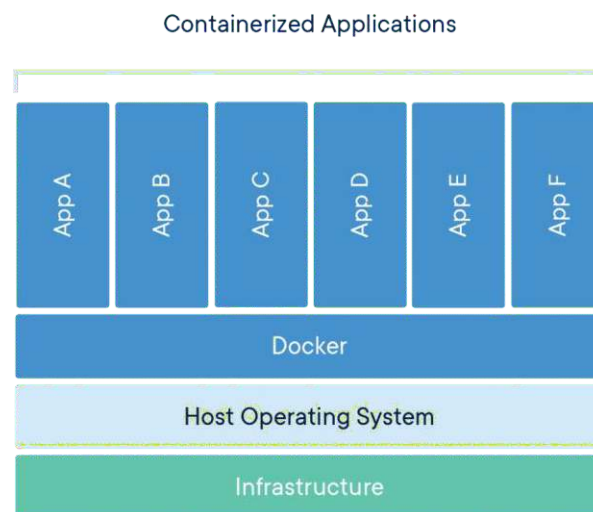
### Docker Container:-

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Container images become containers at runtime and in the case of Docker containers – images become containers when they run on Docker Engine. Available for both Linux and Windows-based applications, containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

### Docker containers that run on Docker Engine:

- ❖ **Standard:** Docker created the industry standard for containers, so they could be portable anywhere
- ❖ **Lightweight:** Containers share the machine's OS system kernel and therefore do not require an OS per application, driving higher server efficiencies and reducing server and licensing costs
- ❖ **Secure:** Applications are safer in containers and Docker provides the strongest default isolation capabilities in the industry



## Docker Containers Are Everywhere: Linux, Windows, Data center, Cloud, Serverless, etc.

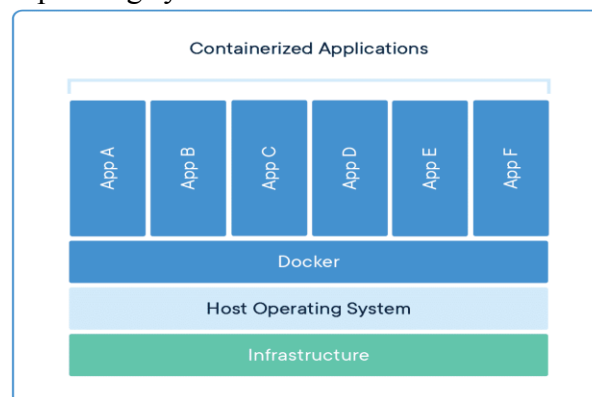
- Docker container technology was launched in 2013 as an open source Docker Engine.
- It leveraged existing computing concepts around containers and specifically in the Linux world, primitives known as cgroups and namespaces. Docker's technology is unique because it focuses on the requirements of developers and systems operators to separate application dependencies from infrastructure.
- Success in the Linux world drove a partnership with Microsoft that brought Docker containers and its functionality to Windows Server.
- Technology available from Docker and its open source project, Moby has been leveraged by all major data center vendors and cloud providers. Many of these providers are leveraging Docker for their container-native IaaS offerings. Additionally, the leading open source serverless frameworks utilize Docker container technology.

## Comparing Containers and Virtual Machines:-

Containers and virtual machines have similar resource isolation and allocation benefits, but function differently because containers virtualize the operating system instead of hardware. Containers are more portable and efficient.

## Containers:-

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems.



