

Full Stack Courses By Developersparks

FULL STACK DEVELOPMENT

INDEX

SNO	UNIT	TOPIC	PAGE NO
1	I	Web development Basics - HTML	1
2	I	HTML, CSS	24
3	I	Web servers	27
4	I	UNIX CLI Version control - Git & Github	28
5	II	Javascript basics OOPS Aspects of JavaScript Memory usage and Functions in JS	31
6	II	jQuery, AJAX for data exchange with server jQuery Framework jQuery events	35
7	II	JSON data format.	44
8	III	Introduction to React	47
9	III	React Router and Single Page Applications React Forms	61
10	III	Flow Architecture	66
11	III	Introduction to Redux More Redux	69
12	III	Client-Server Communication	75
13	IV	Java Web Development	78
14	IV	Model View Controller (MVC) Pattern	78
15	IV	MVC Architecture using Spring RESTful API	83
16	IV	Spring Framework Building an application using Maven	87
17	V	Relational schemas	98
18	V	Normalization Structured Query Language (SQL)	102
19	V	Data persistence using Spring JDBC Agile development principles	105
20	V	Deploying application in Cloud	120

(R20A0516) FULL STACK DEVELOPMENT

COURSE OBJECTIVES:

1. To become knowledgeable about the most recent web development technologies.
2. Idea for creating two tier and three tier architectural web applications.
3. Design and Analyse real time web applications.
4. Constructing suitable client and server side applications.
5. To learn core concept of both front end and back end programming.

UNIT - I

Web Development Basics: Web development Basics - HTML & Web servers Shell - UNIX CLI Version control - Git & Github HTML, CSS

UNIT - II

Frontend Development: Javascript basics OOPS Aspects of JavaScript Memory usage and Functions in JS AJAX for data exchange with server jQuery Framework jQuery events, UI components etc. JSON data format.

UNIT - III

REACT JS: Introduction to React React Router and Single Page Applications React Forms, Flow Architecture and Introduction to Redux More Redux and Client-Server Communication

UNIT - IV

Java Web Development: JAVA PROGRAMMING BASICS, Model View Controller (MVC) Pattern MVC Architecture using Spring RESTful API using Spring Framework Building an application using Maven

UNIT - V

Databases & Deployment: Relational schemas and normalization Structured Query Language (SQL) Data persistence using Spring JDBC Agile development principles and deploying application in Cloud

TEXT BOOKS:

1. Web Design with HTML, CSS, JavaScript and JQuery Set Book by Jon Duckett Professional JavaScript for Web Developers Book by Nicholas C. Zakas
2. Learning PHP, MySQL, JavaScript, CSS & HTML5: A Step-by-Step Guide to Creating Dynamic Websites by Robin Nixon
3. Full Stack JavaScript: Learn Backbone.js, Node.js and MongoDB. Copyright © 2015 BY AZAT MARDAN

REFERENCE BOOKS:

1. Full-Stack JavaScript Development by Eric Bush.
2. Mastering Full Stack React Web Development Paperback – April 28, 2017 by Tomasz Dyl , Kamil Przeorski , Maciej Czarnecki

COURSE OUTCOMES:

1. Develop a fully functioning website and deploy on a web server.
2. Gain Knowledge about the front end and back end Tools
3. Find and use code packages based on their documentation to produce working results in a project.
4. Create web pages that function using external data.
5. Implementation of web application employing efficient database access.

UNIT - I

Web Development Basics: Web development Basics - HTML & Web servers Shell - UNIX CLI Version control - Git & Github HTML, CSS

HTML is an acronym which stands for **Hyper Text Markup Language** which is used for creating web pages and web applications. Let's see what is meant by Hypertext Markup Language, and Web page.

Hyper Text: HyperText simply means "Text within Text." A text has a link within it, is a hypertext. Whenever you click on a link which brings you to a new webpage, you have clicked on a hypertext. HyperText is a way to link two or more web pages (HTML documents) with each other.

Markup language: A markup language is a computer language that is used to apply layout and formatting conventions to a text document. Markup language makes text more interactive and dynamic. It can turn text into images, tables, links, etc.

Web Page: A web page is a document which is commonly written in HTML and translated by a web browser. A web page can be identified by entering an URL. A Web page can be of the static or dynamic type. **With the help of HTML only, we can create static web pages.**

Hence, HTML is a markup language which is used for creating attractive web pages with the help of styling, and which looks in a nice format on a web browser. An HTML document is made of many HTML tags and each HTML tag contains different content.

Let's see a simple example of HTML.

```
<!DOCTYPE>
<html>
<head>
<title>Web page title</title>
</head>
<body>
<h1>Write Your First Heading</h1>
<p>Write Your First Paragraph.</p>
</body>
</html>
```

Description of HTML Example

<!DOCTYPE>: It defines the document type or it instruct the browser about the version of HTML.

<html >: This tag informs the browser that it is an HTML document. Text between html tag describes the web document. It is a container for all other elements of HTML except <!DOCTYPE>

<head>: It should be the first element inside the <html> element, which contains the metadata (information about the document). It must be closed before the body tag opens.

<title>: As its name suggested, it is used to add title of that HTML page which appears at the top of the browser window. It must be placed inside the head tag and should close immediately. (Optional)

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

<h1>: Text between <h1> tag describes the first level heading of the webpage.

<p>: Text between <p> tag describes the paragraph of the webpage.

HTML Tags

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

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

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

Syntax

<tag> content </tag>

HTML Tag Examples

<p> Paragraph Tag </p>

<h2> Heading Tag </h2>

 Bold Tag

<i> *Italic Tag* </i>

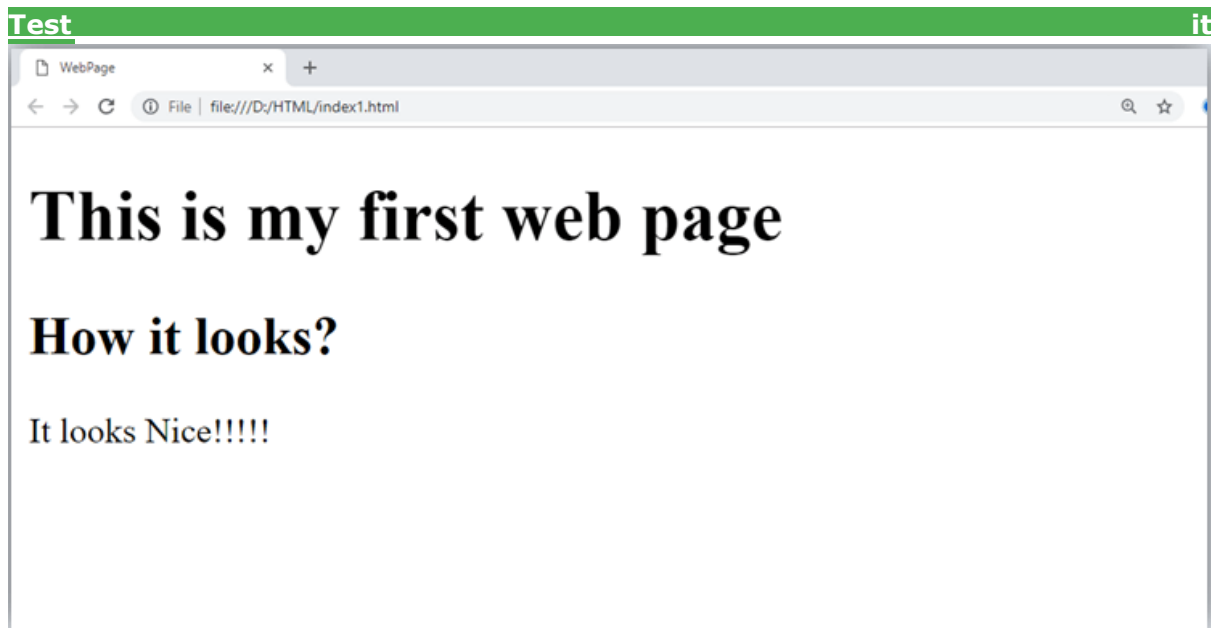
<u> Underline Tag </u>

HTML Elements

An HTML file is made of elements. These elements are responsible for creating web pages and define content in that webpage. An element in HTML usually consist of a start tag <tag name>, close tag </tag name> and content inserted between them. **Technically, an element is a collection of start tag, attributes, end tag, content between them.**

Example

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



HTML Heading

A HTML heading or HTML h tag can be defined as a title or a subtitle which you want to display on the webpage. When you place the text within the heading tags `<h1>.....</h1>`, it is displayed on the browser in the bold format and size of the text depends on the number of heading.

There are six different HTML headings which are defined with the `<h1>` to `<h6>` tags, from highest level h1 (main heading) to the least level h6 (least important heading).

h1 is the largest heading tag and h6 is the smallest one. So h1 is used for most important heading and h6 is used for least important.

Headings in HTML helps the search engine to understand and index the structure of web page.

Note: The main keyword of the whole content of a webpage should be display by h1 heading tag.

See this example:

1. `<h1>`Heading no. 1`</h1>`
2. `<h2>`Heading no. 2`</h2>`
3. `<h3>`Heading no. 3`</h3>`
4. `<h4>`Heading no. 4`</h4>`

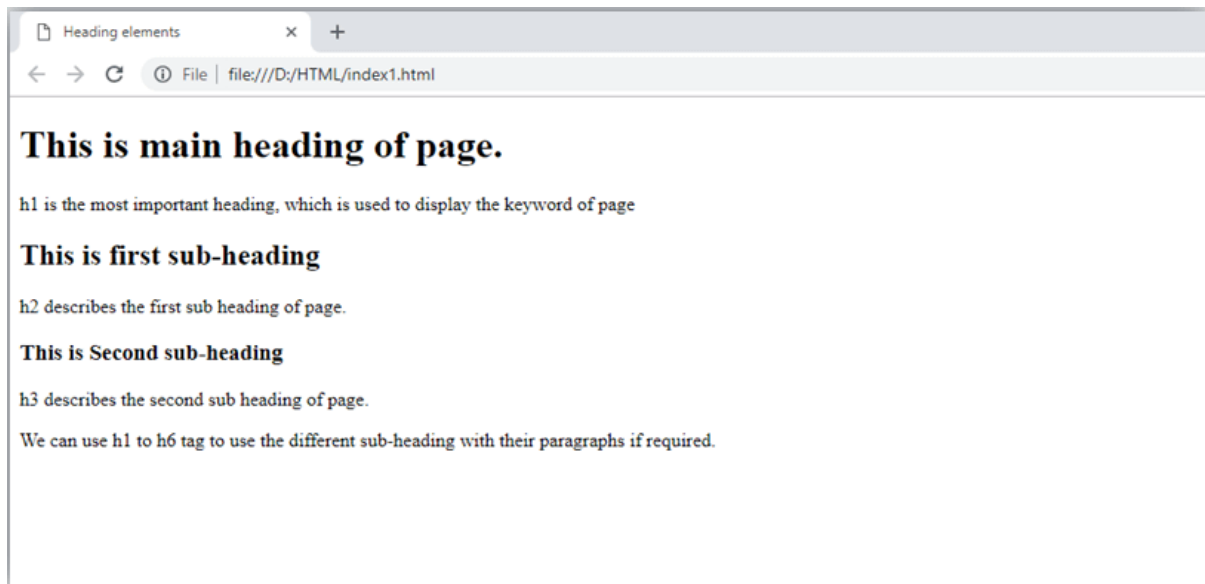
5. `<h5>`Heading no. 5`</h5>`
6. `<h6>`Heading no. 6`</h6>`

Output:

Heading no. 1
Heading no. 2
Heading no. 3
Heading no. 4
Heading no. 5
Heading no. 6

Example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Heading elements</title>
  </head>
  <body>
    <h1>This is main heading of page. </h1>
    <p>h1 is the most important heading, which is used to display the keyword o
f page </p>
    <h2>This is first sub-heading</h2>
    <p>h2 describes the first sub heading of page. </p>
    <h3>This is Second sub-heading</h3>
    <p>h3 describes the second sub heading of page.</p>
    <p>We can use h1 to h6 tag to use the different sub-
heading with their paragraphs if
      required.
    </p>
  </body>
</html>
```

Output:

HTML Anchor

The **HTML anchor tag** defines *a hyperlink that links one page to another page*. It can create hyperlink to other web page as well as files, location, or any URL. The "href" attribute is the most important attribute of the HTML a tag. and which links to destination page or URL.

href attribute of HTML anchor tag

The href attribute is used to define the address of the file to be linked. In other words, it points out the destination page.

The syntax of HTML anchor tag is given below.

```
<a href = "....."> Link Text </a>
```

Let's see an example of HTML anchor tag.

```
<a href="second.html">Click for Second Page</a>
```

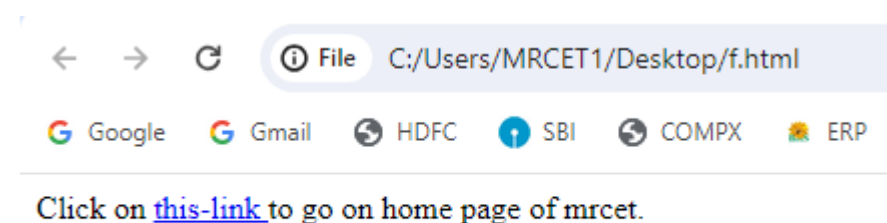
Specify a location for Link using target attribute

If we want to open that link to another page then we can use target attribute of <a> tag. With the help of this link will be open in next page.

Example:

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
<p>Click on <a href="https://www.mrcet.com/" target="_blank"> this-
link </a> to go on home page of mrcet.</p>
</body>
</html>
```

Output:



HTML Image

HTML img tag is used to display image on the web page. HTML img tag is an empty tag that contains attributes only, closing tags are not used in HTML image element.

Let's see an example of HTML image.

1. **<h2>**HTML Image Example**</h2>**
2. ****

Output:



Attributes of HTML img tag

The src and alt are important attributes of HTML img tag. All attributes of HTML image tag are given below.

1) *src*

It is a necessary attribute that describes the source or path of the image. It instructs the browser where to look for the image on the server.

The location of image may be on the same directory or another server.

2) *alt*

The alt attribute defines an alternate text for the image, if it can't be displayed. The value of the alt attribute describe the image in words. The alt attribute is considered good for SEO prospective.

3) *width*

It is an optional attribute which is used to specify the width to display the image. It is not recommended now. You should apply CSS in place of width attribute.

4) *height*

It h3 the height of the image. The HTML height attribute also supports iframe, image and object elements. It is not recommended now. You should apply CSS in place of height attribute.

Use of height and width attribute with img tag

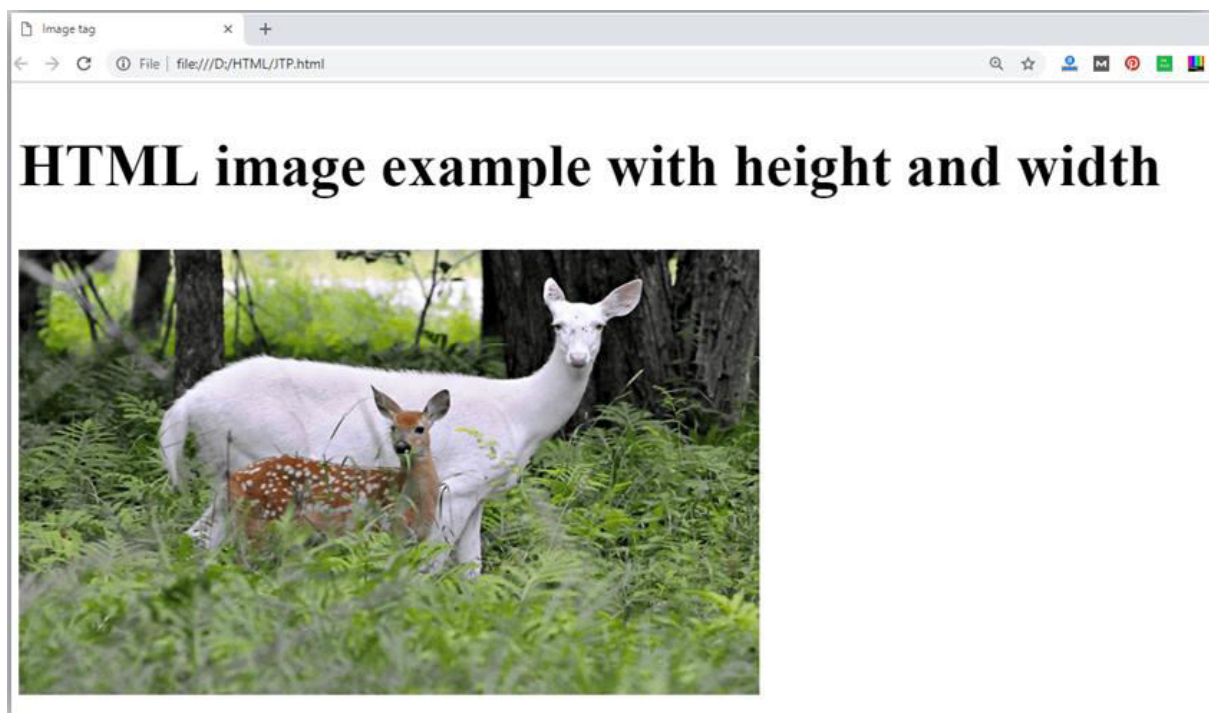
You have learnt about how to insert an image in your web page, now if we want to give some height and width to display image according to our requirement, then we can set it with height and width attributes of image.

Example:

```

```

Output:



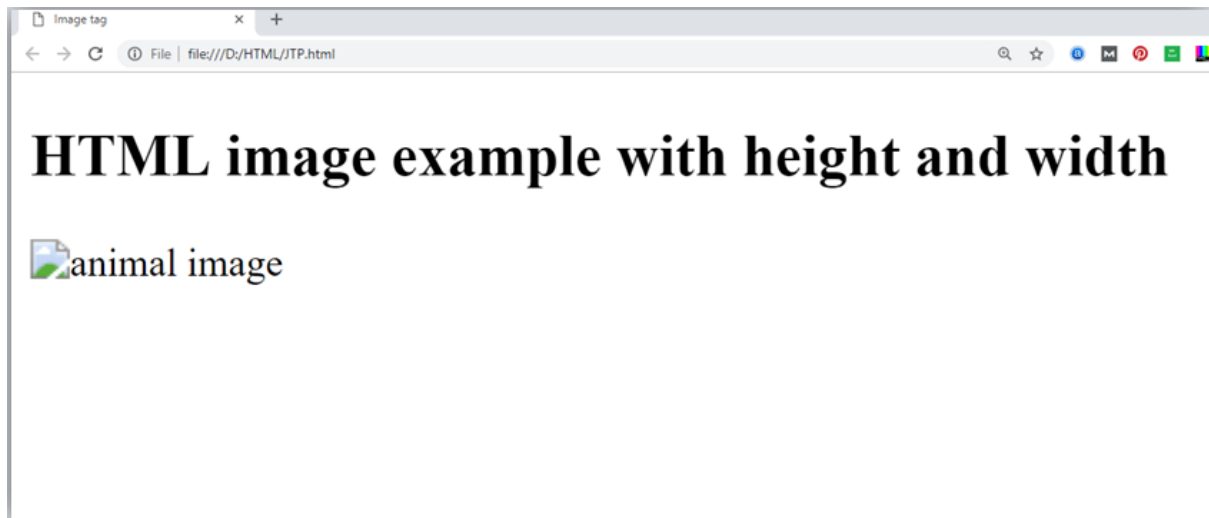
Note: Always try to insert the image with height and width, else it may flicker while displaying on webpage.

Use of alt attribute

We can use alt attribute with `` tag. It will display an alternative text in case if image cannot be displayed on browser. Following is the example for alt attribute:

1. ``

Output:



HTML Table

HTML table tag is used to display data in tabular form (row * column). There can be many columns in a row.

We can create a table to display data in tabular form, using `<table>` element, with the help of `<tr>`, `<td>`, and `<th>` elements.

In Each table, table row is defined by `<tr>` tag, table header is defined by `<th>`, and table data is defined by `<td>` tags.

HTML tables are used to manage the layout of the page e.g. header section, navigation bar, body content, footer section etc. But it is recommended to use `div` tag over table to manage the layout of the page .

```
<html>
<head>
  <title> </title>
</head>
<body>
<table border="5">
<tr><th>First_Name</th><th>Last_Name</th><th>Marks</th></tr>
<tr><td>Sonoo</td><td>Jaiswal</td><td>60</td></tr>
<tr><td>James</td><td>William</td><td>80</td></tr>
<tr><td>Swati</td><td>Sironi</td><td>82</td></tr>
<tr><td>Chetna</td><td>Singh</td><td>72</td></tr>
</table>
</body>
</html>
```

Output:

First_Name	Last_Name	Marks
Sonoo	Jaiswal	60
James	William	80
Swati	Sironi	82
Chetna	Singh	72

HTML Lists

HTML Lists are used to specify lists of information. All lists may contain one or more list elements. There are three different types of HTML lists:

1. Ordered List or Numbered List (ol)
2. Unordered List or Bulleted List (ul)

HTML Ordered List or Numbered List

In the ordered HTML lists, all the list items are marked with numbers by default. It is known as numbered list also. The ordered list starts with `` tag and the list items start with `` tag.

```
<!DOCTYPE>
<html>
<body>
  <ol>
    <li>Aries</li>
    <li>Bingo</li>
    <li>Leo</li>
    <li>Oracle</li>
  </ol>

</body>
</html>
```

Output:

1. Aries
2. Bingo
3. Leo
4. Oracle

HTML Unordered List or Bulleted List

In HTML Unordered list, all the list items are marked with bullets. It is also known as bulleted list also. The Unordered list starts with `` tag and list items start with the `` tag.