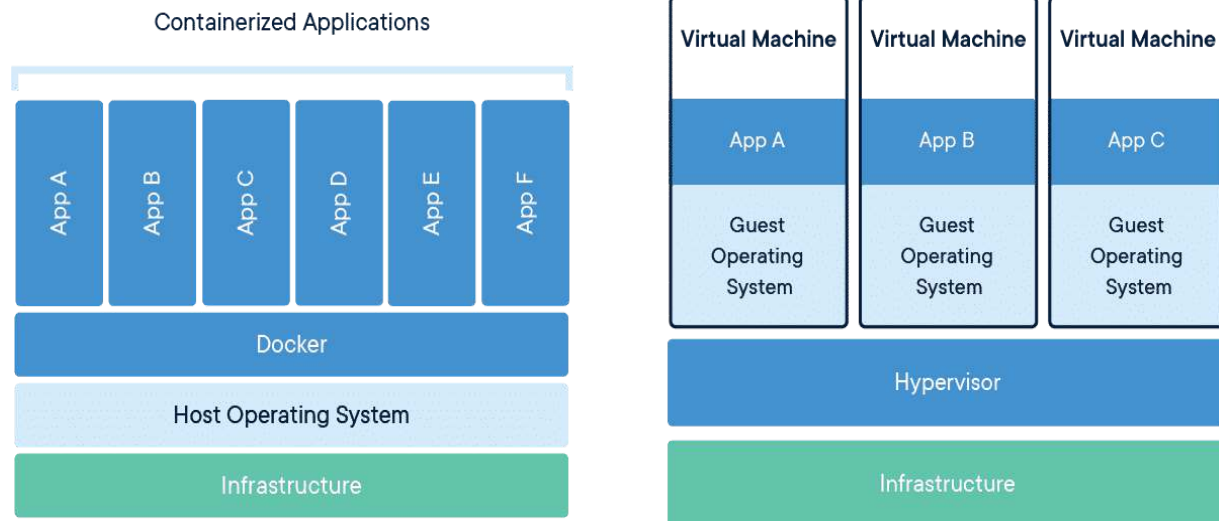
## Virtual Machines:-

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries – taking up tens of GBs. VMs can also be slow to boot.



## Containers and Virtual Machines Together

- Containers and VMs used together provide a great deal of flexibility in deploying and managing app



## Container Standards and Industry Leadership:-

The launch of Docker in 2013 jump started a revolution in application development – by democratizing software containers. Docker developed a Linux container technology – one that is portable, flexible and easy to deploy. Docker open sourced libcontainer and partnered with a worldwide community of contributors to further its development. In June 2015, Docker donated the container image specification and runtime code now known as runc, to the Open Container Initiative (OCI) to help establish standardization as the container ecosystem grows and matures.

Following this evolution, Docker continues to give back with the containerd project, which Docker donated to the Cloud Native Computing Foundation (CNCF) in 2017. containerd is an industry-standard container runtime that leverages runc and was created with an emphasis on simplicity, robustness and portability. Containerd is the core container runtime of the Docker Engine.

## Kubernetes:- What

### is Kubernetes?

Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

The name Kubernetes originates from Greek, meaning helmsman or pilot. K8s as an abbreviation results from counting the eight letters between the "K" and the "s". Google open-sourced the Kubernetes project in 2014. Kubernetes combines over 15 years of Google's experience running production workloads at scale with best-of-breed ideas and practices from the community.

**Traditional deployment era:** Early on, organizations ran applications on physical servers. There was no way to define resource boundaries for applications in a physical server, and this caused resource allocation issues. For example, if multiple applications run on a physical server, there can be instances where one application would take up most of the resources, and as a result, the other applications would underperform. A solution for this would be to run each application on a different physical server. But this did not scale as resources were underutilized, and it was expensive for organizations to maintain many physical servers.

**Virtualized deployment era:** As a solution, virtualization was introduced. It allows you to run multiple Virtual Machines (VMs) on a single physical server's CPU. Virtualization allows applications to be isolated between VMs and provides a level of security as the information of one application cannot be freely accessed by another application.

Virtualization allows better utilization of resources in a physical server and allows better scalability because an application can be added or updated easily, reduces hardware costs, and much more. With virtualization you can present a set of physical resources as a cluster of disposable virtual machines.

Each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware.

**Container deployment era:** Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications. Therefore, containers are considered lightweight. Similar to a VM, a container has its own filesystem, share of CPU, memory, process space, and more. As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions.

Containers have become popular because they provide extra benefits, such as:

- Agile application creation and deployment: increased ease and efficiency of container image creation compared to VM image use.
- Continuous development, integration, and deployment: provides for reliable and frequent container image build and deployment with quick and efficient rollbacks (due to image immutability).
- Dev and Ops separation of concerns: create application container images at build/release time rather than deployment time, thereby decoupling applications from infrastructure.