- disc
- circle
- square
- none

HTML Unordered List Example

1. ``
2. `HTML`
3. `Java`
4. `JavaScript`
5. `SQL`
6. ``

Test it Now

Output:

- HTML
- Java
- JavaScript
- SQL

ul type="circle"

1. `<ul type="circle">`
2. `HTML`
3. `Java`
4. `JavaScript`

5. `SQL`

6. ``

Test it Now

Output:

- HTML
- Java
- JavaScript
- SQL

ul type="square"

```
<ul type="square">  
<li>HTML</li>  
<li>Java</li>  
<li>JavaScript</li>  
<li>SQL</li>  
</ul>
```

Output:

- HTML
- Java
- JavaScript
- SQL

HTML Form

An **HTML form** is a section of a document which contains controls such as text fields, password fields, checkboxes, radio buttons, submit button, menus etc.

An HTML form facilitates the user to enter data that is to be sent to the server for processing such as name, email address, password, phone number, etc. .

Why use HTML Form

HTML forms are required if you want to collect some data from of the site visitor.

For example: If a user want to purchase some items on internet, he/she must fill the form such as shipping address and credit/debit card details so that item can be sent to the given address.

HTML Form Syntax

```
<form action="server url" method="get|post">  
  //input controls e.g. textfield, textarea, radiobutton, button  
</form>
```

HTML TextField Control

The type="text" attribute of input tag creates textfield control also known as single line textfield control. The name attribute is optional, but it is required for the server side component such as JSP, ASP, PHP etc.

```
<form>  
  First Name: <input type="text" name="firstname"/> <br/>  
  Last Name: <input type="text" name="lastname"/> <br/>  
</form>
```

HTML <textarea> tag in form

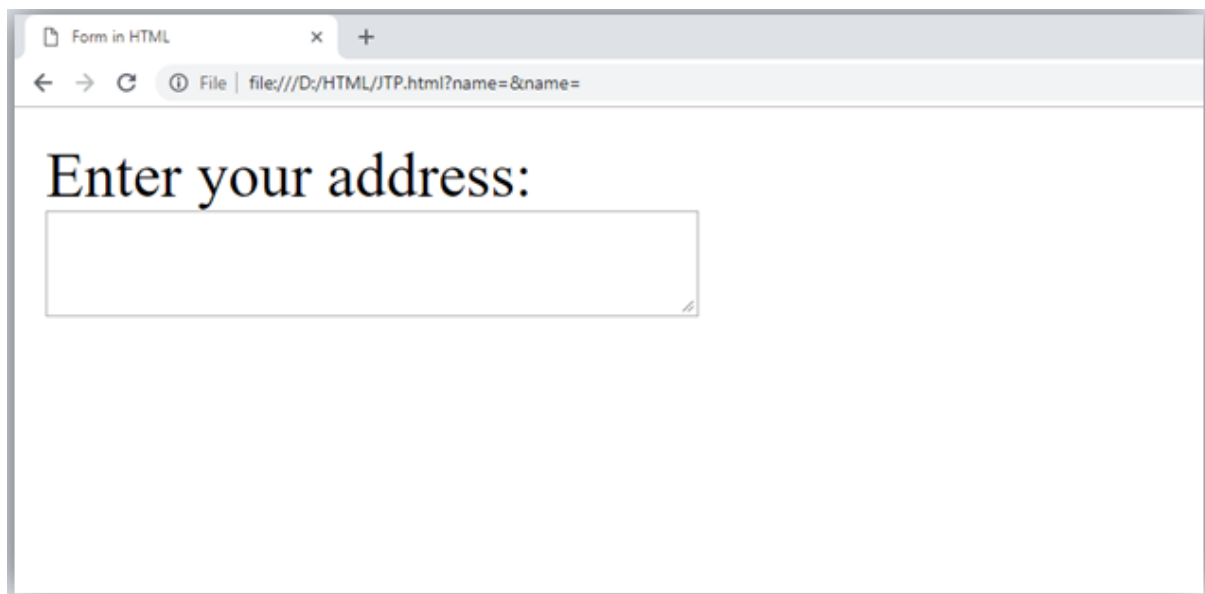
The <textarea> tag in HTML is used to insert multiple-line text in a form. The size of <textarea> can be specify either using "rows" or "cols" attribute or by CSS.

Example:

```
<!DOCTYPE html>  
<html>  
<head>  
  <title>Form in HTML</title>  
</head>
```



```
<body>
  <form>
    Enter your address:<br>
    <textarea rows="2" cols="20"> </textarea>
  </form>
</body>
</html>
```

Output:

Label Tag in Form

It is considered better to have label in form. As it makes the code parser/browser/user friendly.

If you click on the label tag, it will focus on the text control. To do so, you need to have for attribute in label tag that must be same as id attribute of input tag.

NOTE: It is good to use <label> tag with form, although it is optional but if you will use it, then it will provide a focus when you tap or click on label tag. It is more worthy with touchscreens.

```
<form>
  <label for="firstname">First Name: </label> <br/>
  <input type="text" id="firstname" name="firstname"/> <br/>
  <label for="lastname">Last Name: </label>
  <input type="text" id="lastname" name="lastname"/> <br/>
```

```
</form>
```

Output:

First Name:

Last Name:

HTML Password Field Control

The password is not visible to the user in password field control.

1.

```
<form>
```
2.

```
<label for="password">Password: </label>
```
3.

```
<input type="password" id="password" name="password"/> <br/>
```
4.

```
</form>
```

Output:

Password:

HTML 5 Email Field Control

The email field is new in HTML 5. It validates the text for correct email address. You must use @ and . in this field.

```
<form>
  <label for="email">Email: </label>
  <input type="email" id="email" name="email"/> <br/>
</form>
```

It will display in browser like below:

Email:

Radio Button Control

ADVERTISEMENT

The radio button is used to select one option from multiple options. It is used for selection of gender, quiz questions etc.

If you use one name for all the radio buttons, only one radio button can be selected at a time.

Using radio buttons for multiple options, you can only choose a single option at a time.

```
<form>
  <label for="gender">Gender: </label>
  <input type="radio" id="gender" name="gender" value="male"/>Male
  <input type="radio" id="gender" name="gender" value="female"/>Female
  <br/>
</form>
```

Gender: ☐ Male ☒ Female

Checkbox Control

The checkbox control is used to check multiple options from given checkboxes.

```
<form>
```

```
Hobby:<br>
```

```
  <input type="checkbox" id="cricket" name="cricket" value="cricket"/>
```

```
  <label for="cricket">Cricket</label> <br>
```

```
  <input type="checkbox" id="football" name="football" value="football"/>
```

```
  <label for="football">Football</label> <br>
```

```
  <input type="checkbox" id="hockey" name="hockey" value="hockey"/>
```

```
  <label for="hockey">Hockey</label>
```

```
</form>
```

Note: These are similar to radio button except it can choose multiple options at a time and radio button can select one button at a time, and its display.

Output:

Hobby:

- ☒ Cricket
- ☒ Football
- ☐ Hockey

Submit button control

HTML **<input type="submit">** are used to add a submit button on web page. When user clicks on submit button, then form get submit to the server.

Syntax:

1. **<input type="submit" value="submit">**

The type = submit , specifying that it is a submit button

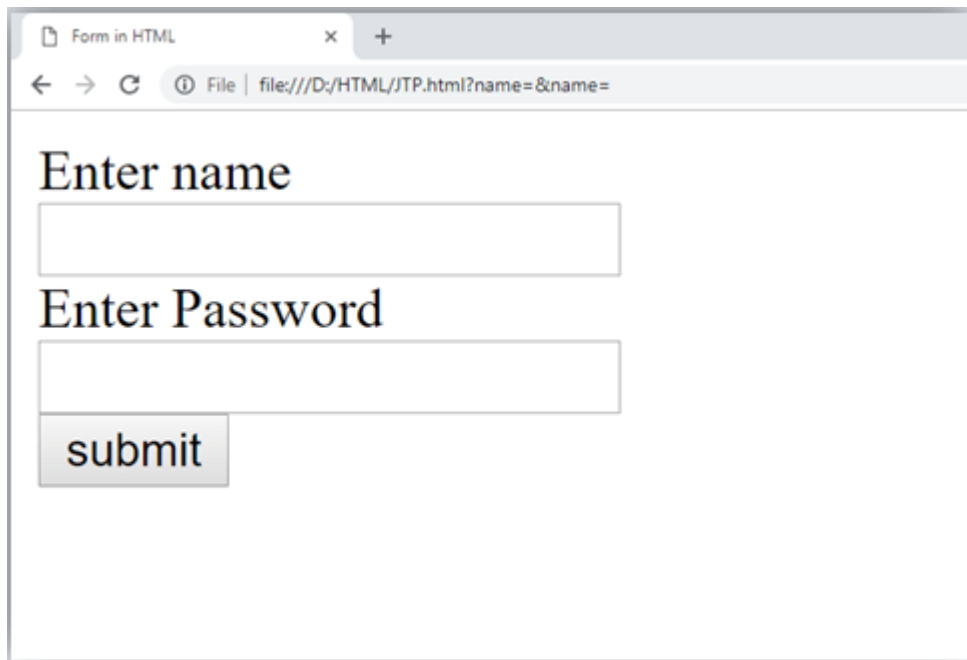
The value attribute can be anything which we write on button on web page.

The name attribute can be omit here.

Example:

```
<form>
  <label for="name">Enter name</label><br>
  <input type="text" id="name" name="name"> <br>
  <label for="pass">Enter Password</label><br>
  <input type="Password" id="pass" name="pass"> <br>
  <input type="submit" value="submit">
</form>
```

Output:



HTML <fieldset> element:

The <fieldset> element in HTML is used to group the related information of a form. This element is used with <legend> element which provide caption for the grouped elements.

Example:

```
<form>
  <fieldset>
    <legend>User Information:</legend>
    <label for="name">Enter name</label> <br>
    <input type="text" id="name" name="name"> <br>
    <label for="pass">Enter Password</label> <br>
    <input type="Password" id="pass" name="pass"> <br>
    <input type="submit" value="submit">
  </fieldset>
</form>
```

Output:



User Information:

Enter name

Enter Password

submit

HTML Form Example

Following is the example for a simple form of registration.

```
<!DOCTYPE html>
<html>
<head>
<title>Form in HTML</title>
</head>
<body>
<h2>Registration form</h2>
<form>
<fieldset>
<legend>User personal information</legend>
<label>Enter your full name</label><br>
<input type="text" name="name"><br>
<label>Enter your email</label><br>
<input type="email" name="email"><br>
<label>Enter your password</label><br>
<input type="password" name="pass"><br>
<label>confirm your password</label><br>
<input type="password" name="pass"><br>
<br><label>Enter your gender</label><br>
<input type="radio" id="gender" name="gender" value="male"/>Male <br>
<input type="radio" id="gender" name="gender" value="female"/>Female
<br/>
<input type="radio" id="gender" name="gender" value="others"/>others <br/>

<br>Enter your Address:<br>
<textarea></textarea><br>
```

```

        <input type="submit" value="sign-up">
    </fieldset>
</form>
</body>
</html>

```

Output:

HTML Form Example

Let's see a simple example of creating HTML form.

```

<form action="#">
  <table>
  <tr>
    <td class="tdLabel"> <label for="register_name" class="label">Enter name:</label>
  </td>
    <td><input type="text" name="name" value="" id="register_name" style="width:160px"/></td>
  </tr>
  <tr>
    <td class="tdLabel"> <label for="register_password" class="label">Enter password:</label>
  </td>
    <td><input type="password" name="password" id="register_password" style="width:160px"/></td>
  </tr>
  </table>
  <input type="submit" value="sign-up">
</form>

```



```

="width:160px"/></td>
</tr>
<tr>
  <td class="tdLabel"><label for="register_email" class="label">Enter Email:</label><
/td>
  <td
><input type="email" name="email" value="" id="register_email" style="width:160px"/
></td>
</tr>
<tr>
  <td class="tdLabel"><label for="register_gender" class="label">Enter Gender:
</label></td>
  <td>
<input type="radio" name="gender" id="register_gendermale" value="male"/>
<label for="register_gendermale">male</label>
<input type="radio" name="gender" id="register_genderfemale" value="female"
/>
<label for="register_genderfemale">female</label>
</td>
</tr>
<tr>
  <td class="tdLabel"><label for="register_country" class="label">Select Country:</la
bel></td>
  <td><select name="country" id="register_country" style="width:160px">
<option value="india">india</option>
<option value="pakistan">pakistan</option>
<option value="africa">africa</option>
<option value="china">china</option>
<option value="other">other</option>
</select>
</td>
</tr>
<tr>
  <td colspan="2"><div align="right"><input type="submit" id="register_0" va
lue="register"/>
</div></td>
</tr>

```

```
</table>  
</form>
```

HTML style using CSS

Our web page with CSS (Cascading Stylesheet) properties.

CSS is used to apply the style in the web page which is made up of HTML elements. It describes the look of the webpage.

CSS provides various style properties such as background color, padding, margin, border-color, and many more, to style a webpage.

Each property in CSS has a name-value pair, and each property is separated by a semicolon (;).

Three ways to apply CSS

To use CSS with HTML document, there are three ways:

- **Inline CSS:** Define CSS properties using style attribute in the HTML elements.
- **Internal or Embedded CSS:** Define CSS using <style> tag in <head> section.
- **External CSS:** Define all CSS property in a separate .css file, and then include the file with HTML file using tag in section.

Inline CSS:

Inline CSS is used to apply CSS in a single element. It can apply style uniquely in each element.

To apply inline CSS, you need to use style attribute within HTML element. We can use as many properties as we want, but each property should be separated by a semicolon (;).

Example:

```
<html>  
<head>  
  <title></title>  
</head>  
<body>  
  <h3 style="color: red;
```

```
font-style: italic;
text-align: center;
font-size: 50px;
padding-top: 25px;">Learning HTML using Inline CSS</h3>
</body>
</html>
```

Output:

Learning HTML using Inline CSS

Internal CSS:

An Internal stylesheets contains the CSS properties for a webpage in <head> section of HTML document. To use Internal CSS, we can use class and id attributes.

We can use internal CSS to apply a style for a single HTML page.

Example:

```
<!DOCTYPE html>
<html>
<head>
  <style>
    /*Internal CSS using element name*/
    body{background-color:lavender;
      text-align: center;}
    h2{font-style: italic;
      font-size: 30px;
      color: #f08080;}
    p{font-size: 20px;}
    /*Internal CSS using class name*/
    .blue{color: blue;}
    .red{color: red;}
    .green{color: green;}
  </style>
</head>
<body>
```

```
<h2>Learning HTML with internal CSS</h2>
<p class="blue">This is a blue color paragraph</p>
<p class="red">This is a red color paragraph</p>
<p class="green">This is a green color paragraph</p>
</body>
</html>
```

OUTPUT:

Learning HTML with internal CSS

This is a blue color paragraph

This is a red color paragraph

This is a green color paragraph

External CSS:

An external CSS contains a separate CSS file which only contains style code using the class name, id name, tag name, etc. We can use this CSS file in any HTML file by including it in HTML file using <link> tag.

If we have multiple HTML pages for an application and which use similar CSS, then we can use external CSS.

There are two files need to create to apply external CSS

- First, create the HTML file
- Create a CSS file and save it using the .css extension (This file only will only contain the styling code.)
- Link the CSS file in your HTML file using tag in header section of HTML document.

Example:

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

```
<h2>Learning HTML with External CSS</h2>
<p class="blue">This is a blue color paragraph</p>
<p class="red">This is a red color paragraph</p>
<p class="green">This is a green color paragraph</p>
</body>
</html>
```

CSS file:

```
body{
background-color:lavender;
text-align: center;
}
h2{
font-style: italic;
size: 30px;
color: #f08080;
}
p{
font-size: 20px;
}

.blue{
color: blue;
}
.red{
color: red;
}
.green{
color: green;
}
```

WEB SERVER

A web server is a computer that stores web server software and a website's component files (for example, HTML documents, images, CSS stylesheets, and JavaScript files). A web server connects to the Internet and supports physical data interchange with other devices connected to the web.

A web server includes several parts that control how web users access hosted files. At a minimum, this is an HTTP server. An HTTP server is software that understands URLs (web addresses) and HTTP (the protocol your browser uses to view webpages). An HTTP server can be accessed through the domain names of the websites it stores, and it delivers the

content of these hosted websites to the end user's device.

At the most basic level, whenever a browser needs a file that is hosted on a web server, the browser requests the file via HTTP. When the request reaches the correct (hardware) web server, the (software) HTTP server accepts the request, finds the requested document, and sends it back to the browser, also through HTTP. (If the server doesn't find the requested document, it returns a 404 response instead.)

Basic representation of a client/server connection through HTTP

To publish a website, you need either a static or a dynamic web server.

A static web server, or stack, consists of a computer (hardware) with an HTTP server (software). We call it "static" because the server sends its hosted files as-is to your browser.

A dynamic web server consists of a static web server plus extra software, most commonly an application server and a database. We call it "dynamic" because the application server updates the hosted files before sending content to your browser via the HTTP server.

For example, to produce the final webpages you see in the browser, the application server might fill an HTML template with content from a database. Sites like MDN or Wikipedia have thousands of webpages. Typically, these kinds of sites are composed of only a few HTML templates and a giant database, rather than thousands of static HTML documents. This setup makes it easier to maintain and deliver the content.

Git & Github

What is Git?

Git is a popular version control system. It was created by Linus Torvalds in 2005, and has been maintained by Junio Hamano since then.

It is used for:

- Tracking code changes
- Tracking who made changes
- Coding collaboration

What does Git do?

- Manage projects with **Repositories**
- **Clone** a project to work on a local copy
- Control and track changes with **Staging** and **Committing**
- **Branch** and **Merge** to allow for work on different parts and versions of a project

- **Pull** the latest version of the project to a local copy
- **Push** local updates to the main project

Working with Git

- Initialize Git on a folder, making it a **Repository**
- Git now creates a hidden folder to keep track of changes in that folder
- When a file is changed, added or deleted, it is considered **modified**
- You select the modified files you want to **Stage**
- The **Staged** files are **Committed**, which prompts Git to store a **permanent** snapshot of the files
- Git allows you to see the full history of every commit.
- You can revert back to any previous commit.
- Git does not store a separate copy of every file in every commit, but keeps track of changes made in each commit!

Github

GitHub is a code hosting platform for version control and collaboration. It lets you and others work together on projects from anywhere.

This tutorial teaches you GitHub essentials like repositories, branches, commits, and pull requests. You'll create your own Hello World repository and learn GitHub's pull request workflow, a popular way to create and review code.

In this quickstart guide, you will:

- Create and use a repository
- Start and manage a new branch
- Make changes to a file and push them to GitHub as commits
- Open and merge a pull request

To complete this tutorial, you need a GitHub account and Internet access. You don't need to know how to code, use the command line, or install Git (the version control software that GitHub is built on). If you have a question about any of the expressions used in this guide, head on over to the glossary to find out more about our terminology.

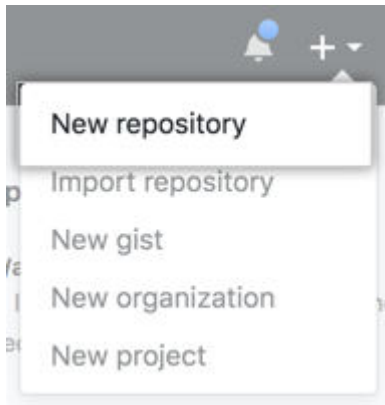
Creating a repository

A repository is usually used to organize a single project. Repositories can contain folders and files, images, videos, spreadsheets, and data sets -- anything your project needs. Often, repositories include a README file, a file with information about your project. README files are written in the plain text Markdown language. You can use this cheat sheet to get started with Markdown syntax. GitHub lets you add a README file at the same time you create your new repository. GitHub also offers other common

options such as a license file, but you do not have to select any of them now.

Your hello-world repository can be a place where you store ideas, resources, or even share and discuss things with others.

1. In the upper-right corner of any page, use the drop-down menu, and select



2. New repository.
3. In the Repository name box, enter hello-world.
4. In the Description box, write a short description.
5. Select Add a README file.
6. Select whether your repository will be Public or Private.
7. Click Create repository.

UNIT - II

Frontend Development: Javascript basics OOPS Aspects of JavaScript Memory usage and Functions in JS AJAX for data exchange with server jQuery Framework jQuery events, UI components etc. JSON data format.