- **Observability:** not only surfaces OS-level information and metrics, but also application health and other signals.
- **Environmental consistency** across development, testing, and production: Runs the same on a laptop as it does in the cloud.
- **Cloud and OS distribution portability:** Runs on Ubuntu, RHEL, CoreOS, on-premises, on major public clouds, and anywhere else.
- **Application-centric management:** Raises the level of abstraction from running an OS on virtual hardware to running an application on an OS using logical resources.
- **Loosely coupled, distributed, elastic, liberated micro-services:** applications are broken into smaller, independent pieces and can be deployed and managed dynamically – not a monolithic stack running on one big single-purpose machine.
- **Resource isolation:** predictable application performance.
- **Resource utilization:** high efficiency and density.

## Why you need Kubernetes and what it can do

Containers are a good way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start. Wouldn't it be easier if this behavior was handled by a system?

That's how Kubernetes comes to the rescue! Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more. For example, Kubernetes can easily manage a canary deployment for your system.

Kubernetes provides you with:

- **Service discovery and load balancing** Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
- **Storage orchestration** Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks** You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.
- **Automatic bin packing** You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
- **Self-healing** Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.

- **Secret and configuration management** Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.

## What Kubernetes is not?

Kubernetes is not a traditional, all-inclusive PaaS (Platform as a Service) system. Since Kubernetes operates at the container level rather than at the hardware level, it provides some generally applicable features common to PaaS offerings, such as deployment, scaling, load balancing, and lets users integrate their logging, monitoring, and alerting solutions. However, Kubernetes is not monolithic, and these default solutions are optional and pluggable. Kubernetes provides the building blocks for building developer platforms, but preserves user choice and flexibility where it is important.

## Kubernetes:

- Does not limit the types of applications supported. Kubernetes aims to support an extremely diverse variety of workloads, including stateless, stateful, and data-processing workloads. If an application can run in a container, it should run great on Kubernetes.
- Does not deploy source code and does not build your application. Continuous Integration, Delivery, and Deployment (CI/CD) workflows are determined by organization cultures and preferences as well as technical requirements.
- Does not provide application-level services, such as middleware (for example, message buses), data-processing frameworks (for example, Spark), databases (for example, MySQL), caches, nor cluster storage systems (for example, Ceph) as built-in services. Such components can run on Kubernetes, and/or can be accessed by applications running on Kubernetes through portable mechanisms, such as the Open Service Broker.
- Does not dictate logging, monitoring, or alerting solutions. It provides some integrations as proof of concept, and mechanisms to collect and export metrics.
- Does not provide nor mandate a configuration language/system (for example, Jsonnet). It provides a declarative API that may be targeted by arbitrary forms of declarative specifications.
- Does not provide nor adopt any comprehensive machine configuration, maintenance, management, or self-healing systems.
- Additionally, Kubernetes is not a mere orchestration system. In fact, it eliminates the need for orchestration. The technical definition of orchestration is execution of a defined workflow: first do A, then B, then C. In contrast, Kubernetes comprises a set of independent, composable control processes that continuously drive the current state towards the provided desired state. It shouldn't matter how you get from A to C. Centralized control is also not required. This results in a system that is easier to use and more powerful, robust, resilient, and extensible.

## What's next

### Kubernetes Components

- A Kubernetes cluster consists of the components that represent the control plane and a set of machines called nodes.
- When you deploy Kubernetes, you get a cluster.
- A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.
- The worker node(s) host the Pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.
- This document outlines the various components you need to have for a complete and working Kubernetes cluster.
- The components of a Kubernetes cluster

### Control Plane Components:-

- The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new pod when a deployment's `replicas` field is unsatisfied).
- Control plane components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all control plane components on the same machine, and do not run user containers on this machine. See [Creating Highly Available clusters with kubeadm](#) for an example control plane setup that runs across multiple machines.

### kube-apiserver :-

The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane. The main implementation of a Kubernetes API server is kube-apiserver. kube-apiserver is designed to scale horizontally—that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

### Etd:-

Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data. If your Kubernetes cluster uses etcd as its backing store, make sure you have a backup plan for those data. You can find in-depth information about etcd in the [official documentation](#).

### kube-scheduler:-

Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on. Factors taken into account for scheduling decisions include:

individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

### **kube-controller-manager:-**

Control plane component that runs controller processes. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

Some types of these controllers are:

- **Node controller:** Responsible for noticing and responding when nodes go down.
- **Job controller:** Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.
- **Endpoints controller:** Populates the Endpoints object (that is, joins Services & Pods).
- **Service Account & Token controllers:** Create default accounts and API access tokens for new namespaces.

### **cloud-controller-manager:-**

A Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster.