HTML JavaScript

A Script is a small program which is used with HTML to make web pages more attractive, dynamic and interactive, such as an alert popup window on mouse click. Currently, the most popular scripting language is JavaScript used for websites.

Example:

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Date and Time example</h1>
<button type="button"
onclick="document.getElementById('demo').innerHTML = Date()">
Click me to display Date and Time.</button>
<p id="demo"></p>
</body>
</html>
```

HTML events with JavaScript

An event is something which user does, or browser does such as mouse click or page loading are examples of events, and JavaScript comes in the role if we want something to happen on these events.

HTML provides event handler attributes which work with JavaScript code and can perform some action on an event.

Syntax:

<element event = "JS code">

Example:

```
<!DOCTYPE html>
<html>
<body>
    <h2>Click Event Example</h2>
    <p>Click on the button and you can see a pop-up window with a
message</p>
    <input type="button" value="Click" onclick="alert('Hi,how are you')">
</body>
</html>
```

HTML can have following events such as:

- **Form events:** reset, submit, etc.
- **Select events:** text field, text area, etc.
- **Focus event:** focus, blur, etc.
- **Mouse events:** select, mouseup, mousemove, mousedown, click, dblclick, etc.

Following are the list for Window event attributes:

Event	Event Name	Handler Name	Occurs when
	onBlur	blur	When form input loses focus
	onClick	click	When the user clicks on a form element or a link
	onSubmit	submit	When user submits a form to the server.
	onLoad	load	When page loads in a browser.
	onFocus	focus	When user focuses on an input field.
	onSelect	select	When user selects the form input filed.

Note: You will learn more about JavaScript Events in our JavaScript tutorial.

Let's see what JavaScript can do:

1) JavaScript can change HTML content.

Example:

```
<!DOCTYPE html>
<html>
<body>
<p>JavaScript can change the content of an HTML element:</p>
<button type="button" onclick="myFunction()">Click Me!</button>
<p id="demo"></p>
<script>
function myFunction() {
document.getElementById("demo").innerHTML = "Hello MRCET!";
}
</script>
</body>
</html>
```

1) JavaScript can change HTML content.

Example:

```
<!DOCTYPE html>
<html>
<body>
<p>JavaScript can change the content of an HTML element:</p>
<button type="button" onclick="myFunction()">Click Me!</button>
<p id="demo"></p>
<script>
function myFunction() {
document.getElementById("demo").innerHTML = "Hello MRCET!";
}
</script>
</body>
</html>
```

2) JavaScript can change HTML style

Example:

```
<!DOCTYPE html>
<html>
<body>
<p id="demo">JavaScript can change the style of an HTML element.</p>
```

```
<script>
function myFunction() {
  document.getElementById("demo").style.fontSize = "25px";
  document.getElementById("demo").style.color = "brown";
  document.getElementById("demo").style.backgroundColor = "lightgreen";
}
</script>
<button type="button" onclick="myFunction()">Click Me!</button>
</body>
</html>
```

3) JavaScript can change HTML attributes.

Example:

```
<!DOCTYPE html>
<html>
<body>
<script>
function light(sw) {
  var pic;
  if (sw == 0) {
    pic = "pic_lightoff.png"
  } else {
    pic = "pic_lighton.png"
  }
  document.getElementById('myImage').src = pic;
}
</script>

<p>
<button type="button" onclick="light(1)">Light On</button>
<button type="button" onclick="light(0)">Light Off</button>
</p>
</body>
</html>
```

Use of External javascript

Suppose, you have various HTML files which should have same script, then we can put our JavaScript code in separate file and can call in HTML file. Save JavaScript external files using .js extension.

Note: Do not add <script> tag in the external file, and provide the complete path where you have put the JS file.

Syntax:

```
<script type="text/javascript" src="URL "></script>
```

Example:

```
<!DOCTYPE html>
<html>
<head>
<script type="text/javascript" src="external.js"></script>
```

```
</head>
<body>
  <h2>External JavaScript Example</h2>
  <form onsubmit="fun()">
    <label>Enter your name:</label><br>
    <input type="text" name="uname" id="frm1"><br>
    <label>Enter your Email-address:</label><br>
    <input type="email" name="email"><br>
    <input type="submit">
  </form>
</body>
</html>
```

JavaScript code:external.js

```
function fun() {
  var x = document.getElementById("frm1").value;
  alert("Hi"+" "+x+" you have successfully submitted the details");
}
```

JavaScript Objects

A JavaScript object is an entity having state and behavior (properties and method). For example: car, pen, bike, chair, glass, keyboard, monitor etc.

JavaScript is an object-based language. Everything is an object in JavaScript.

JavaScript is template based not class based. Here, we don't create class to get the object. But, we directly create objects.

Creating Objects in JavaScript

There are 3 ways to create objects.

1. By object literal
2. By creating instance of Object directly (using new keyword)
3. By using an object constructor (using new keyword)

1) JavaScript Object by object literal

The syntax of creating object using object literal is given below:

object={property1:value1,property2:value2.....propertyN:valueN}

As you can see, property and value is separated by : (colon).

Let's see the simple example of creating object in JavaScript.

```
<html>
<body>
<script>
emp={id:102,name:"Shyam Kumar",salary:40000}
document.write(emp.id+" "+emp.name+" "+emp.salary);
</script>
</body>
</html>
```

Output

102 Shyam Kumar 40000

2) By creating instance of Object

The syntax of creating object directly is given below:

```
var objectname=new Object();
```

Here, **new keyword** is used to create object.

Let's see the example of creating object directly.

```
<html>
<body>
<script>
var emp=new Object();
emp.id=101;
emp.name="Ravi Malik";
emp.salary=50000;
document.write(emp.id+" "+emp.name+" "+emp.salary);
</script>
</body>
</html>
```

Out put:

101 Ravi Malik 50000

3) By using an Object constructor

Here, you need to create function with arguments. Each argument value can be assigned in the current object by using this keyword.

The **this keyword** refers to the current object.

The example of creating object by object constructor is given below.

```
<html>
<body>
<script>
function emp(id,name,salary){
this.id=id;
this.name=name;
this.salary=salary;
}
e=new emp(103,"Vimal Jaiswal",30000);
document.write(e.id+" "+e.name+" "+e.salary);
</script>
</body>
</html>
```

Out put:

103 Vimal Jaiswal 30000

Class Declarations

A class can be defined by using a class declaration. A class keyword is used to declare a class with any particular name. According to JavaScript naming conventions, the name of the class always starts with an uppercase letter.

Class Declarations Example

Let's see a simple example of declaring the class.

```
<!DOCTYPE html>
<html>
<body>
<script>
//Declaring class
class Employee
{
//Initializing an object
constructor(id,name)
{
    this.id=id;
    this.name=name;
}
//Declaring method
detail()
{
    document.writeln(this.id+" "+this.name+"<br>")
}
}
//passing object to a variable
var e1=new Employee(101,"Martin Roy");
var e2=new Employee(102,"Duke William");
e1.detail(); //calling method
e2.detail();
</script>
</body>
</html>
```

OUTPUT:

```
101 Martin Roy
102 Duke William
```

JQuery

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

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

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

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

- jQuery simplifies AJAX call and DOM manipulation.

It is always advised to a fresher to learn the basics of web designing before starting to learn jQuery. We should learn HTML, CSS and JavaScript first.

jQuery Example

File: firstjquery.html

```
<!DOCTYPE html>
<html>
<head>
  <title>First jQuery Example</title>
  <script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
  </script>
  <script type="text/javascript" language="javascript">
    $(document).ready(function() {
      $("p").css("background-color", "pink");
    });
  </script>
</head>
<body>
  <p>This is first paragraph.</p>
  <p>This is second paragraph.</p>
  <p>This is third paragraph.</p>
</body>
</html>
```

jQuery is a [JavaScript framework](#). It facilitates the readability and the manipulation of HTML DOM elements, event handling, animations, and AJAX calls. It's also free, [open-source software](#) that adheres to the MIT License. As a result, it is one of the most popular JavaScript libraries.

jQuery Events

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

These are some examples of events.

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

jQuery click() event

When you click on an element, the click event occurs and once the click event occurs it execute the click () method or attaches a function to run.

It is generally used together with other events of jQuery.

Syntax:

\$(selector).click()

example to demonstrate jQuery click() event.

```
<!DOCTYPE html>
<html>
<head>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js">
</script>
<script>
$(document).ready(function(){
    $("p").click(function(){
        alert("This paragraph was clicked.");
    });
});
</script>
</head>
<body>
<p>Click on the statement.</p>
</body>
</html>
```

example to demonstrate the jQuery click() event. In this example, when you click on the heading element, it will hide the current heading.

```
<!DOCTYPE html>
<html>
<head>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>

<script>
$(document).ready(function(){
    $("h1,h2,h3").click(function(){
        $(this).hide();
    });
});
</script>
</head>
<body>
<h1>This heading will disappear if you click on this.</h1>
<h2>I will also disappear.</h2>
<h3>Me too.</h3>
</body>
</html>
```

jQuery focus()

The jQuery focus event occurs when an element gains focus. It is generated by a mouse click or by navigating to it.

This event is implicitly used to limited sets of elements such as form elements like <input>, <select> etc. and links <a href>. The focused elements are usually highlighted in some way by the browsers.

The focus method is often used together with blur () method.

Syntax:

`$(selector).focus()`

It triggers the focus event for selected elements.

`$(selector).focus(function)`

It adds a function to the focus event.

Example of jQuery focus() event

Let's take an example to demonstrate jQuery focus() event.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>focus demo</title>
  <style>
    span {
      display: none;
    }
  </style>
  <script src="https://code.jquery.com/jquery-1.10.2.js"></script>
</head>
<body>
  <p><input type="text"> <span>Focus starts.. Write your name.</span></p>
  <p><input type="password"> <span>Focus starts.. Write your password.</span></p>
  <script>
    $( "input" ).focus(function() {
      $( this ).next( "span" ).css( "display", "inline" ).fadeOut( 2000 );
    });
  </script>
</body>
</html>
```

jQuery submit()

jQuery submit event is sent to the element when the user attempts to submit a form. This event is only attached to the `<form>` element. Forms can be submitted either by clicking on the submit button or by pressing the enter button on the keyboard when that certain form elements have focus. When the submit event occurs, the submit() method attaches a function with it to run.

Syntax:

`$(selector).submit()`

Example of jQuery submit() event

Let's take an example to demonstrate jQuery submit() event.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>submit demo</title>
  <style>
    p {
      margin: 0;
      color: blue;
    }
    div,p {
      margin-left: 10px;
    }
    span {
      color: red;
    }
  </style>
  <script src="https://code.jquery.com/jquery-1.10.2.js"> </script>
</head>
<body>
<p>Type 'MRCET' to submit this form finally.</p>
<form action="javascript:alert( 'success!' );">
  <div>
    <input type="text">
    <input type="submit">
  </div>
</form>
<span></span>
<script>
$( "form" ).submit(function( event ) {
  if ( $( "input:first" ).val() === "MRCET" ) {
    $( "span" ).text( "Submitted Successfully." ).show();
    return;
  }
  $( "span" ).text( "Not valid!" ).show().fadeOut( 2000 );
  event.preventDefault();
});
</script>
</body>
</html>
```

jQuerymouseover()

The mouseover event is occurred when you put your mouse cursor over the selected element .Once the mouseover event is occurred, it executes the mouseover () method or attach a function to run.

This event is generally used with mouseout() event.

Note: Most of the people are confused between mouseenter and mouseover.

Difference between mouseenter() and mouseover()

The mouseenter event is only triggered if the mouse pointer enters the selected element whereas the mouseover event triggers if the mouse cursor enters any child elements as well as the selected element.

Syntax:

`$(selector).mouseover()`

It triggers the mouseover event for selected elements.

`$(selector).mouseover(function)`

Example of jQuery mouseover() event

Let's take an example to demonstrate jQuery mouseover() event.

```
<!DOCTYPE html>
<html>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
<script>
$(document).ready(function(){
  $("p").mouseover(function(){
    $("p").css("background-color", "lightgreen");
  });
  $("p").mouseout(function(){
    $("p").css("background-color", "orange");
  });
});
</script>
</head>
<body>
<p>Move your cursor over this paragraph.</p>
</body>
</html>
```

jQuery mouseover() event example 2

Let's see another example of jQuery mouseover() event.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>mouseover demo</title>
<style>
div.out {
  width: 40%;
  height: 120px;
```

```
margin: 0 15px;
background-color: lightgreen;
}
div.in {
width: 60%;
height: 60%;
background-color: red;
margin: 10px auto;
}
</style>
<script src="https://code.jquery.com/jquery-1.10.2.js"></script>
</head>
<body>
<div class="out">
<span style="padding:20px">move your mouse</span>
<div class="in"></div>
</div>
<script>
$( "div.out" )
.mouseover(function() {
$( this ).find( "span" ).text( "mouse over " );
})
.mouseout(function() {
$( this ).find( "span" ).text( "mouse out " );
});
</script>
</body>
</html>
```

jQuery UI Categorization

We can categorize the jQuery UI into four groups.

1. Interactions
2. Widgets
3. Effects
4. Utilities

1) Interactions: Interactions are the set of plug-ins which facilitates users to interact with DOM elements. These are the mostly used interactions:

- Draggable
- Droppable
- Resizable
- Selectable
- Sortable

2) Widgets: Widgets are the jQuery plug-ins which makes you able to create user interface elements like date picker, progress bar etc. These are the mostly used widgets:

- Accordion
- Autocomplete
- Dialog
- Button
- Date Picker
- Menu
- Progress Bar
- Tabs
- Tooltip
- Slider
- Spinner

3) Effects: The internal jQuery effects contain a full suite of custom animation and transition for DOM elements.

- Hide
- Show
- Add Class
- Remove Class
- Switch Class
- Toggle Class
- Color Animation
- Effect
- Toggle

4) Utilities: Utilities are the modular tools, used by jQuery library internally.

- **Position:** It is used to set the position of the element according to the other element's alignment (position).

Ajax programming

AJAX stands for Asynchronous JavaScript and XML. It describes a concept of asynchronous data transfer (here: data encapsulated in XML) between the client (usually a web browser) and a server to only exchange/ alter a part of the webpage without the need of a full pagereload. That means the browser will issue an XML Http Request in the background and receive only a part of the page - usually tied to one or more html-tags holding uids.

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <p id="demo"> </p>
```

```
<hi> ajax demo</hi>
<button onclick="fun1()">click here</button>
<script>
  function fun1()
  {
    var xhr=new XMLHttpRequest();
    xhr.onload=function()
    {
      document.getElementById("demo").innerHTML=xhr.responseText;
    }
    xhr.open("GET","ajaxinfo.txt",true);
    xhr.send();
  }
</script>
</body>
</html>
```

Output:
ajaxinfo.txt
hello ajax wants to interact with you

JSON Example

JSON example can be created by object and array. Each object can have different data such as text, number, boolean etc. Let's see different JSON examples using object and array.

JSON Object Example

A JSON object contains data in the form of key/value pair. The keys are strings and the values are the JSON types. Keys and values are separated by colon. Each entry (key/value pair) is separated by comma.

The { (curly brace) represents the JSON object.

```
{
  "employee": {
    "name":    "sonoo",
    "salary": 56000,
    "married": true
  }
}
```

JavaScript Memory usage

JavaScript automatically allocates memory when objects are created and frees it when they are not used anymore (garbage collection). This automaticity is a potential source of confusion: it can give developers the false impression that they don't need to worry about memory management.

Memory life cycle

Regardless of the programming language, the memory life cycle is pretty much always the same:

1. Allocate the memory you need
2. Use the allocated memory (read, write)
3. Release the allocated memory when it is not needed anymore

The second part is explicit in all languages. The first and last parts are explicit in low-level languages but are mostly implicit in high-level languages like JavaScript.

Allocation in JavaScript

Value initialization

In order to not bother the programmer with allocations, JavaScript will automatically allocate memory when values are initially declared.

```
const n = 123; // allocates memory for a number
const s = "azerty"; // allocates memory for a string

const o = {
  a: 1,
  b: null,
}; // allocates memory for an object and contained values

// (like object) allocates memory for the array and
// contained values
const a = [1, null, "abra"];

function f(a) {
  return a + 2;
} // allocates a function (which is a callable object)

// function expressions also allocate an object
someElement.addEventListener(
  "click",
  () => {
    someElement.style.backgroundColor = "blue";
  },
  false,
);
```

Allocation via function calls

Some function calls result in object allocation.

```
const d = new Date(); // allocates a Date object
```

```
const e = document.createElement("div"); // allocates a DOM element  
Copy to Clipboard
```

Some methods allocate new values or objects:

```
const s = "azerty";  
const s2 = s.substr(0, 3); // s2 is a new string  
// Since strings are immutable values,  
// JavaScript may decide to not allocate memory,  
// but just store the [0, 3] range.
```

```
const a = ["ouais ouais", "nan nan"];  
const a2 = ["generation", "nan nan"];  
const a3 = a.concat(a2);  
// new array with 4 elements being  
// the concatenation of a and a2 elements.
```

Using values

Using values basically means reading and writing in allocated memory. This can be done by reading or writing the value of a variable or an object property or even passing an argument to a function.

Release when the memory is not needed anymore

The majority of memory management issues occur at this phase. The most difficult aspect of this stage is determining when the allocated memory is no longer needed.

Low-level languages require the developer to manually determine at which point in the program the allocated memory is no longer needed and to release it.

UNIT-3

REACT JS: Introduction to React React Router and Single Page Applications React Forms, Flow Architecture and Introduction to Redux More Redux and Client-Server Communication

React Introduction

ReactJS is a declarative, efficient, and flexible JavaScript library for building reusable UI components. It is an open-source, component-based front end library responsible only for the view layer of the application. It was created by **Jordan Walke**, who was a software engineer at **Facebook**. It was initially developed and maintained by Facebook and was later used in its products like **WhatsApp** & **Instagram**. Facebook developed ReactJS in **2011** in its newsfeed section, but it was released to the public in the month of **May 2013**.

Today, most of the websites are built using MVC (model view controller) architecture. In MVC architecture, React is the 'V' which stands for view, whereas the architecture is provided by the Redux or Flux.

A ReactJS application is made up of multiple components, each component responsible for outputting a small, reusable piece of HTML code. The components are the heart of all React applications. These Components can be nested with other components to allow complex applications to be built of simple building blocks. ReactJS uses virtual DOM based mechanism to fill data in HTML DOM. The virtual DOM works fast as it only changes individual DOM elements instead of reloading complete DOM every time.

To create React app, we write React components that correspond to various elements. We organize these components inside higher level components which define the application structure. For example, we take a form that consists of many elements like input fields, labels, or buttons. We can write each element of the form as React components, and then we combine it into a higher-level component, i.e., the form component itself. The form components would specify the structure of the form along with elements inside of it.

React create-react-app

Starting a new React project is very complicated, with so many build tools. It uses many dependencies, configuration files, and other requirements such as Babel, Webpack, ESLint before writing a single line of React code. Create React App CLI tool removes all that complexities and makes React app simple. For this, you need to install the package using NPM, and then run a few simple commands to get a new React project.

The **create-react-app** is an excellent tool for beginners, which allows you to create and run React project very quickly. It does not take any configuration manually. This tool is wrapping all of the required dependencies like **Webpack**, **Babel** for React

project itself and then you need to focus on writing React code only. This tool sets up the development environment, provides an excellent developer experience, and optimizes the app for production.

Requirements

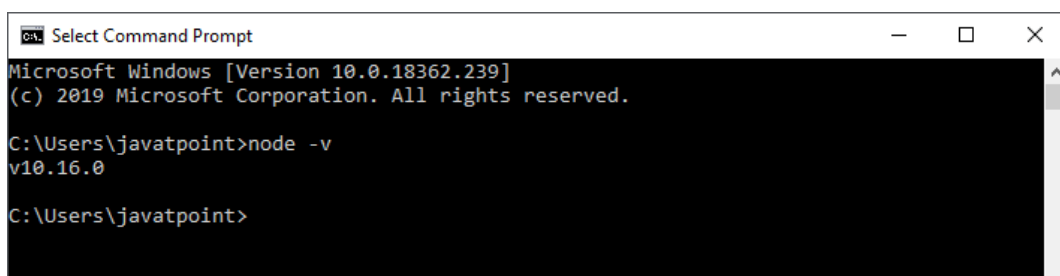
The Create React App is maintained by **Facebook** and can work on any **platform**, for example, macOS, Windows, Linux, etc. To create a React Project using create-react-app, you need to have installed the following things in your system.

1. Node version ≥ 8.10
2. NPM version ≥ 5.6

Let us check the current version of **Node** and **NPM** in the system.

Run the following command to check the Node version in the command prompt.

1. `$ node -v`



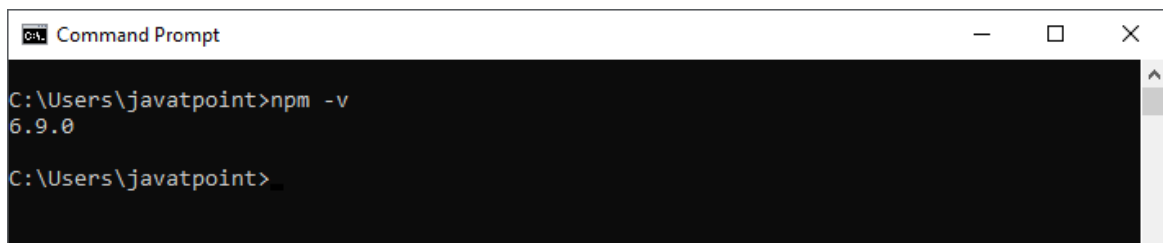
```
Microsoft Windows [Version 10.0.18362.239]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\javatpoint>node -v
v10.16.0

C:\Users\javatpoint>
```

Run the following command to check the NPM version in the command prompt.

1. `$ npm -v`



```
Command Prompt

C:\Users\javatpoint>npm -v
6.9.0

C:\Users\javatpoint>
```

Installation

Here, we are going to learn how we can install React using **CRA** tool. For this, we need to follow the steps as given below.

Install React

We can install React using npm package manager by using the following command. There is no need to worry about the complexity of React installation. The create-react-app npm package manager will manage everything, which is needed for React project.

1. `C:\Users\javatpoint> npm install -g create-react-app`

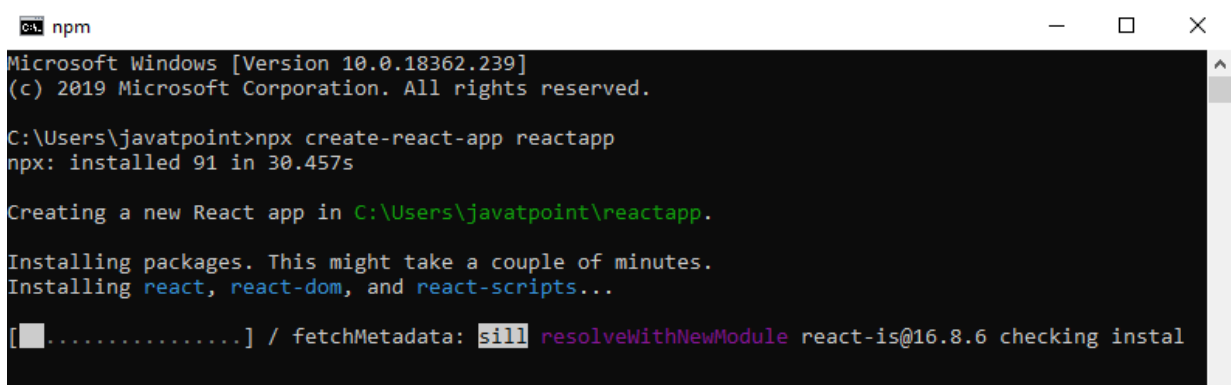
Create a new React project

Once the React installation is successful, we can create a new React project using create-react-app command. Here, I choose "reactproject" name for my project.

1. `C:\Users\javatpoint> create-react-app reactproject`

NOTE: We can combine the above two steps in a single command using npx. The npx is a package runner tool which comes with npm 5.2 and above version.

1. C:\Users\javatpoint> npx create-react-app reactproject



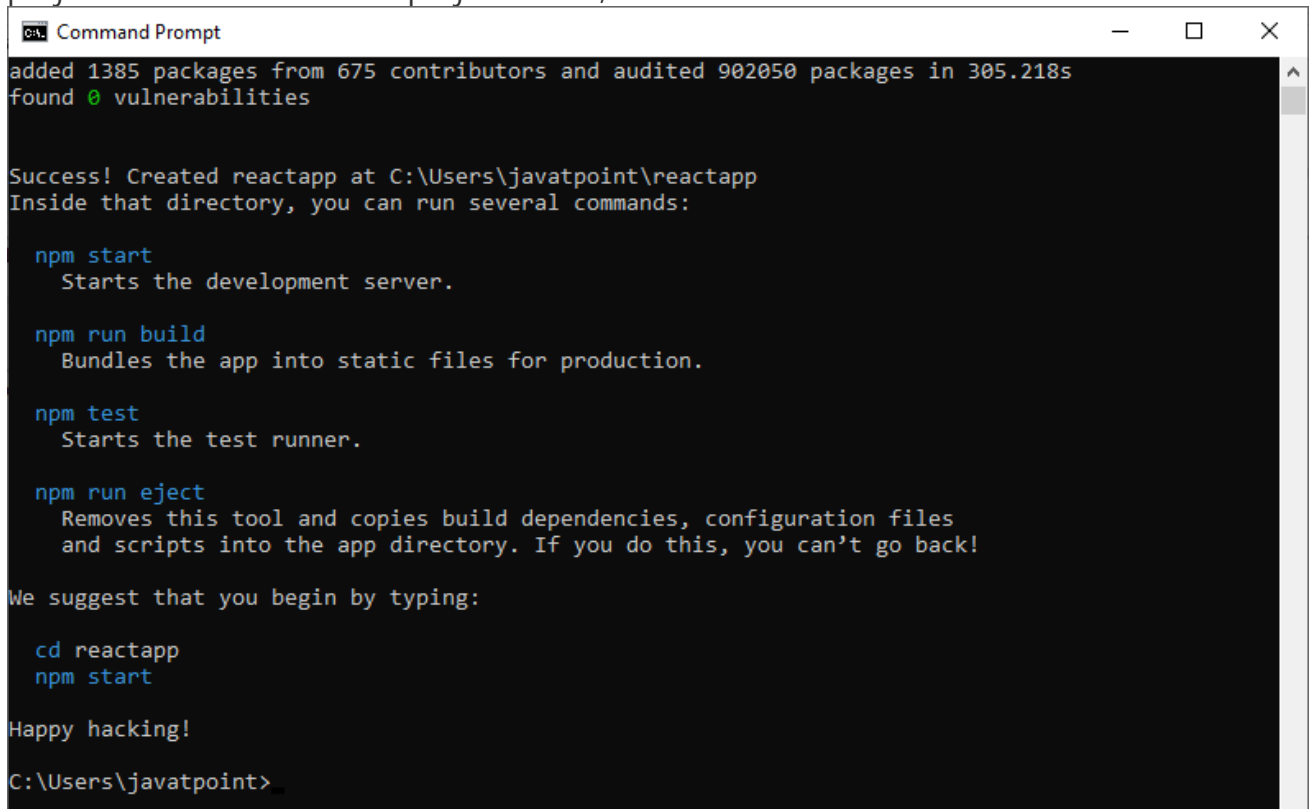
```
C:\Users\javatpoint>npx create-react-app reactapp
npx: installed 91 in 30.457s

Creating a new React app in C:\Users\javatpoint\reactapp.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts...

[ ] ..... / fetchMetadata: sill resolveWithNewModule react-is@16.8.6 checking instal
```

The above command will take some time to install the React and create a new project with the name "reactproject." Now, we can see the terminal as like below.



```
added 1385 packages from 675 contributors and audited 902050 packages in 305.218s
found 0 vulnerabilities

Success! Created reactapp at C:\Users\javatpoint\reactapp
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

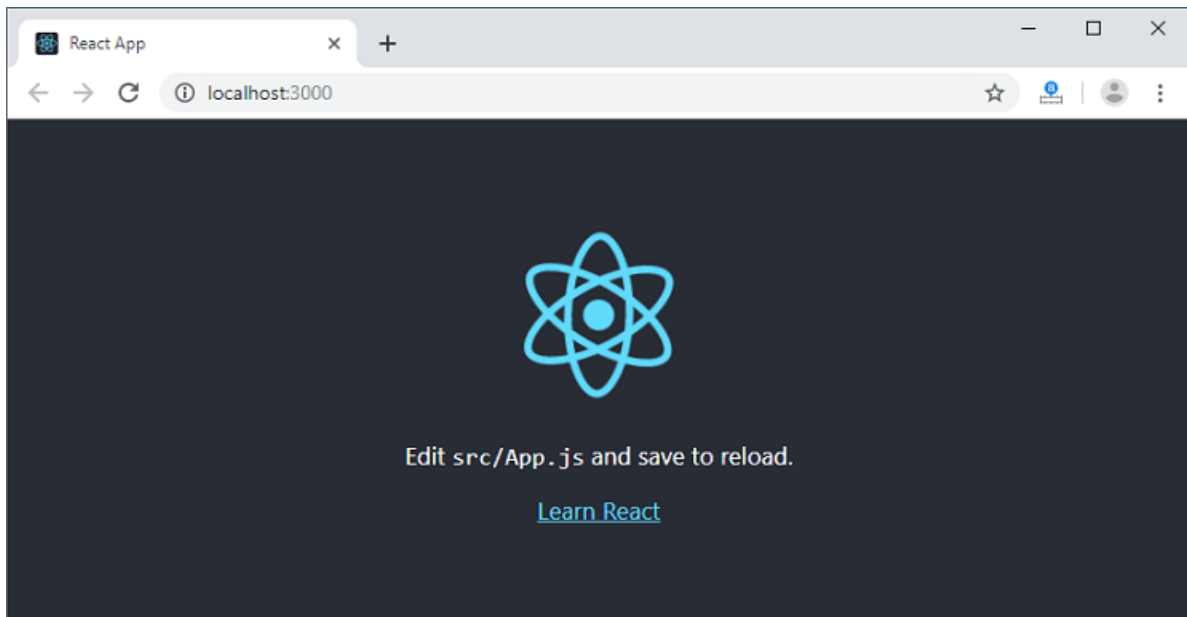
  cd reactapp
  npm start

Happy hacking!
C:\Users\javatpoint>
```

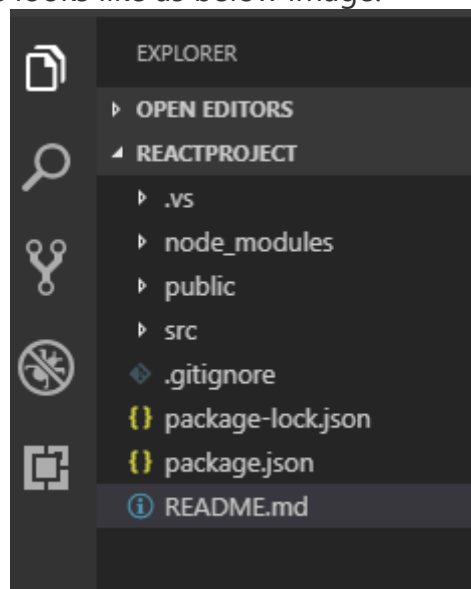
The above screen tells that the React project is created successfully on our system. Now, we need to start the server so that we can access the application on the browser. Type the following command in the terminal window.

1. \$ cd Desktop
2. \$ npm start

NPM is a package manager which starts the server and access the application at default server <http://localhost:3000>. Now, we will get the following screen.



Next, open the project on Code editor. Here, I am using Visual Studio Code. Our project's default structure looks like as below image.



In React application, there are several files and folders in the root directory. Some of them are as follows:

1. **node_modules:** It contains the React library and any other third party libraries needed.
2. **public:** It holds the public assets of the application. It contains the index.html where React will mount the application by default on the `<div id="root"></div>` element.
3. **src:** It contains the App.css, App.js, App.test.js, index.css, index.js, and serviceWorker.js files. Here, the App.js file always responsible for displaying the output screen in React.
4. **package-lock.json:** It is generated automatically for any operations where npm package modifies either the node_modules tree or package.json. It cannot be published. It will be ignored if it finds any other place rather than the top-level package.

5. **package.json:** It holds various metadata required for the project. It gives information to npm, which allows to identify the project as well as handle the project's dependencies.
6. **README.md:** It provides the documentation to read about React topics.

React Environment Setup

Now, open the **src >> App.js** file and make changes which you want to display on the screen. After making desired changes, **save** the file. As soon as we save the file, Webpack recompiles the code, and the page will refresh automatically, and changes are reflected on the browser screen. Now, we can create as many components as we want, import the newly created component inside the **App.js** file and that file will be included in our main **index.html** file after compiling by Webpack.

Next, if we want to make the project for the production mode, type the following command. This command will generate the production build, which is best optimized.

```
$ npm build
```

React Features

Currently, ReactJS is gaining quick popularity as the best JavaScript framework among web developers. It is playing an essential role in the front-end ecosystem. The important features of ReactJS are as following.

- JSX
- Components
- One-way Data Binding
- Virtual DOM
- Simplicity
- Performance

JSX

JSX stands for JavaScript XML. It is a JavaScript syntax extension. It's an XML or HTML like syntax used by ReactJS. This syntax is processed into JavaScript calls of React Framework. It extends the ES6 so that HTML like text can co-exist with JavaScript react code. It is not necessary to use JSX, but it is recommended to use in ReactJS.

Components

ReactJS is all about components. ReactJS application is made up of multiple components, and each component has its own logic and controls. These components can be reusable which help you to maintain the code when working on larger scale projects.

One-way Data Binding

ReactJS is designed in such a manner that follows unidirectional data flow or one-way data binding. The benefits of one-way data binding give you better control throughout the application. If the data flow is in another direction, then it requires additional features. It is because components are supposed to be immutable and the data within them cannot be changed. Flux is a pattern that helps to keep your data

unidirectional. This makes the application more flexible that leads to increase efficiency.

Virtual DOM

A virtual DOM object is a representation of the original DOM object. It works like a one-way data binding. Whenever any modifications happen in the web application, the entire UI is re-rendered in virtual DOM representation. Then it checks the difference between the previous DOM representation and new DOM. Once it has done, the real DOM will update only the things that have actually changed. This makes the application faster, and there is no wastage of memory.

Simplicity

ReactJS uses JSX file which makes the application simple and to code as well as understand. We know that ReactJS is a component-based approach which makes the code reusable as your need. This makes it simple to use and learn.

Performance

ReactJS is known to be a great performer. This feature makes it much better than other frameworks out there today. The reason behind this is that it manages a virtual DOM. The DOM is a cross-platform and programming API which deals with HTML, XML or XHTML. The DOM exists entirely in memory. Due to this, when we create a component, we did not write directly to the DOM. Instead, we are writing virtual components that will turn into the DOM leading to smoother and faster performance.

React JSX

As we have already seen that, all of the React components have a **render** function. The render function specifies the HTML output of a React component. JSX(JavaScript Extension), is a React extension which allows writing JavaScript code that looks like HTML. In other words, JSX is an HTML-like syntax used by React that extends ECMAScript so that **HTML-like** syntax can co-exist with JavaScript/React code. The syntax is used by **preprocessors** (i.e., transpilers like babel) to transform HTML-like syntax into standard JavaScript objects that a JavaScript engine will parse.

JSX provides you to write HTML/XML-like structures (e.g., DOM-like tree structures) in the same file where you write JavaScript code, then preprocessor will transform these expressions into actual JavaScript code. Just like XML/HTML, JSX tags have a tag name, attributes, and children.

Example

Here, we will write JSX syntax in JSX file and see the corresponding JavaScript code which transforms by preprocessor(babel).

JSX File

```
<div>Hello JavaTpoint</div>
```

Corresponding Output

1. `React.createElement("div", null, "Hello JavaTpoint");`

The above line creates a **react element** and passing **three arguments** inside where the first is the name of the element which is div, second is the **attributes** passed in

the div tag, and last is the **content** you pass which is the "Hello JavaTpoint."

Why use JSX?

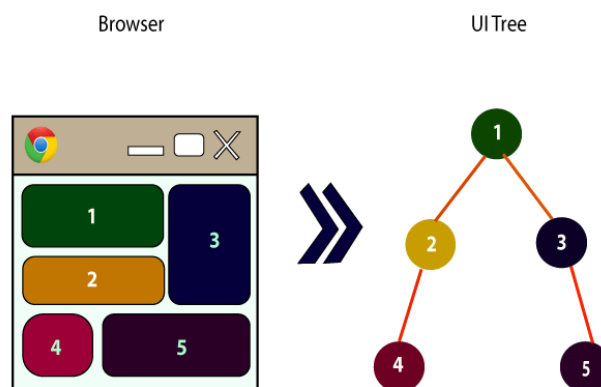
- It is faster than regular JavaScript because it performs optimization while translating the code to JavaScript.
- Instead of separating technologies by putting markup and logic in separate files, React uses components that contain both. We will learn components in a further section.
- It is type-safe, and most of the errors can be found at compilation time.
- It makes easier to create templates.

React Components

Earlier, the developers write more than thousands of lines of code for developing a single page application. These applications follow the traditional DOM structure, and making changes in them was a very challenging task. If any mistake found, it manually searches the entire application and update accordingly. The component-based approach was introduced to overcome an issue. In this approach, the entire application is divided into a small logical group of code, which is known as components.

A Component is considered as the core building blocks of a React application. It makes the task of building UIs much easier. Each component exists in the same space, but they work independently from one another and merge all in a parent component, which will be the final UI of your application.

Every React component have their own structure, methods as well as APIs. They can be reusable as per your need. For better understanding, consider the entire UI as a tree. Here, the root is the starting component, and each of the other pieces becomes branches, which are further divided into sub-branches.



In ReactJS, we have mainly two types of components. They are

1. Functional Components
2. Class Components

Functional Components

In React, function components are a way to write components that only contain a render method and don't have their own state. They are simply JavaScript functions that may or may not receive data as parameters. We can create a function that takes props(properties) as input

and returns what should be rendered. A valid functional component can be shown in the below example.

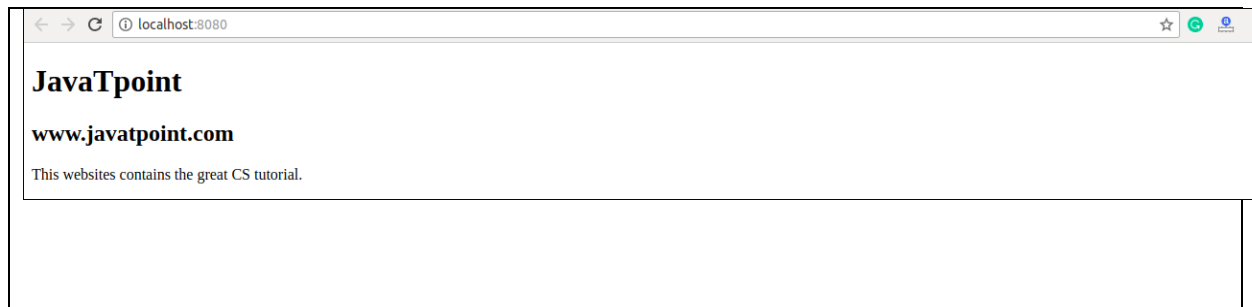
```
function WelcomeMessage(props) {  
  return <h1>Welcome to the , {props.name}</h1>;  
}
```

The functional component is also known as a stateless component because they do not hold or manage state. It can be explained in the below example.

Example

```
import React, { Component } from 'react';  
class App extends React.Component {  
  render() {  
    return (  
      <div>  
        <First/>  
        <Second/>  
      </div>  
    );  
  }  
}  
class First extends React.Component {  
  render() {  
    return (  
      <div>  
        <h1>JavaTpoint</h1>  
      </div>  
    );  
  }  
}  
class Second extends React.Component {  
  render() {  
    return (  
      <div>  
        <h2>www.javatpoint.com</h2>  
        <p>This websites contains the great CS tutorial.</p>  
      </div>  
    );  
  }  
}  
export default App;
```

Output:



Class Components

Class components are more complex than functional components. It requires you to extend from `React.Component` and create a render function which returns a React element. You can pass data from one class to other class components. You can create a class by defining a class that extends `Component` and has a render function. Valid class component is shown in the below example.

```
class MyComponent extends React.Component {
  render() {
    return (
      <div>This is main component.</div>
    );
  }
}
```

The class component is also known as a stateful component because they can hold or manage local state. It can be explained in the below example.

Example

In this example, we are creating the list of unordered elements, where we will dynamically insert `StudentName` for every object from the data array. Here, we are using ES6 arrow syntax (`=>`) which looks much cleaner than the old JavaScript syntax. It helps us to create our elements with fewer lines of code. It is especially useful when we need to create a list with a lot of items.