The cloud-controller-manager only runs controllers that are specific to your cloud provider. If you are running Kubernetes on your own premises, or in a learning environment inside your own PC, the cluster does not have a cloud controller manager.

As with the kube-controller-manager, the cloud-controller-manager combines several logically independent control loops into a single binary that you run as a single process. You can scale horizontally (run more than one copy) to improve performance or to help tolerate failures.

### **The following controllers can have cloud provider dependencies:**

- **Node controller:** For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
- **Route controller:** For setting up routes in the underlying cloud infrastructure
- **Service controller:** For creating, updating and deleting cloud provider load balancers

### **Node Components:-**

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

### **Kubelet:-**

- An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.
- The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

### **kube-proxy:-**

kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept. kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster. kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

### **Container runtime:-**

The container runtime is the software that is responsible for running containers. Kubernetes supports container runtimes such as containerd, CRI-O, and any other implementation of the Kubernetes CRI (Container Runtime Interface).

### **Addons:-**

Addons use Kubernetes resources (DaemonSet, Deployment, etc) to implement cluster features. Because these are providing cluster-level features, namespaced resources for addons belong within the `kube-system` namespace. Selected addons are described below; for an extended list of available addons, please see Addons.

### **DNS:-**

While the other addons are not strictly required, all Kubernetes clusters should have cluster DNS, as many examples rely on it. Cluster DNS is a DNS server, in addition to the other DNS server(s) in your environment, which serves DNS records for Kubernetes services.

Containers started by Kubernetes automatically include this DNS server in their DNS searches.

### **Web UI (Dashboard)**

Dashboard is a general purpose, web-based UI for Kubernetes clusters. It allows users to manage and troubleshoot applications running in the cluster, as well as the cluster itself.

### **Container Resource Monitoring:-**

Container Resource Monitoring records generic time-series metrics about containers in a central database, and provides a UI for browsing that data.

### **Cluster-level Logging:-**

## A cluster-le

vel logging mechanism is responsible for saving container logs to a central log store with search/browsing interface.

## The Kubernetes API:-

The Kubernetes API lets you query and manipulate the state of objects in Kubernetes. The core of Kubernetes' control plane is the API server and the HTTP API that it exposes. Users, the different parts of your cluster, and external components all communicate with one another through the API server. The core of Kubernetes' control plane is the API server. The API server exposes an HTTP API that lets end users, different parts of your cluster, and external components communicate with one another.

The Kubernetes API lets you query and manipulate the state of API objects in Kubernetes (for example: Pods, Namespaces, ConfigMaps, and Events). Most operations can be performed through the kubectl command-line interface or other command-line tools, such as kubeadm, which in turn use the API. However, you can also access the API directly using REST calls. Consider using one of the client libraries if you are writing an application using the Kubernetes API.

## OpenAPI specification:-

Complete API details are documented using OpenAPI.

## OpenAPI V2:-

The Kubernetes API server serves an aggregated OpenAPI v2 spec via the /openapi/v2 endpoint. You can request the response format using request headers as follows:

Header	Possible values	Notes
Accept-Encoding	Gzip	<i>not supplying this header is also acceptable</i>
	application/com.github.proto-openapi.spec.v2@v1.0+protobuf	<i>mainly for intra-cluster use</i>
Accept	application/json	<i>default</i>
	*	<i>serves application/json</i>

Kubernetes implements an alternative Protobuf based serialization format that is primarily intended for intra-cluster communication. For more information about this format, see the Kubernetes Protobuf serialization design proposal and the Interface Definition Language (IDL) files for each schema located in the Go packages that define the API objects.



## OpenAPI V3

**FEATURE STATE:** Kubernetes v1.24 [beta]

Kubernetes v1.24 offers beta support for publishing its APIs as OpenAPI v3; this is a beta feature that is enabled by default. You can disable the beta feature by turning off the feature gate named `OpenAPIV3` for the kube-apiserver component.

A discovery endpoint `/openapi/v3` is provided to see a list of all group/versions available. This endpoint only returns JSON. These group/versions are provided in the following format:

```
{
  "paths": {
    ...
    "api/v1": {
      "serverRelativeURL":           },
    "apis/admissionregistration.k8s.io/v1": {
      "serverRelativeURL":           },
    ...
  }
}
```

**ALL THE BEST**

\*\*\*\*\*