```
import React, { Component } from 'react';
class App extends React.Component {
  constructor() {
    super();
    this.state = {
      data:
      [
        {
          "name":"Abhishek"
        },
        {
          "name":"Saharsh"
        },
        {
          "name":"Ajay"
        }
      ]
    }
  }
}
```

```

    ]
  }
}
render() {
  return (
    <div>
      <StudentName/>
      <ul>
        {this.state.data.map((item) => <List data = {item} />)}
      </ul>
    </div>
  );
}
}
class StudentName extends React.Component {
  render() {
    return (
      <div>
        <h1>Student Name Detail</h1>
      </div>
    );
  }
}
class List extends React.Component {
  render() {
    return (
      <ul>
        <li>{this.props.data.name}</li>
      </ul>
    );
  }
}
export default App;

```

Output:

React State

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

A state must be kept as simple as possible. It can be set by using the **useState()** method and calling `setState()` method triggers UI updates. A state

represents the component's local state or information. It can only be accessed or modified inside the component or by the component directly. To set an initial state before any interaction occurs, we need to use the **getInitialState()** method.

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

Defining State

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

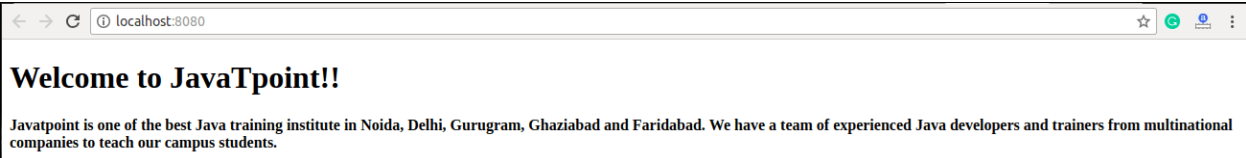
Example

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

```
import React, { Component } from 'react';
class App extends React.Component {
  constructor() {
    super();
    this.state = { displayBio: true };
  }
  render() {
    const bio = this.state.displayBio ? (
      <div>
        <p><h3>Javatpoint is one of the best Java training institute in Noida,
        Delhi, Gurugram, Ghaziabad and Faridabad. We have a team of experienced Java
        developers and trainers from multinational companies to teach our campus students.</h3></p>
      </div>
    ) : null;
    return (
      <div>
        <h1> Welcome to JavaTpoint!! </h1>
        { bio }
      </div>
    );
  }
}
export default App;
```

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

Output



Changing the State

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

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

Example

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

```
import React, { Component } from 'react';
class App extends React.Component {
  constructor() {
    super();
    this.state = { displayBio: false };
    console.log('Component this', this);
    this.toggleDisplayBio = this.toggleDisplayBio.bind(this);
  }
  toggleDisplayBio(){
    this.setState({displayBio: !this.state.displayBio});
  }
  render() {
    return (
      <div>
        <h1>Welcome to JavaTpoint!!</h1>
        {
          this.state.displayBio ? (
            <div>
              <p><h4>Javatpoint is one of the best Java training institute in
                Noida, Delhi, Gurugram, Ghaziabad and Faridabad. We have a team of experience
                d Java developers and trainers from multinational companies to teach our campu
                s students.</h4></p>
              <button onClick={this.toggleDisplayBio}> Show Less </button>
            </div>
          ) : (
            <div>
              <button onClick={this.toggleDisplayBio}> Read More </but
            </div>
          )
        }
      </div>
    );
  }
}
```

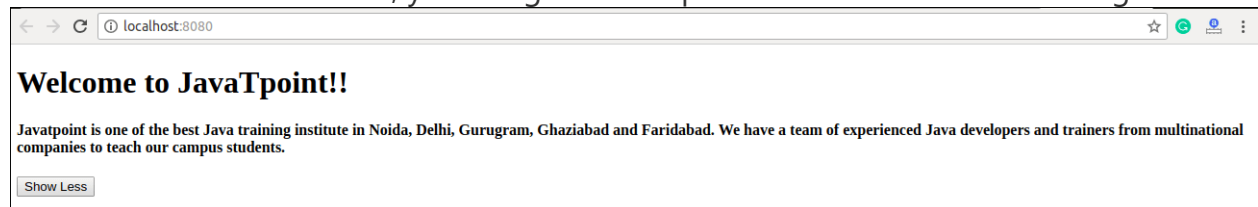
```

    </div>
  )
}
</div>
)
}
}
export default App;

```

Output:

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



React Props

Props stand for "**Properties**." They are **read-only** components. It is an object which stores the value of attributes of a tag and work similar to the HTML attributes. It gives a way to pass data from one component to other components. It is similar to function arguments. Props are passed to the component in the same way as arguments passed in a function.

Props are **immutable** so we cannot modify the props from inside the component. Inside the components, we can add attributes called props. These attributes are available in the component as **this.props** and can be used to render dynamic data in our render method.

When you need immutable data in the component, you have to add props to **ReactDOM.render()** method in the **main.js** file of your ReactJS project and used it inside the component in which you need. It can be explained in the below example.

Example

App.js

```

import React, { Component } from 'react';
class App extends React.Component {
  render() {
    return (
      <div>
        <h1> Welcome to { this.props.name } </h1>
        <p> <h4> Javatpoint is one of the best Java training institute in Noida, De
        lhi, Gurugram, Ghaziabad and Faridabad. </h4> </p>

```

```

    </div>
  );
}
}
export default App;

```

Main.js

```

import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';

```

```

ReactDOM.render(<App name = "JavaTpoint!!" />, document.getElementById('ap
p'));

```

Output**Default Props**

It is not necessary to always add props in the `ReactDOM.render()` element. You can also set **default** props directly on the component constructor. It can be explained in the below example.

Example**App.js**

```

import React, { Component } from 'react';
class App extends React.Component {
  render() {
    return (
      <div>
        <h1>Default Props Example</h1>
        <h3>Welcome to {this.props.name}</h3>
        <p>Javatpoint is one of the best Java training institute in Noida, Delhi, Gu
rugram, Ghaziabad and Faridabad.</p>
      </div>
    );
  }
}
App.defaultProps = {
  name: "JavaTpoint"
}
export default App;

```

Main.js

```

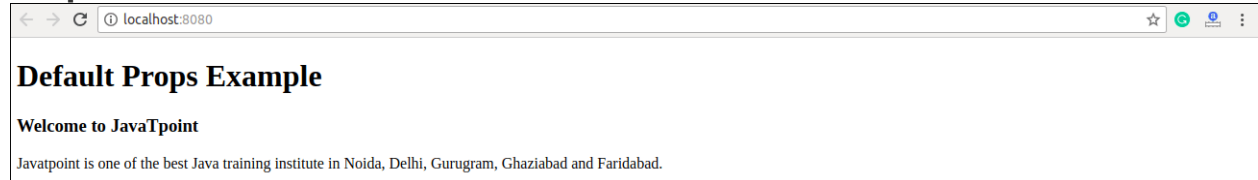
import React from 'react';

```

```
import ReactDOM from 'react-dom';
import App from './App.js';
```

```
ReactDOM.render(<App/>, document.getElementById('app'));
```

Output



State and Props

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

Example

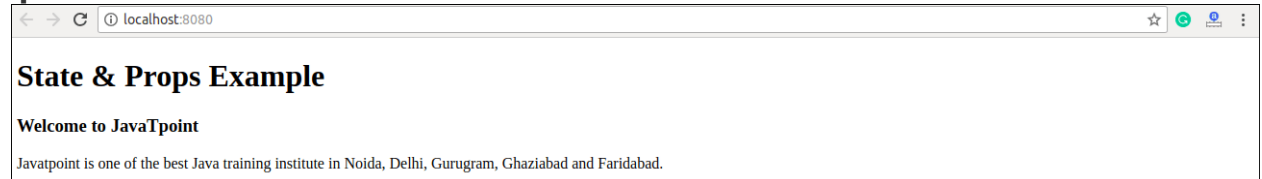
App.js

```
import React, { Component } from 'react';
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "JavaTpoint",
    }
  }
  render() {
    return (
      <div>
        <JTP jtpProp = {this.state.name}/>
      </div>
    );
  }
}
class JTP extends React.Component {
  render() {
    return (
      <div>
        <h1>State & Props Example</h1>
        <h3>Welcome to {this.props.jtpProp}</h3>
        <p>Javatpoint is one of the best Java training institute in Noida, Delhi, Gurugram, Ghaziabad and Faridabad.</p>
      </div>
    );
  }
}
```

```
}  
export default App;
```

Main.js

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

Output:

React Forms

Forms are an integral part of any modern web application. It allows the users to interact with the application as well as gather information from the users. Forms can perform many tasks that depend on the nature of your business requirements and logic such as authentication of the user, adding user, searching, filtering, booking, ordering, etc. A form can contain text fields, buttons, checkbox, radio button, etc.

Creating Form

React offers a stateful, reactive approach to build a form. The component rather than the DOM usually handles the React form. In React, the form is usually implemented by using controlled components.

There are mainly two types of form input in React.

1. Uncontrolled component
2. Controlled component

Uncontrolled component

The uncontrolled input is similar to the traditional HTML form inputs. The DOM itself handles the form data. Here, the HTML elements maintain their own state that will be updated when the input value changes. To write an uncontrolled component, you need to use a ref to get form values from the DOM. In other words, there is no need to write an event handler for every state update. You can use a ref to access the input field value of the form from the DOM.

Example

In this example, the code accepts a field **username** and **company name** in an uncontrolled component.

```
import React, { Component } from 'react';  
class App extends React.Component {  
  constructor(props) {  
    super(props);
```



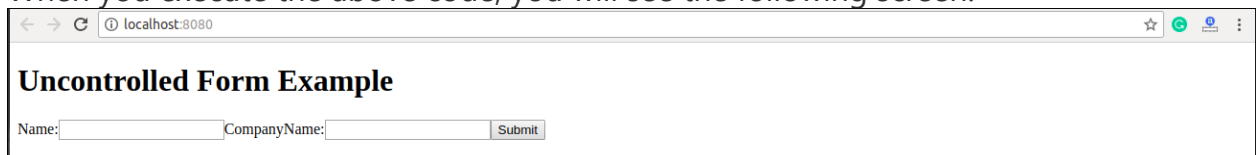
```
this.updateSubmit = this.updateSubmit.bind(this);
this.input = React.createRef();
}
updateSubmit(event) {
  alert('You have entered the UserName and CompanyName successfully.');
```

event.preventDefault();

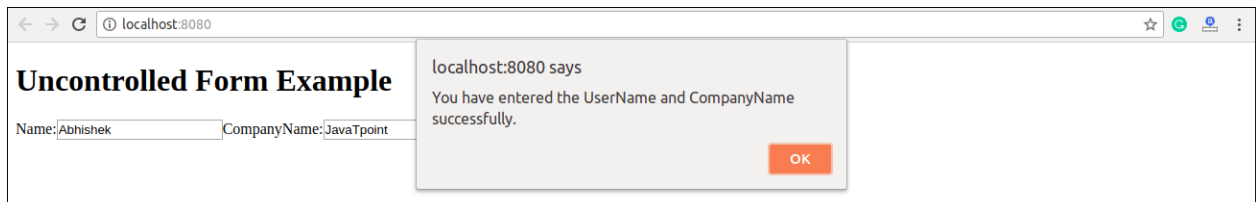
```
}
render() {
  return (
    <form onSubmit={this.updateSubmit}>
      <h1>Uncontrolled Form Example</h1>
      <label>Name:
        <input type="text" ref={this.input} />
      </label>
      <label>
        CompanyName:
        <input type="text" ref={this.input} />
      </label>
      <input type="submit" value="Submit" />
    </form>
  );
}
}
export default App;
```

Output

When you execute the above code, you will see the following screen.



After filling the data in the field, you get the message that can be seen in the below screen.



Controlled Component

In HTML, form elements typically maintain their own state and update it according to the user input. In the controlled component, the input form element is handled by the component rather than the DOM. Here, the mutable state is kept in the state property and will be updated only with **setState()** method.

Controlled components have functions that govern the data passing into them on every **onChange event**, rather than grabbing the data only once, e.g., when you click

a **submit button**. This data is then saved to state and updated with `setState()` method. This makes component have better control over the form elements and data.

A controlled component takes its current value through **props** and notifies the changes through **callbacks** like an `onChange` event. A parent component "controls" this changes by handling the callback and managing its own state and then passing the new values as props to the controlled component. It is also called as a "dumb component."

Example

```
import React, { Component } from 'react';
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ""};
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleChange(event) {
    this.setState({value: event.target.value});
  }
  handleSubmit(event) {
    alert("You have submitted the input successfully: " + this.state.value);
    event.preventDefault();
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <h1>Controlled Form Example</h1>
        <label>
          Name:
          <input type="text" value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
export default App;
```

Output

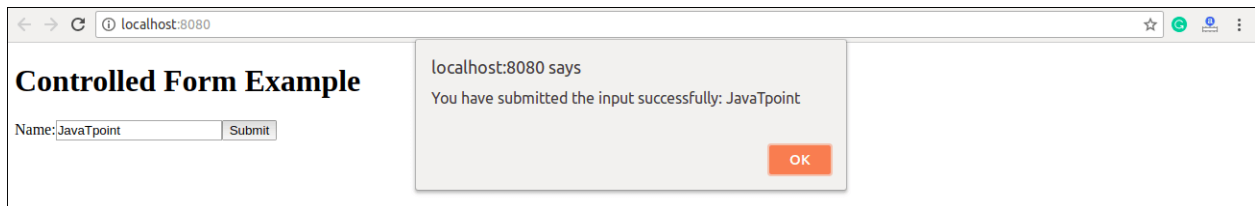
When you execute the above code, you will see the following screen.



Controlled Form Example

Name:

After filling the data in the field, you get the message that can be seen in the below screen.



Controlled Form Example

Name:

localhost:8080 says

You have submitted the input successfully: JavaTpoint

OK

Handling Multiple Inputs in Controlled Component

If you want to handle multiple controlled input elements, add a **name** attribute to each element, and then the handler function decided what to do based on the value of **event.target.name**.

Example

```
import React, { Component } from 'react';
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      personGoing: true,
      numberOfPersons: 5
    };
    this.handleChange = this.handleChange.bind(this);
  }
  handleChange(event) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
    const name = target.name;
    this.setState({
      [name]: value
    });
  }
  render() {
    return (
      <form>
        <h1>Multiple Input Controlled Form Example</h1>
        <label>
          Is Person going:
          <input
            name="personGoing"
            type="checkbox"
          />
        </label>
      </form>
    );
  }
}
```

```
        checked={this.state.personGoing}
        onChange={this.handleChange} />
    </label>
    <br />
    <label>
      Number of persons:
      <input
        name="numberOfPersons"
        type="number"
        value={this.state.numberOfPersons}
        onChange={this.handleChange} />
    </label>
  </form>
);
}
}
export default App;
```

Output



Multiple Input Controlled Form Example

Is Person going: ☒

Number of persons:

The Example

Before we go further, take a look at the following example:

What you have here is a simple React app that uses React Router to provide all of the navigation and view-loading goodness! Click on the various links to load the relevant content, and feel free to [open up this page in its own browser window](#) to use the back and forward buttons to see them working.

In the following sections, we are going to be building this app in pieces. By the end, not only will you have re-created this app, you'll hopefully have learned enough about React Router to build cooler and more awesomer things.

Getting Started

The first thing we need to do is get our project setup. We'll use our trusty **create-react-app** command to do this. From your favorite terminal, navigate to the folder you want to create your app, and type the following:

```
create-react-app react_spa
```

This will create our new project inside a folder called **react_spa**. Go ahead and navigate into this folder:

```
cd react_spa
```

Normally, this is where we start messing around with deleting the existing content to start from a blank slate. We will do that, but first, we are going to install React Router. To do that, run the following command:

```
npm i react-router-dom --save
```

This copies the appropriate React Router files and registers it in our **package.json** so that our app is made aware of its existence. That's good stuff, right?

Now that you've done this, it is time to clean up our project to start from a clean slate. From inside your **react_spa** folder, delete everything found inside your **public** and **src** folders. Once you've done this, let's create our index.html file that will serve as our app's starting point. In your **public** folder, create a file called **index.html** and add the following contents into it:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <title>React Router Example</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

Take a quick glance at the HTML. There shouldn't be anything surprising here. Next, we'll create our JavaScript entry point. Inside the **src** folder, create a file called **index.js** and add the following contents into it:

```
import React from "react";
import ReactDOM from "react-dom";
import Main from "./Main";

ReactDOM.render(
  <Main/>,
  document.getElementById("root")
)
```

```
);
```

Our ReactDOM.render call lives here, and what we are rendering is our Main component...which doesn't exist yet. The Main component will be the starting point for our SPA expedition using React Router, and we'll see how starting with the next section.

Building our Single Page App

The way we build our app is no different than all the apps we've been building so far. We will have a main parent component. Each of the individual "pages" of our app will be separate components that feed into the main component. The magic React Router brings to the table is basically choosing which components to show and which to hide. To make this feel natural, all of this *navigating* is tied in with our browser's address bar and back/forward buttons, so it is all made to look seamless..

Displaying the Initial Frame

When building a SPA, there will always be a part of your page that will remain static. This static part, also referred to as an **app frame**, could just be one invisible HTML element that acts as the container for all of your content, or could include some additional visual things like a header, footer, navigation, etc. In our case, our app frame will just be a component that contains UI elements for our navigation header and an empty area for content to load in.

Inside our **src** folder, create a new file called **Main.js** and add the following content into it:

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

class Main extends Component {
  render() {
    return (
      <div>
        <h1>Simple SPA</h1>
        <ul className="header">
          <li><a href="/">Home</a></li>
          <li><a href="/stuff">Stuff</a></li>
          <li><a href="/contact">Contact</a></li>
        </ul>
        <div className="content">

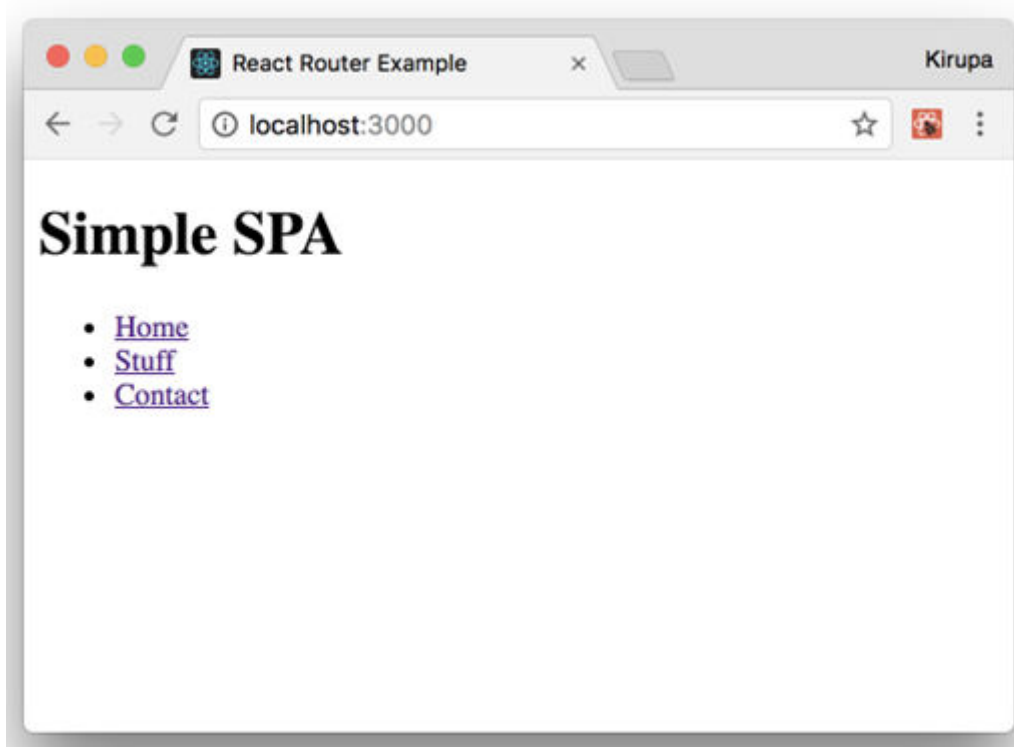
          </div>
        </div>
      );
    }
  }
```

```
}
```

```
export default Main;
```

Once you have pasted this, take a look at what we have here. We have a component called `Main` that returns some HTML. That's it. To see what we have so far in action, **npm start** it up and see what is going on in your browser.

You should see an unstyled version of an app title and some list items appear:



I know that this doesn't look all fancy and styled, but that's OK for now. We will deal with that later. The important thing to call out is that there is nothing React Router specific here. ABSOLUTELY NOTHING!

Creating our Content Pages

Our app will have three pages of content. This content will just be a simple component that prints out some JSX. Let's just get those created and out of the way! First, create a file called **Home.js** in our **src** directory and add the following content:

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

class Home extends Component {
  render() {
    return (
      <div>
        <h2>HELLO</h2>
        <p>Cras facilisis urna ornare ex volutpat, et
```

```

        convallis erat elementum. Ut aliquam, ipsum vitae
        gravida suscipit, metus dui bibendum est, eget rhoncus nibh
        metus nec massa. Maecenas hendrerit laoreet augue
        nec molestie. Cum sociis natoque penatibus et magnis
        dis parturient montes, nascetur ridiculus mus.</p>

        <p>Duis a turpis sed lacus dapibus elementum sed eu lectus.</p>
    </div>
    );
  }
}

export default Home;

```

Next, create a file called **Stuff.js** in the same location and add in the following:

```

import React, { Component } from "react";

class Stuff extends Component {
  render() {
    return (
      <div>
        <h2>STUFF</h2>
        <p>Mauris sem velit, vehicula eget sodales vitae,
        rhoncus eget sapien:</p>
        <ol>
          <li>Nulla pulvinar diam</li>
          <li>Facilisis bibendum</li>
          <li>Vestibulum vulputate</li>
          <li>Eget erat</li>
          <li>Id porttitor</li>
        </ol>
      </div>
    );
  }
}

export default Stuff;

```

We just have one more page left. Create a file called **Contact.js** in our **src** folder and make sure its contents are the following:

```

import React, { Component } from "react";

class Contact extends Component {
  render() {
    return (
      <div>
        <h2>GOT QUESTIONS?</h2>

```



```
    <p>The easiest thing to do is post on  
    our <a href="http://forum.kirupa.com">forums</a>.  
    </p>  
  </div>  
);  
}  
}  
  
export default Contact;
```

That's the last of our content we are going to add. If you took a look at what it is you were adding, you'll see that these components can't get any simpler. They just returned some boilerplate JSX content. Now, make sure to save all of your changes to these three files. We'll look at how to make them useful shortly.

Using React Router

We have our app frame in the form of our `Main` component. We have our content pages represented by the `Home`, `Stuff`, and `Contact` components. What we need to do is tie all of these together to create our app. This is where React Router comes in. To start using it, go back to **Main.js**, and ensure your import statements look as follows:

```
import React, { Component } from "react";  
import {  
  Route,  
  NavLink,  
  HashRouter  
} from "react-router-dom";  
import Home from "./Home";  
import Stuff from "./Stuff";  
import Contact from "./Contact";
```

We are importing **Route**, **NavLink**, and **HashRouter** from the `react-router-dom` NPM package we installed earlier. In addition, we are importing our `Home`, `Stuff`, and `Contact` components since we will be referencing them as part of loading our content.

The way React Router works is by defining what I call a **routing region**. Inside this region, you will have two things:

- i. Your navigation links
- ii. The container to load your content into

There is a close correlation between what URL your navigation links specify and the content that ultimately gets loaded. There is no way to easily explain this without first getting our hands dirty and implementing what we just read about.

The first thing we are going to do is define our **routing region**. Inside our Main component's render method, add the following highlighted lines:

```
class Main extends Component {
  render() {
    return (
      <HashRouter>
        <div>
          <h1>Simple SPA</h1>
          <ul className="header">
            <li><a href="/">Home</a></li>
            <li><a href="/stuff">Stuff</a></li>
            <li><a href="/contact">Contact</a></li>
          </ul>
          <div className="content">

            </div>
        </div>
      </HashRouter>
    );
  }
}
```

The HashRouter component provides the foundation for the navigation and browser history handling that routing is made up of. What we are going to do next is define our navigation links. We already have list elements with the a element defined. We need to replace them with the more specialized NavLink component, so go ahead and make the following highlighted changes:

```
class Main extends Component {
  render() {
    return (
      <HashRouter>
        <div>
          <h1>Simple SPA</h1>
          <ul className="header">
            <li><NavLink to="/">Home</NavLink></li>
            <li><NavLink to="/stuff">Stuff</NavLink></li>
            <li><NavLink to="/contact">Contact</NavLink></li>
          </ul>
          <div className="content">

            </div>
        </div>
      </HashRouter>
    );
  }
}
```

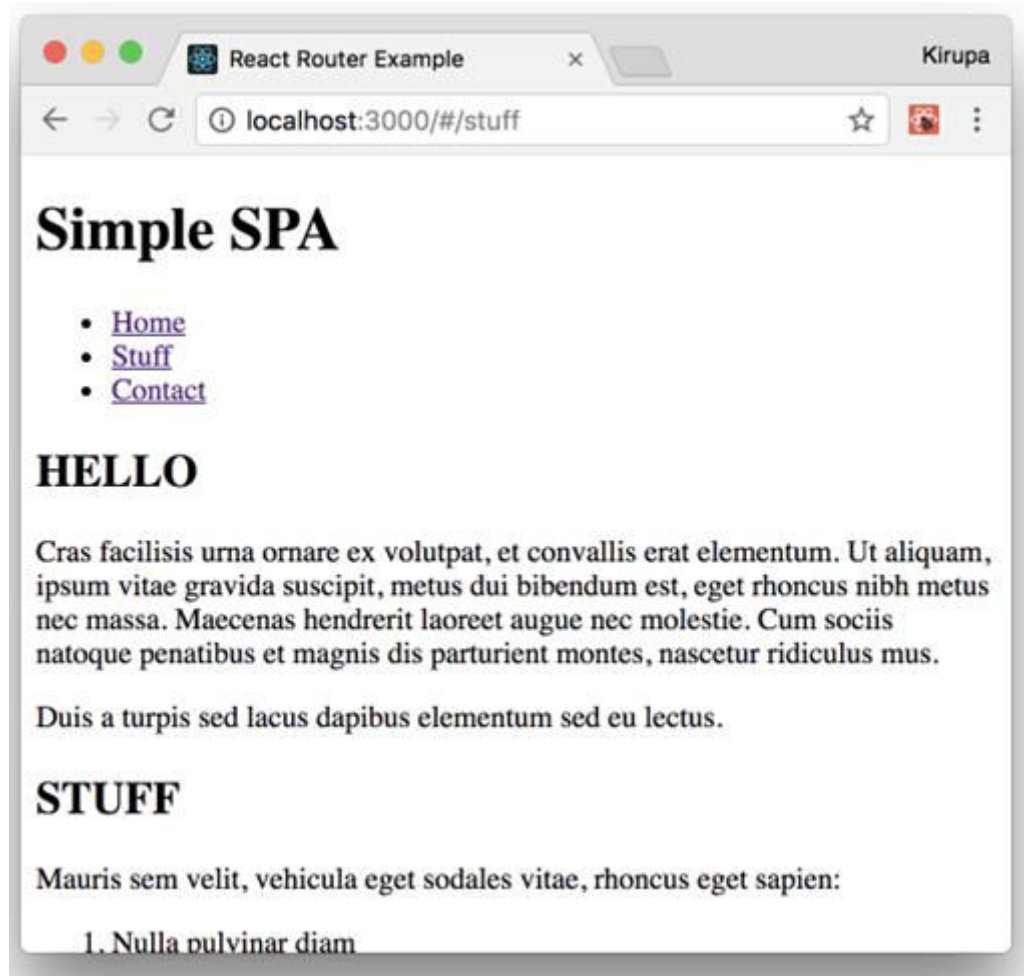
```
}
```

For each link, pay attention to the URL we are telling our router to navigate to. This URL value (defined by the `to` prop) acts as an identifier to ensure the right content gets loaded. The way we match the URL with the content is by using a `Route` component. Go ahead and add the following highlighted lines:

```
class Main extends Component {  
  render() {  
    return (  
      <HashRouter>  
        <div>  
          <h1>Simple SPA</h1>  
          <ul className="header">  
            <li><NavLink to="/">Home</NavLink></li>  
            <li><NavLink to="/stuff">Stuff</NavLink></li>  
            <li><NavLink to="/contact">Contact</NavLink></li>  
          </ul>  
          <div className="content">  
            <Route path="/" component={Home}/>  
            <Route path="/stuff" component={Stuff}/>  
            <Route path="/contact" component={Contact}/>  
          </div>  
        </div>  
      </HashRouter>  
    );  
  }  
}
```

As you can see, the `Route` component contains a `path` prop. The value you specify for the `path` determines when this route is going to be active. When a route is active, the component specified by the `component` prop gets rendered. For example, when we click on the **Stuff** link (whose path is **/stuff** as set by the `NavLink` component's `to` prop), the route whose `path` value is also **/stuff** becomes active. This means the contents of our `Stuff` component get rendered.

You can see all of this for yourself. Jump back to your browser to see the live updates or run `npm start` again. Click around on the links to see the content loading in and out. Something seems off, though, right? The content for our home page seems to always display even if we are clicking on the `Stuff` or `Contact` links:



That seems problematic. We'll look at how to fix that and do many more little housekeeping tasks in the next section when we go one level deeper into using React Router.

It's the Little Things

In the previous section, we got our SPA mostly up and running. We just wrapped our entire routing region inside a `HashRouter` component, and we separated our links and the place our links would load by using the `NavLink` and `Route` components respectively. Getting our example *mostly* up and running and *fully* up and running are two different things. In the following sections, we'll close those differences.

Fixing our Routing

We ended the previous section by calling out that our routing has a bug in it. The contents of our `Home` component is always displaying. The reason for it is because the path for loading our `Home` component is `/`. Our `Stuff` and `Contact` components have the `/` character as part of their paths as well. This means our **Home** component always matches whatever path we are trying to navigate to. The fix for that is simple. In

the `Route` component representing our home content, add the `exact` prop as highlighted below:

```
<div className="content">
  <Route exact path="/" component={Home}/>
  <Route path="/stuff" component={Stuff}/>
  <Route path="/contact" component={Contact}/>
</div>
```

This prop ensures the `Route` is active only if the path is an exact match for what is being loaded. If you preview your app now, you'll see that the content loads correctly with the home content only displaying when our app is in the home view.

Adding Some CSS

Right now, our app is completely unstyled. The fix for that is easy. In your **src** folder, create a file called **index.css** and add the following style rules into it:

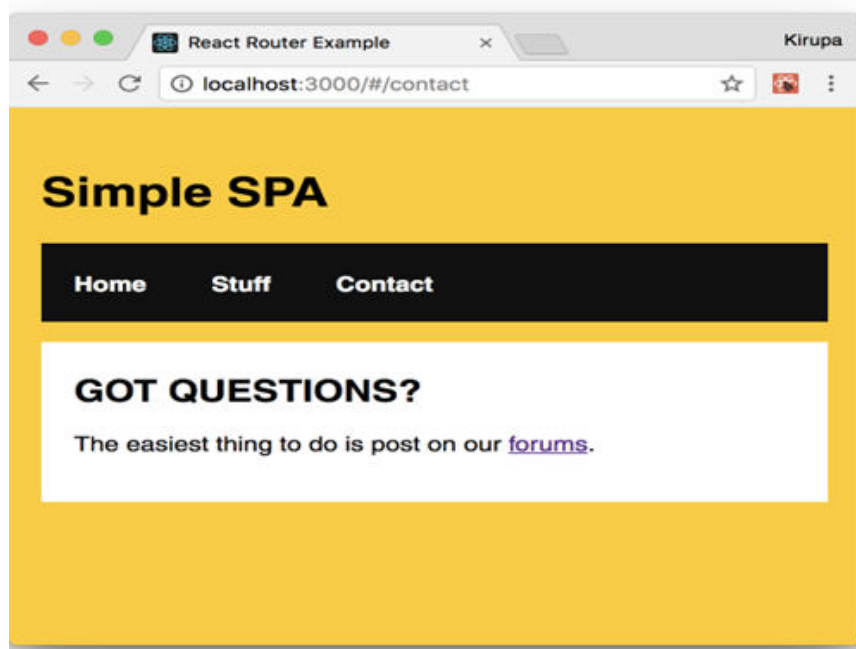
```
body {
  background-color: #FFCC00;
  padding: 20px;
  margin: 0;
}
h1, h2, p, ul, li {
  font-family: sans-serif;
}
ul.header li {
  display: inline;
  list-style-type: none;
  margin: 0;
}
ul.header {
  background-color: #111;
  padding: 0;
}
ul.header li a {
  color: #FFF;
  font-weight: bold;
  text-decoration: none;
  padding: 20px;
  display: inline-block;
}
.content {
  background-color: #FFF;
  padding: 20px;
}
.content h2 {
  padding: 0;
  margin: 0;
}
```

```
.content li {  
  margin-bottom: 10px;  
}
```

After you've done this, we need to reference this style sheet in our app. At the top of **index.js**, add the import statement to do just that:

```
import React from "react";  
import ReactDOM from "react-dom";  
import Main from "./Main";  
import "./index.css";  
ReactDOM.render(  
  <Main/>,  
  document.getElementById("root")  
>);
```

Save all of your changes if you haven't done so yet. If you preview the app now, you'll notice that it is starting to look a bit more like the example we started out with:



We are almost done here! There is just a few more things we need to do.

Highlighting the Active Link

Right now, it's hard to tell which link corresponds to content that is currently loaded. It would be useful to have some sort of a visual cue to solve this. The creators of React Router have already thought of that! When you click on a link, a **class** value of **active** is automatically assigned to it.

For example, this is what the HTML for clicking on the Stuff link looks like:

```
<a aria-current="true" href="#/stuff" class="active">Stuff</a>
```

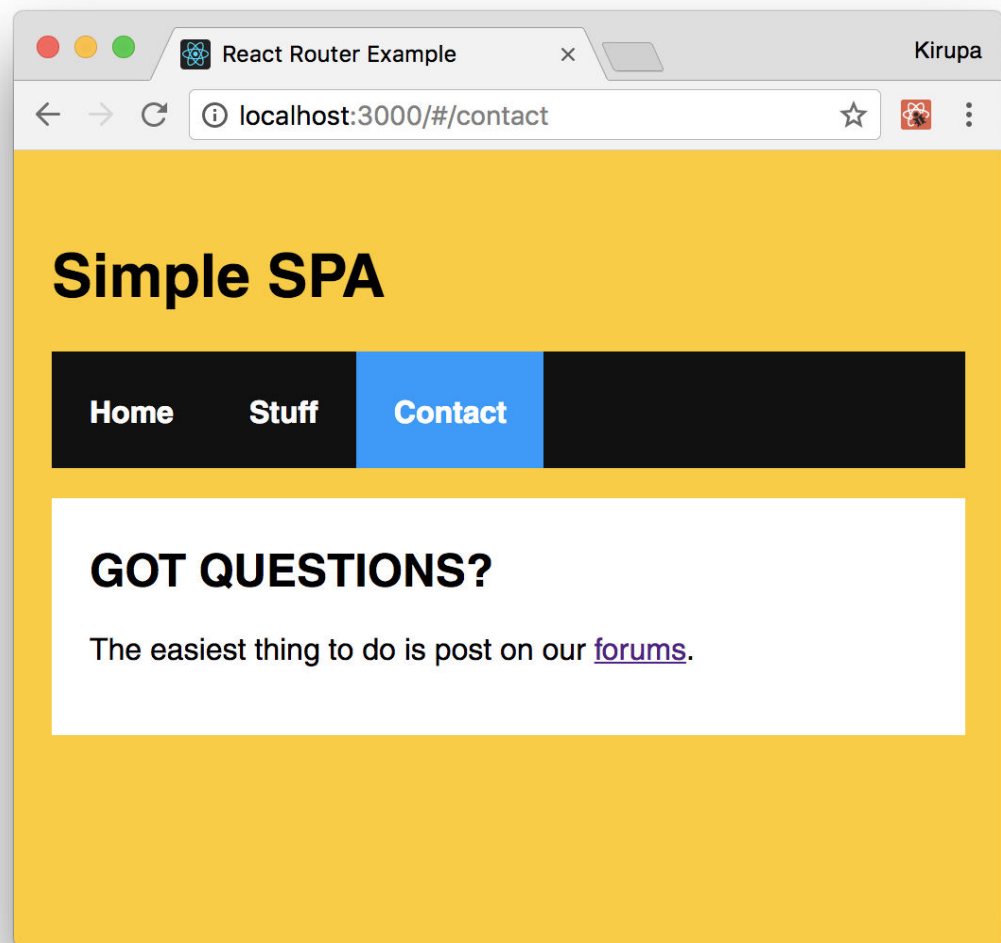
This means all we really have to do is add the appropriate CSS that lights up when an element has a `class` value of **active** set on it. To make this happen, go back to **index.css** and add the following style rule towards the bottom of your document:

```
.active {  
  background-color: #0099FF;  
}
```

Once you have added this rule and saved your document, go back to your browser and click around on the links in our example. You'll see that the active link whose content is displayed from is highlighted with a blue color. What you will also see is our Home link always highlighted. That isn't correct. The fix for that is simple. Just add the **exact** prop to our `NavLink` component representing our home content:

```
<li><NavLink exact to="/">Home</NavLink></li>  
<li><NavLink to="/stuff">Stuff</NavLink></li>  
<li><NavLink to="/contact">Contact</NavLink></li>
```

Once you have done that, go back to our browser. You'll see that our Home link only gets the active color treatment when the home content is displayed:



UNIT - IV

Java Web Development: JAVA PROGRAMMING BASICS, Model View Controller (MVC) Pattern MVC Architecture using Spring RESTful API using Spring Framework Building an application using Maven

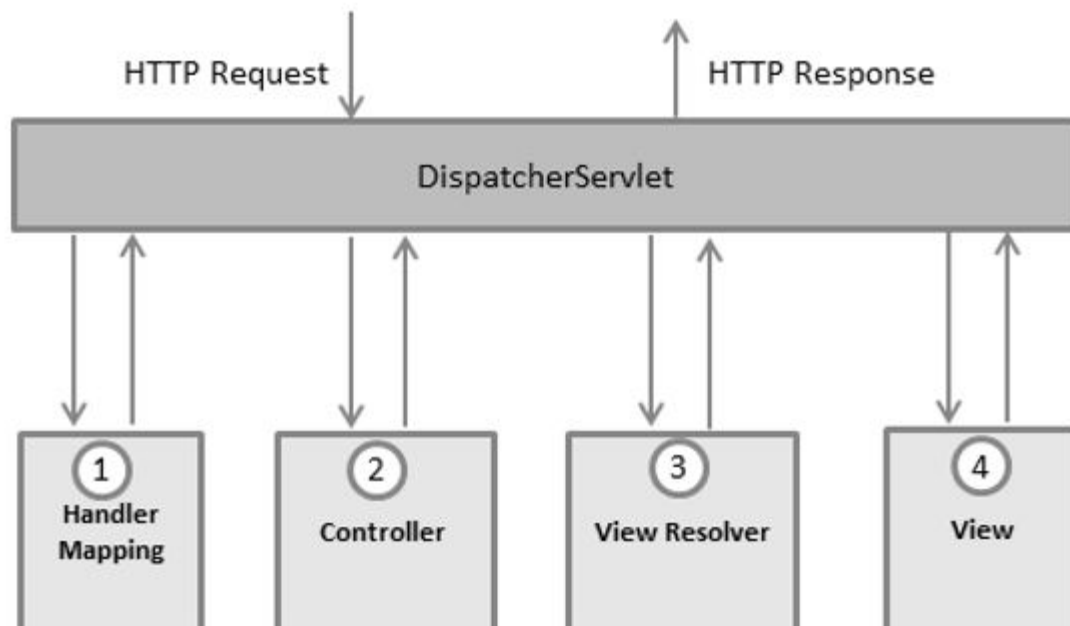
Model View Controller (MVC) Pattern MVC Architecture using Spring

The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

- The **Model** encapsulates the application data and in general they will consist of POJO.
- The **View** is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
- The **Controller** is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.

The DispatcherServlet

The Spring Web model-view-controller (MVC) framework is designed around a *DispatcherServlet* that handles all the HTTP requests and responses. The request processing workflow of the Spring Web MVC *DispatcherServlet* is illustrated in the following diagram –



Following is the sequence of events corresponding to an incoming HTTP request to *DispatcherServlet* –

- After receiving an HTTP request, *DispatcherServlet* consults the *HandlerMapping* to call the appropriate *Controller*.
- The *Controller* takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the *DispatcherServlet*.
- The *DispatcherServlet* will take help from *ViewResolver* to pickup the defined view for the request.
- Once view is finalized, The *DispatcherServlet* passes the model data to the view which is finally rendered on the browser.

All the above-mentioned components, i.e. *HandlerMapping*, *Controller*, and *ViewResolver* are parts of *WebApplicationContext* which is an extension of the plain *ApplicationContext* with some extra features necessary for web applications.

Required Configuration

You need to map requests that you want the *DispatcherServlet* to handle, by using a URL mapping in the **web.xml** file. The following is an example to show declaration and mapping for **HelloWeb** *DispatcherServlet* example –

```
<web-app id = "WebApp_ID" version = "2.4"
  xmlns = "http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>Spring MVC Application</display-name>

  <servlet>
    <servlet-name>HelloWeb</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>HelloWeb</servlet-name>
    <url-pattern>*.jsp</url-pattern>
  </servlet-mapping>

</web-app>
```

The **web.xml** file will be kept in the WebContent/WEB-INF directory of your web application. Upon initialization of **HelloWeb** *DispatcherServlet*, the framework will try to load the application context from a file named **[servlet-name]-servlet.xml** located in the application's WebContent/WEB-INF directory. In this case, our file will be **HelloWebServlet.xml**.

Next, `<servlet-mapping>` tag indicates what URLs will be handled by which *DispatcherServlet*. Here all the HTTP requests ending with **.jsp** will be handled by

the **HelloWeb** DispatcherServlet.

If you do not want to go with default filename as `[servlet-name]-servlet.xml` and default location as `WebContent/WEB-INF`, you can customize this file name and location by adding the servlet listener `ContextLoaderListener` in your `web.xml` file as follows –

```
<web-app...>

  <!------- DispatcherServlet definition goes here----->
  ....
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/HelloWeb-servlet.xml</param-value>
  </context-param>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

</web-app>
```

Now, let us check the required configuration for **HelloWeb-servlet.xml** file, placed in your web application's `WebContent/WEB-INF` directory –

```
<beans xmlns = "http://www.springframework.org/schema/beans"
  xmlns:context = "http://www.springframework.org/schema/context"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <context:component-scan base-package = "com.tutorialspoint" />

  <bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name = "prefix" value = "/WEB-INF/jsp/" />
    <property name = "suffix" value = ".jsp" />
  </bean>

</beans>
```

Following are the important points about **HelloWeb-servlet.xml** file –

- The `[servlet-name]-servlet.xml` file will be used to create the beans defined, overriding the definitions of any beans defined with the same name in the global scope.
- The `<context:component-scan...>` tag will be use to activate Spring MVC annotation scanning capability which allows to make use of annotations like `@Controller` and `@RequestMapping` etc.

- The *InternalResourceViewResolver* will have rules defined to resolve the view names. As per the above defined rule, a logical view named **hello** is delegated to a view implementation located at */WEB-INF/jsp/hello.jsp*.

The following section will show you how to create your actual components, i.e., Controller, Model, and View.

Defining a Controller

The *DispatcherServlet* delegates the request to the controllers to execute the functionality specific to it. The **@Controller** annotation indicates that a particular class serves the role of a controller. The **@RequestMapping** annotation is used to map a URL to either an entire class or a particular handler method.

```
@Controller
@RequestMapping("/hello")
public class HelloController {
    @RequestMapping(method = RequestMethod.GET)
    public String printHello(ModelMap model) {
        model.addAttribute("message", "Hello Spring MVC Framework!");
        return "hello";
    }
}
```

The **@Controller** annotation defines the class as a Spring MVC controller. Here, the first usage of **@RequestMapping** indicates that all handling methods on this controller are relative to the */hello* path.

Next annotation **@RequestMapping(method = RequestMethod.GET)** is used to declare the *printHello()* method as the controller's default service method to handle HTTP GET request. You can define another method to handle any POST request at the same URL.

You can write the above controller in another form where you can add additional attributes in **@RequestMapping** as follows –

```
@Controller
public class HelloController {
    @RequestMapping(value = "/hello", method = RequestMethod.GET)
    public String printHello(ModelMap model) {
        model.addAttribute("message", "Hello Spring MVC Framework!");
        return "hello";
    }
}
```

The **value** attribute indicates the URL to which the handler method is mapped and the **method** attribute defines the service method to handle HTTP GET request. The following important points are to be noted about the controller defined above –

- You will define required business logic inside a service method. You can call another method inside this method as per requirement.
- Based on the business logic defined, you will create a model within this method. You can use setter different model attributes and these attributes will

be accessed by the view to present the final result. This example creates a model with its attribute "message".

- A defined service method can return a String, which contains the name of the **view** to be used to render the model. This example returns "hello" as logical view name.

Creating JSP Views

Spring MVC supports many types of views for different presentation technologies. These include - JSPs, HTML, PDF, Excel worksheets, XML, Velocity templates, XSLT, JSON, Atom and RSS feeds, JasperReports, etc. But most commonly we use JSP templates written with JSTL.

Let us write a simple **hello** view in /WEB-INF/hello/hello.jsp –

```
<html>
<head>
  <title>Hello Spring MVC</title>
</head>

<body>
  <h2>${message} </h2>
</body>
</html>
```

Here **\${message}** is the attribute which we have set up inside the Controller. You can have multiple attributes to be displayed inside your view.

Spring Web MVC Framework Examples

Based on the above concepts, let us check few important examples which will help you in building your Spring Web Applications –

Sr.No.	Example & Description
1	<u>Spring MVC Hello World Example</u> This example will explain how to write a simple Spring Web Hello World application.
2	<u>Spring MVC Form Handling Example</u> This example will explain how to write a Spring Web application using HTML forms to submit the data to the controller and display a processed result.
3	<u>Spring Page Redirection Example</u> Learn how to use page redirection functionality in Spring MVC Framework.

4	<u>Spring Static Pages Example</u> Learn how to access static pages along with dynamic pages in Spring MVC Framework.
5	<u>Spring Exception Handling Example</u> Learn how to handle exceptions in Spring MVC Framework.

https://www.tutorialspoint.com/spring/spring_web_mvc_framework.htm

RESTful API using Spring Framework

Web-based application development is a common part of [Java](#) development. It is part and parcel of the enterprise domain wherein they share many common attributes of building a production-ready application. Spring uniquely addresses the concern for building a Web application through its MVC framework. It is called *MVC* because it is based upon the MVC (Model-View-Controller) pattern. Refer to [Wikipedia: Model-view-controller](#) for quick information about this. Web applications, in most cases, have a REST counterpart for resource sharing. This article builds up on both the idea and ends with a quick example to describe them in a terse manner.

Spring MVC

A Web application is inherently multi-layered because the intricacies between the user request and server response go through several in-between stages of information processing. Each stage is handled by a layer. For example, the Web interaction begins with user interaction with the browser, such as by triggering a request and getting a response from the server. The request response paradigm is nothing more than an exchange of certain arranged data, which can be anywhere from trivial to heavily loaded information gathered from, for example, a form submitted by the user. The URL encapsulates the request from the user and flutters into the network oblivion. Voilà! It is returned back with the digital PIZZA you have requested onto the platter called a browser. The request actually goes through a bunch of agents under the purview of the Spring MVC framework. Each of these agents performs specific functions, technically called *request processing*, before actually responding back to the requester. Here is an illustration to give you an idea.

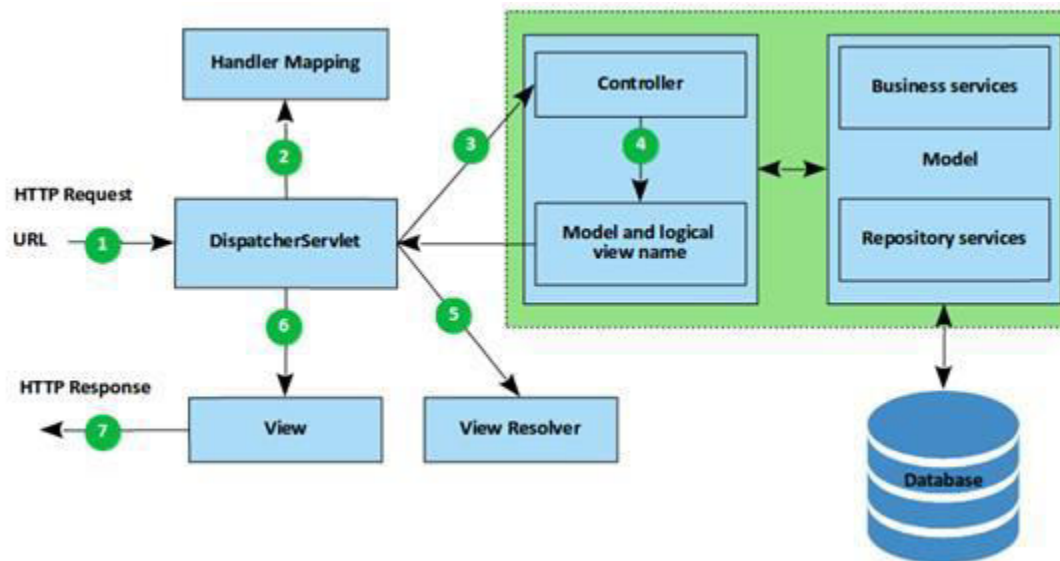


Figure 1: The Spring framework

1. The journey begins with the *HTTP request* (sometimes with data payload; for example, due to form submission) in a URL. It first stations at *DispatcherServlet*. The *DispatcherServlet* is a class defined in the *org.springframework.web.servlet* package. It is the central dispatcher, a Java Servlet Component for the Spring MVC framework. This front controller receives all incoming HTTP client requests and delegates responsibilities to other components for further processing of the request payload.
2. The *handler mapping* decides where the request's next stop would be. It acts as a consultant to the central dispatcher (*DispatcherServlet*) for routing the request to the appropriate controller. The handler mapping parses the request URL to make decisions and the dispatcher then delegates the request to the controller.
3. The *controller's* responsibility is to process the information payload received from the request. Typically, a controller is associated with one or more business service classes which, in turn, may have associated database services repository classes. The repository classes fetch database information according to the business service logic. It is the business service classes that contain the crux of processing. The controller class simply carries the information received from one or more service classes to the user. However, the response of the controller classes is still raw data referred to as the *model* and may not be user friendly (with indentation, bullets, tables, images, look-and-feel, and so forth).
4. Therefore, the controller packages the model data along with model and view name back again to the *central dispatcher, DispatcherServlet*.
5. The *view layer* can be designed using any third-party framework such as Node.js, Angular, JSP, and so on. The controller is decoupled from the view by passing the view name to the *DispatcherServlet* and is least interested in it. The *DispatcherServlet* simply carries the logical name and consults with the view resolver to map the logical view name with the actual implementation.

6. Once the mapping between logical view name and the actual view implementation is made, the *DispatcherServlet* delegates the responsibility of rendering model data to the view implementation.
7. The *view implementation* finally carries the response back to the client browser.

REST

REST is the acronym of *Representational State Transfer*. It is a term coined in [Roy Thomas Fielding's doctoral thesis](#) where REST is a part that encompasses the architecture of transferring the state of resources. The REST architecture is made popular as an alternative to a SOAP implementation of Web services. Although REST has a much wider connotation than just Web services, here we'll limit our discussion to dealing with REST resources only. The idea Web services are basically resource sharing in the Web architecture that forms the cornerstone of distributed machine-to-machine communication. The Spring MVC framework resides pretty well with REST and provides the necessary API support to implement it seamlessly, with little effort.

The URL and HTTP Methods

The REST resources are located on a remote host using URL. The idea is based on the foundation of the protocol called *HTTP*. For example, the URL <http://www.payroll.com/employees> may mean a list of employees to search and <http://www.payroll.com/employees/101> may mean the detail of an employee with, say, ID 101. Hence, the URL/URI actually represents the actual location of a resource on the Web. The resource may be anything a Web page, an image, audio, video content, or the like. The HTTP protocol specifies several methods. If they are combined with the URL that points to the resource, we can get the following CRUD results as outlined below.

URL	Method	Outcome
http://www.payroll.com/employees	POST	Creates a list of employees
http://www.payroll.com/employees	PUT or PATCH	Updates a list of employees
http://www.payroll.com/employees	DELETE	Deletes a list of employees
http://www.payroll.com/employees	GET	Gets a list of employees

http://www.payroll.com/employees/101	POST	Creates a employee with ID 101
http://www.payroll.com/employees/101	PUT or PATCH	Updates employee with ID 101
http://www.payroll.com/employees/101	DELETE	Deletes employee with ID 101
http://www.payroll.com/employees/101	GET	Gets employee details with ID 101

Though the URL is associated with HTTP methods in REST, there are no strict rules to adhere to the outcome described above. The point is that RESTful URL structure should be able to locate a resource on the server. For instance, the PUT instruction can be used to create a new resource and POST can be used to update a resource.

REST in Spring

The REST API support was introduced in Spring from version 3.0 onwards; since then, it has steadily evolved to the present day. We can create REST resources in the following ways:

- Using controllers which are used to handle HTTP requests such as GET, POST, PUT, and so forth. The PATCH command is supported by Spring 3.2 and higher versions.
- Using the *@PathVariable* annotation. This annotation is used to handle parameterized URLs. This is usually associated with the *@RequestMapping* handler method in a Servlet environment.
- There are multiple ways to represent a REST resource using Spring views and view resolvers with rendering model data as XML, JSON, Atom, and RSS.
- The type of model data view suits the client can be resolved via *ContentNegotiatingViewResolver*. The *ContentNegotiatingViewResolver*, however, does not resolve views itself but delegates to other *ViewResolvers*. By default, these other view resolvers are picked up automatically from the application context, though they can also be set explicitly by using the *viewResolver* property.
- Consuming REST resources using *RestTemplate*.

A Quick Example: Creating a Simple REST Endpoint

When working with REST services with Spring, we either publish application data as a REST service or access data in the application from third-party REST services. Here in this sample application, we combine Spring MVC to work with a REST endpoint in a controller named *EmployeeController*.

Firstly, we create a model class named *Employee*. This may be designated with JPA annotation to persist in the backend database. But, to keep it simple, we'll not use JPA; instead, we'll supply dummy data through the *EmployeeService* class. In a real situation, data is fetched from the backend database server and the data access methods are defined in a repository class. To give a hint, in our case, if we had used JPA with a back-end database, it may have been an interface that extends *CrudRepository*, something like this.

```
public interface EmployeeRepository extends  
  
    CrudRepository<Employee, String>{  
  
    // ...  
  
}
```

Employee.java

```
package  
com.mano.spring_mvc_rest_example.spring_mvc_rest.employee  
;  
  
public class Employee {  
  
    private String id;  
  
    private String name;  
  
    private String address;  
  
    public Employee() {  
  
    }  
  
}
```

```
public Employee(String id, String name, String
address) {

    this.id = id;

    this.name = name;

    this.address = address;

}

public String getId() {

    return id;

}

public void setId(String id) {

    this.id = id;

}

public String getName() {

    return name;

}

public void setName(String name) {

    this.name = name;

}

public String getAddress() {

    return address;

}

public void setAddress(String address) {
```

```
        this.address = address;

    }

}
```

EmployeeService.java

```
package com.mano.spring_mvc_rest_example.spring_

    mvc_rest.employee;

import org.springframework.stereotype.Service;

import java.util.Arrays;

import java.util.List;

@Service

public class EmployeeService {

    List<Employee> employeeList= Arrays.asList(

        new Employee("spiderman","Peter Parker",

            "New York"),

        new Employee("batman","Bruce Wayne",

            "Gotham City"),

        new Employee("superman","Clark Kent",

            "Metropolis"),

        new Employee("blackpanther","T'Challa",

            "Wakanda"),

        new Employee("ironman","Tony Stark",
```

```
        "New York")

    );

    public List<Employee> getEmployees() {

        return employeeList;

    }

    public Employee getEmployee(String id) {

        return employeeList.stream().filter(e->e.getId()

            .equals(id)).findFirst().get();

    }

    public void addEmployee(Employee employee) {

    }

    public void updateEmployee(Employee employee, String
id) {

        for(int i=0;i<employeeList.size();i++){

            Employee e=employeeList.get(i);

            if(e.getId().equals(id)) {

                employeeList.set(i, employee);

                break;

            }

        }

    }

    public void deleteEmployee(String id){
```

```
        employeeList.removeIf(e->e.getId().equals(id));  
  
    }  
  
}
```

EmployeeController.java

```
package  
com.mano.spring_mvc_rest_example.spring_mvc_rest.employee  
;  
  
import  
org.springframework.beans.factory.annotation.Autowired;  
  
import org.springframework.web.bind.annotation.*;  
  
import java.util.List;  
  
@RestController  
  
public class EmployeeController {  
  
    @Autowired  
  
    private EmployeeService employeeService;  
  
    @RequestMapping("/employees")  
  
    public List<Employee> getEmployees() {  
  
        return employeeService.getEmployees();  
  
    }  
  
    @RequestMapping("/employees/{empid}")  
  
    public Employee getEmployee(@PathVariable("empid")  
  
        String id) {  
  
        return employeeService.getEmployee(id);  
  
    }  
  
}
```

```
}

@RequestMapping(method= RequestMethod.POST,

    value="/employees")

    public void addEmployee(@RequestBody Employee
employee) {

        employeeService.addEmployee (employee) ;

    }

@RequestMapping(method = RequestMethod.PUT,

    value="/employees/{id}")

    public void updateEmployee(@RequestBody Employee
employee,

        @PathVariable String id){

        employeeService.updateEmployee (employee, id);

    }

@RequestMapping(method = RequestMethod.DELETE,

    value="/employees/{id}")

    public void deleteEmployee(@PathVariable String id){

        employeeService>.deleteEmployee (id) ;

    }

}
```

Observe that the Web controller class named `EmployeeController` is designated as a `@RestController` annotation. This is a convenience annotation that actually combines the `@Controller` and `@ResponseBody` annotations.

The `@Controller` annotation designates a POJO as a Web controller and is a specialization of `@Component`. When we designate a POJO class with `@Controller` or `@Component`, or even a `@RestController`, Spring auto detects them by considering

them as a candidate while class path scanning. The `@ResponseBody` annotation indicates that the method response value should be bound to the Web response body.

The valid URL requests for publishing REST resources for the above code are as follows:

- Get all employees: <http://localhost:8080/employees>
- Get one employee: <http://localhost:8080/employees/batman>

Conclusion

For REST CRUD operations such as adding, updating, and deleting *Employee*, we need a HTTP client application that enables us to test Web services, such as [postman](#); otherwise, we need to implement the view layer of the application with the help of JavaScript frameworks such as [jQuery](#), [AngularJS](#), and the like. To keep the write-up well within limits, we have not implemented them here. If possible, we'll take them up in a separate write-up. By the way, we have only scratched the surface of Spring MVC and Spring REST support. Take this as a warm-up before the deep plunge you may want to take into the stream of Spring. As you swim across, you'll find many interesting sight scenes. 😊

<https://www.developer.com/java/exploring-rest-apis-with-spring-mvc/>

Building an application using Maven

Lifecycle Management

One of the primary objectives of Maven is to manage the lifecycle of a Java project. While building a Java application may appear to be a simple, one-step process, there are actually multiple steps that take place. Maven divides this process into three **lifecycles**:

1. **clean**: Prepares the project for building by removing unneeded files and dependencies
2. **default**: Builds the project
3. **site**: Creates project documentation

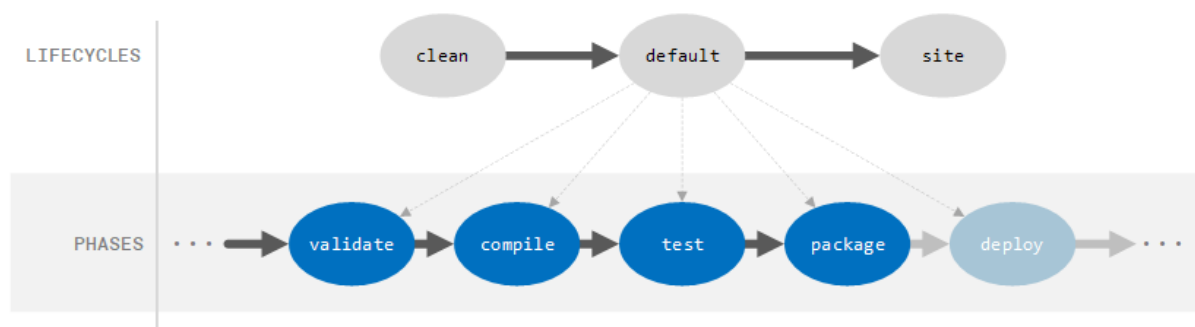
Phases

Maven further subdivides these lifecycles into **phases**, which represent a stage in the build process. For example, the **default** lifecycle includes the following phases (as well as others):

1. **validate**
2. **compile**
3. **test**
4. **package**
5. **deploy**

In the same way as a deployment pipeline (pp. 103 of [Continuous Delivery](#)) granularizes the stages of deployment into discrete steps, Maven also divides its build process into distinct phases. These phases create a chain, where the execution of a later phase executes dependent phases.

For example, if we wish to package an application through a Maven build, our application must first be validated, compiled, and then tested before Maven can generate the resulting package. Thus, when executing the **package** phase of a build, Maven will first execute the **validate**, **compile**, and **test** phases of the build before finally executing the **package** phase. Maven phases, therefore, act as a sequence of ordered steps.



We can execute phases by supplying them as command-line arguments to the **mvn** command:

```
mvn package
```

1

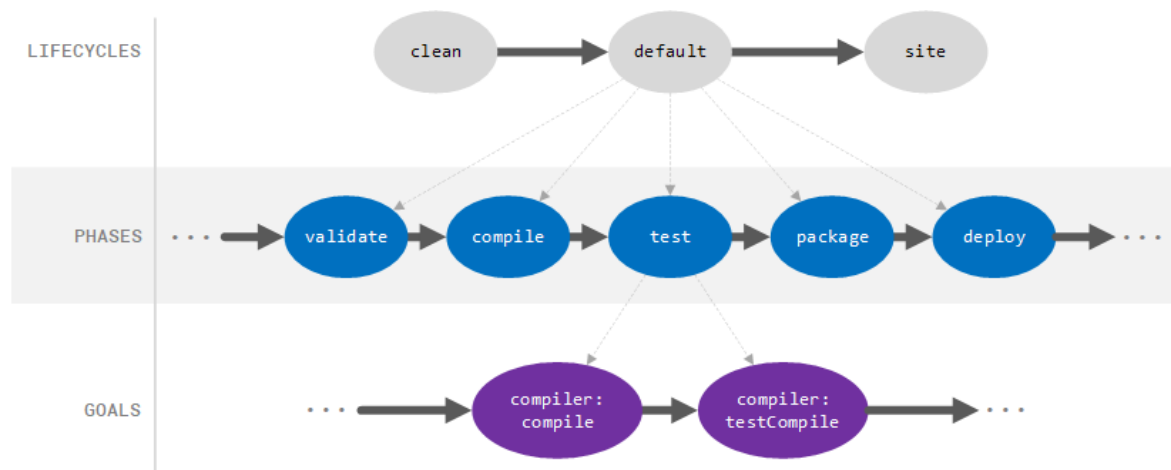
Goals & Plugins

Maven breaks phases down one more time into **goals**, which represent discrete tasks that are executed as part of each phase. For example, when we execute the `compile` phase in a Maven build, we are actually compiling both the main sources that make up our project as well as the test sources that will be used when executing our automated test cases.

Thus, the `compile` phase is composed of two goals:

1. `compiler:compile`
2. `compiler:testCompile`

The `compiler` portion of the goal is the plugin name. A Maven **plugin** is an artifact that supplies Maven goals. The addition of these plugins allows Maven to be extended beyond its basic functionality.



For example, suppose that we wish to add a goal that verifies that our code meets the formatting standard of our company. To do this, we could create a new plugin that has a goal that checks the source code and compares it to our company standard, succeeding if our code meets the standard and failing otherwise.

We can then tie this goal into the `validate` phase so that when Maven runs the `validate` phase (such as when the `compile` phase is run), our custom goal is executed. Creating such a plugin is outside the scope of this article, but detailed information can be found in the official Maven [Plugin Development](#) documentation.

Note that a goal may be associated with zero or more phases. If no phase is associated with the goal, the goal will not be included in a build by default but can be explicitly executed. For example, if we create a

goal `foo:bar` that is not associated with any phase, Maven will not execute this goal for us (since no dependency is created to a phase that Maven is executing), but we can explicitly instruct Maven to execute this goal on the command line:

```
mvn foo:bar
```

1

Likewise, a phase can have zero or more goals associated with it. If a phase does not have any goals associated with it, though, it will not be executed by Maven.

For a complete list of all phases and goals included in Maven by default, see the official Maven [Introduction to the Build Lifecycle](#) documentation.

<https://dzone.com/articles/building-java-applications-with-maven>

UNIT - V

Databases & Deployment: Relational schemas and normalization
Structured Query Language (SQL) Data persistence using Spring JDBC Agile development principles and deploying application in Cloud

Structured Query Language (SQL)

Structured Query Language is a standard Database language which is used to create, maintain and retrieve the relational database. Following are some interesting facts about SQL.

- SQL is case insensitive. But it is a recommended practice to use keywords (like SELECT, UPDATE, CREATE, etc) in capital letters and use user defined things (like table name, column name, etc) in small letters.
- We can write comments in SQL using “--” (double hyphen) at the beginning of any line.
- SQL is the programming language for relational databases (explained below) like MySQL, Oracle, Sybase, SQL Server, Postgre, etc. Other non-relational databases (also called NoSQL) databases like MongoDB, DynamoDB, etc do not use SQL
- Although there is an ISO standard for SQL, most of the implementations slightly vary in syntax. So we may encounter queries that work in SQL Server but do not work in MySQL.

What is Relational Database?

Relational database means the data is stored as well as retrieved in the form of relations (tables). Table 1 shows the relational database with only one relation called **STUDENT** which stores **ROLL_NO**, **NAME**, **ADDRESS**, **PHONE** and **AGE** of students.

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18

2	RAMESH	GURGAON	9652431543	18
3	SUJIT	ROHTAK	9156253131	20
4	SURESH	DELHI	9156768971	18

TABLE 1

These are some important terminologies that are used in terms of relation.

Attribute: Attributes are the properties that define a relation. e.g.; ROLL_NO, NAME etc.

Tuple: Each row in the relation is known as tuple. The above relation contains 4 tuples, one of which is shown as:

1	RAM	DELHI	9455123451	18
---	-----	-------	------------	----

Degree: The number of attributes in the relation is known as degree of the relation. The STUDENT relation defined above has degree 5.

Cardinality: The number of tuples in a relation is known as cardinality. The STUDENT relation defined above has cardinality 4.

Column: Column represents the set of values for a particular attribute. The column ROLL_NO is extracted from relation STUDENT.

ROLL_NO
1
2
3
4

The queries to deal with relational database can be categories as:

Data Definition Language: It is used to define the structure of the database. e.g; CREATE TABLE, ADD COLUMN, DROP COLUMN and so on.

Data Manipulation Language: It is used to manipulate data in the relations. e.g.; INSERT, DELETE, UPDATE and so on.

Data Query Language: It is used to extract the data from the relations. e.g.; SELECT

So first we will consider the Data Query Language. A generic query to retrieve from a relational database is:

```
SELECT [DISTINCT] Attribute_List FROM R1,R2....RM
[WHERE condition]
[GROUP BY (Attributes)][HAVING condition]]
[ORDER BY(Attributes)[DESC]];
```

Part of the query represented by statement 1 is compulsory if you want to retrieve from a relational database. The statements written inside [] are optional. We will look at the possible query combination on relation shown in Table 1.

Case 1: If we want to retrieve attributes **ROLL_NO** and **NAME** of all

students, the query will be:

```
SELECT ROLL_NO, NAME FROM STUDENT;
```

ROLL_NO	NAME
1	RAM
2	RAMESH
3	SUJIT
4	SURESH

Case 2: If we want to retrieve **ROLL_NO** and **NAME** of the students whose **ROLL_NO** is greater than 2, the query will be:

```
SELECT ROLL_NO, NAME FROM STUDENT  
WHERE ROLL_NO>2;
```

ROLL_NO	NAME
3	SUJIT
4	SURESH

CASE 3: If we want to retrieve all attributes of students, we can write * in place of writing all attributes as:

```
SELECT * FROM STUDENT  
WHERE ROLL_NO>2;
```

ROLL_NO	NAME	ADDRESS	PHONE	AGE
3	SUJIT	ROHTAK	9E+09	20
4	SURESH	DELHI	9E+09	18

CASE 4: If we want to represent the relation in ascending order by **AGE**, we can use ORDER BY clause as:

```
SELECT * FROM STUDENT ORDER BY AGE;
```

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
2	RAMESH	GURGAON	9652431543	18

4	SURESH	DELHI	9156768971	18
3	SUJIT	ROHTAK	9156253131	20

Note: ORDER BY **AGE** is equivalent to ORDER BY **AGE** ASC. If we want to retrieve the results in descending order of **AGE**, we can use ORDER BY **AGE** DESC.

CASE 5: If we want to retrieve distinct values of an attribute or group of attribute, DISTINCT is used as in:

SELECT DISTINCT ADDRESS FROM STUDENT;

ADDRESS
DELHI
GURGAON
ROHTAK

If DISTINCT is not used, DELHI will be repeated twice in result set. Before understanding GROUP BY and HAVING, we need to understand aggregations functions in SQL.

AGGRATION FUNCTIONS: Aggregation functions are used to perform mathematical operations on data values of a relation. Some of the common aggregation functions used in SQL are:

1. **COUNT:** Count function is used to count the number of rows in a relation.
e.g; SELECT COUNT (PHONE) FROM STUDENT;

COUNT(PHONE)
4

2. **SUM:** SUM function is used to add the values of an attribute in a relation.
e.g; SELECT SUM (AGE) FROM STUDENT;

SUM(AGE)
74

In the same way, MIN, MAX and AVG can be used. As we have seen above, all aggregation functions return only 1 row.

AVERAGE: It gives the average values of the tuples. It is also defined as sum divided by count values.

Syntax:AVG(attribute name)

OR

Syntax: SUM(attributename)/COUNT(attributename)

The above mentioned syntax also retrieves the average value of tuples.

MAXIMUM: It extracts the maximum value among the set of tuples.

Syntax: MAX(attributename)

MINIMUM: It extracts the minimum value amongst the set of all the tuples.

Syntax: MIN(attributename)

GROUP BY: Group by is used to group the tuples of a relation based on an attribute or group of attribute. It is always combined with aggregation function which is computed on group. e.g.;

SELECT ADDRESS, SUM(AGE) FROM STUDENT

GROUP BY (ADDRESS);

In this query, SUM(**AGE**) will be computed but not for entire table but for each address. i.e.; sum of AGE for address DELHI(18+18=36) and similarly for other address as well. The output is:

ADDRESS	SUM(AGE)
DELHI	36
GURGAON	18
ROHTAK	20

If we try to execute the query given below, it will result in error because although we have computed SUM(AGE) for each address, there are more than 1 ROLL_NO for each address we have grouped. So it can't be displayed in result set. We need to use aggregate functions on columns after SELECT statement to make sense of the resulting set whenever we are using GROUP BY.

**SELECT ROLL_NO, ADDRESS, SUM(AGE) FROM STUDENT
GROUP BY (ADDRESS);**

What is Data Normalization and Why Is It Important?

Normalization is the process of reducing data redundancy in a table and improving data integrity. Then why do you need it? If there is no normalization in SQL, there will be many problems, such as:

- **Insert Anomaly:** This happens when we cannot insert data into the table without another.
- **Update Anomaly:** This is due to data inconsistency caused by data redundancy and data update.
- **Delete exception:** Occurs when some attributes are lost due to the deletion of other attributes.

So [normalization](#) is a way of organizing data in a database. Normalization involves organizing the columns and tables in the database to ensure that their dependencies are correctly implemented using database constraints. Normalization is the process of organizing data in a proper manner. It is used to minimize the duplication of various relationships in the database. It is also used to troubleshoot exceptions such as inserts, deletes, and updates in the table. It helps to split a large table into several small normalized tables. Relational links and links are used to reduce redundancy. Normalization, also known as database normalization or data normalization, is an important part of relational database design because it helps to improve the speed, accuracy, and efficiency of the database.

Now the is a question arises: What is the relationship between SQL and normalization? Well, SQL is the language used to interact with the database. Normalization in SQL improves data distribution. In order to initiate interaction, the data in the database must be normalized. Otherwise, we cannot continue because it will cause an exception. Normalization can also make it easier to design the database to have the best structure for atomic elements (that is, elements that cannot be broken down into smaller parts). Usually, we break large tables into small tables to improve efficiency. Edgar F. Codd defined the first paradigm in 1970, and finally other paradigms. When normalizing a database, organize data into tables and columns. Make sure that each table contains only relevant data. If the data is not directly related, create a new table for that data. Normalization is necessary to ensure that the table only contains data directly related to the primary key, each data field contains only one data element, and to remove redundant (duplicated and unnecessary) data.

DDL

DDL is Data Definition Language and is used to define the structures like schema, database, tables, constraints etc. Examples of DDL are create and alter statements.

DML

DML is Data Manipulation Language and is used to manipulate data. Examples of DML are insert, update and delete statements.

Following are the important differences between DDL and DML.

Sr. No.	Key	DDL	DML
1	Stands for	DDL stands for Data Definition Language.	DML stands for Data Manipulation Language.
2	Usage	DDL statements are used to create database, schema, constraints, users, tables etc.	DML statement is used to insert, update or delete the records.
3	Classification	DDL has no further classification.	DML is further classified into procedural DML and non-procedural DML.
4	Commands	CREATE, DROP, RENAME and ALTER.	INSERT, UPDATE and DELETE.

Transaction Control language is a language that manages transactions within the database. It is used to execute the changes made by the DML statements.

TCL Commands

Transaction Control Language (TCL) Commands are:

- **Commit** – It is used to save the transactions in the database.
- **Rollback** – It is used to restore the database to that state which was last committed.
- **Begin** – It is used at the beginning of a transaction.
- **Savepoint** – The changes done till savepoint will be unchanged and all the transactions after savepoint will be rolled back.

Example

Given below is an example of the usage of the TCL commands in the database management system (DBMS) –

```
BEGIN TRANSACTION
UPDATE employees
SET empname='bob'
WHERE empid='001'

UPDATE employees
SET empname = 'bob'
WHERE city='hyderabad'

IF @@ROWCOUNT=5
```

```

COMMIT TRANSACTION
ELSE
ROLLBACK TRANSACTION

```

In the above example after we begin the transaction, we are trying to update the employee's name with some value of id. If we affect five rows with our first query then, it will COMMIT transaction else It will be ROLLBACK.

Difference between Commit, rollback and savepoint of TCL commands

Sno.	Rollback	Commit	Savepoint
1.	Rollback means the database is restored to the last committed state	DML commands saves modification and it permanently saves the transaction.	Savepoint helps to save the transaction temporarily.
2.	Syntax- ROLLBACK [To SAVEPOINT_NAME];	Syntax- COMMIT;	Syntax- SAVEPOINT [savepoint_name;]
3.	Example- ROLLBACK Update5;	Example- SQL> COMMIT;	Example- SAVEPOINT table_create;

SPRING JDBC EXAMPLE

To understand the concepts related to Spring JDBC framework with JdbcTemplate class, let us write a simple example which will implement all the CRUD operations on the following Student table.

```

CREATE TABLE Student(
  ID INT NOT NULL AUTO_INCREMENT,
  NAME VARCHAR(20) NOT NULL,
  AGE INT NOT NULL,
  PRIMARY KEY (ID)
);

```

Steps to Create a JDBC Spring Application:

Step Description

- 1 Create a project with a name *SpringExample* and create a package *com.mrcet* under the **src** folder in the created project.
- 2 Add required Spring libraries using *Add External JARs*.
- 3 Add Spring JDBC specific latest libraries **mysql-connector-java.jar**, **org.springframework.jdbc.jar** and **org.springframework.transaction.jar** in the project. You can download required libraries if you do not have them already.
- 4 Create DAO interface *StudentDAO* and list down all the required methods.
- 5 Create other required Java classes *Student*, *StudentMapper*, *StudentJdbcTemplate* and *MainApp* under the *com.mrcet* package.

- 6 Make sure you already created **Student** table in TEST database. Also make sure your MySQL server is working fine and you have read/write access on the database using the give username and password.
- 7 Create Beans configuration file *Beans.xml* under the **src** folder.
- 8 The final step is to create the content of all the Java files and Bean Configuration file and run the application as explained below.

1. Spring JDBC - insert Query

The following example will demonstrate how to create a query using Insert query with the help of Spring JDBC. We'll insert a few records in Student Table.

Syntax

```
String insertQuery = "insert into Student (name, age) values (?, ?)";  
jdbcTemplateObject.update( insertQuery, name, age);
```

Where,

- **insertQuery** – Insert query having placeholders.
- **jdbcTemplateObject** – StudentJdbcTemplate object to insert student object in database.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will insert a query. To write our example use the following steps to create a Spring application.

Following is the content of the Data Access Object interface file **StudentDAO.java**.

```
package com.mrcet;  
  
import java.util.List;  
import javax.sql.DataSource;  
  
public interface StudentDAO {  
    /**  
     * This is the method to be used to initialize  
     * database resources ie. connection.  
     */  
    public void setDataSource(DataSource ds);  
  
    /**  
     * This is the method to be used to create  
     * a record in the Student table.  
     */  
    public void create(String name, Integer age);  
  
    /**  
     * This is the method to be used to list down  
     * all the records from the Student table.  
     */  
}
```

```
public List<Student> listStudents();  
}
```

Following is the content of the **Student.java** file.

```
package com.mrcet;  
  
public class Student {  
    private Integer age;  
    private String name;  
    private Integer id;  
  
    public void setAge(Integer age) {  
        this.age = age;  
    }  
    public Integer getAge() {  
        return age;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setId(Integer id) {  
        this.id = id;  
    }  
    public Integer getId() {  
        return id;  
    }  
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.mrcet;  
  
import java.sql.ResultSet;  
import java.sql.SQLException;  
import org.springframework.jdbc.core.RowMapper;  
  
public class StudentMapper implements RowMapper<Student> {  
    public Student mapRow(ResultSet rs, int rowNum) throws SQLException {  
        Student student = new Student();  
        student.setId(rs.getInt("id"));  
        student.setName(rs.getString("name"));  
        student.setAge(rs.getInt("age"));  
        return student;  
    }  
}
```

Following is the implementation class file **StudentJdbcTemplate.java** for the defined DAO interface StudentDAO.

```
package com.mrcet;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class StudentJdbcTemplate implements StudentDAO {
    private DataSource dataSource;
    private JdbcTemplate jdbcTemplateObject;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
        this.jdbcTemplateObject = new JdbcTemplate(dataSource);
    }

    public void create(String name, Integer age) {
        String insertQuery = "insert into Student (name, age) values (?, ?)";
        jdbcTemplateObject.update( insertQuery, name, age);
        System.out.println("Created Record Name = " + name + " Age = " + age);
        return;
    }

    public List<Student> listStudents() {
        String SQL = "select * from Student";
        List<Student> students = jdbcTemplateObject.query(SQL, new StudentMapper());
        return students;
    }
}
```

Following is the content of the **MainApp.java** file.

```
package com.mrcet;

import java.util.List;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.mrcet.StudentJdbcTemplate;

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");

        StudentJdbcTemplate studentJdbcTemplate =
            (StudentJdbcTemplate)context.getBean("studentJdbcTemplate");

        System.out.println("-----Records Creation-----" );
        studentJdbcTemplate.create("Zara", 11);
        studentJdbcTemplate.create("Nuha", 2);
        studentJdbcTemplate.create("Ayan", 15);

        System.out.println("-----Listing Multiple Records-----" );
        List<Student> students = studentJdbcTemplate.listStudents();
    }
}
```

```
for (Student record : students) {  
    System.out.print("ID : " + record.getId() );  
    System.out.print(", Name : " + record.getName() );  
    System.out.println(", Age : " + record.getAge());  
}  
}
```

Following is the configuration file **Beans.xml**.

```
<?xml version = "1.0" encoding = "UTF-8"?>  
<beans xmlns = "http://www.springframework.org/schema/beans"  
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation = "http://www.springframework.org/schema/beans  
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">  
  
    <!-- Initialization for data source -->  
    <bean id = "dataSource"  
        class = "org.springframework.jdbc.datasource.DriverManagerDataSource">  
        <property name = "driverClassName" value = "com.mysql.cj.jdbc.Driver"/>  
        <property name = "url" value = "jdbc:mysql://localhost:3306/TEST"/>  
        <property name = "username" value = "root"/>  
        <property name = "password" value = "admin"/>  
    </bean>  
  
    <!-- Definition for studentJdbcTemplate bean -->  
    <bean id = "studentJdbcTemplate"  
        class = "com.mrcet.StudentJdbcTemplate">  
        <property name = "dataSource" ref = "dataSource" />  
    </bean>  
</beans>
```

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message.

```
-----Records Creation-----  
Created Record Name = Zara Age = 11  
Created Record Name = Nuha Age = 2  
Created Record Name = Ayan Age = 15  
-----Listing Multiple Records-----  
ID : 1, Name : Zara, Age : 11  
ID : 2, Name : Nuha, Age : 2  
ID : 3, Name : Ayan, Age : 15
```

2. Spring JDBC - Select Query

Following example will demonstrate how to read a query using Spring JDBC. We'll read available records in Student Table.

Syntax

```
String selectQuery = "select * from Student";  
List <Student> students = jdbcTemplateObject.query(selectQuery, new StudentMapper());
```

Where,

- **selectQuery** – Select query to read students.
- **jdbcTemplateObject** – StudentJdbcTemplate object to read student object from database.
- **StudentMapper** – StudentMapper is a RowMapper object to map each fetched record to student object.

To understand above mentioned concepts related to Spring JDBC, let us write an example which will select a query. To write our example, use the following steps to create a Spring application.

Following is the content of the Data Access Object interface file **StudentDAO.java**.

```
package com.mrcet;  
  
import java.util.List;  
import javax.sql.DataSource;  
  
public interface StudentDAO {  
    /**  
     * This is the method to be used to initialize  
     * database resources ie. connection.  
     */  
    public void setDataSource(DataSource ds);  
  
    /**  
     * This is the method to be used to list down  
     * all the records from the Student table.  
     */  
    public List<Student> listStudents();  
}
```

Following is the content of the **Student.java** file.

```
package com.mrcet;  
  
public class Student {  
    private Integer age;  
    private String name;  
    private Integer id;  
  
    public void setAge(Integer age) {  
        this.age = age;  
    }  
    public Integer getAge() {  
        return age;  
    }  
}
```



```
public void setName(String name) {
    this.name = name;
}
public String getName() {
    return name;
}
public void setId(Integer id) {
    this.id = id;
}
public Integer getId() {
    return id;
}
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.mrcet;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;

public class StudentMapper implements RowMapper<Student> {
    public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
        Student student = new Student();
        student.setId(rs.getInt("id"));
        student.setName(rs.getString("name"));
        student.setAge(rs.getInt("age"));
        return student;
    }
}
```

Following is the implementation class file **StudentJdbcTemplate.java** for the defined DAO interface StudentDAO.

```
package com.mrcet;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class StudentJdbcTemplate implements StudentDAO {
    private DataSource dataSource;
    private JdbcTemplate jdbcTemplateObject;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
        this.jdbcTemplateObject = new JdbcTemplate(dataSource);
    }

    public List<Student> listStudents() {
        String SQL = "select * from Student";
        List<Student> students = jdbcTemplateObject.query(SQL, new StudentMapper());
        return students;
    }
}
```

```
}  
}
```

Following is the content of the **MainApp.java** file.

```
package com.mrcet;  
  
import java.util.List;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
import com.mrcet.StudentJdbcTemplate;  
  
public class MainApp {  
    public static void main(String[] args) {  
        ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");  
        StudentJdbcTemplate studentJdbcTemplate =  
(StudentJdbcTemplate)context.getBean("studentJdbcTemplate");  
  
        System.out.println("-----Listing Multiple Records-----" );  
        List<Student> students = studentJdbcTemplate.listStudents();  
  
        for (Student record : students) {  
            System.out.print("ID : " + record.getId() );  
            System.out.print(", Name : " + record.getName() );  
            System.out.println(", Age : " + record.getAge());  
        }  
    }  
}
```

Following is the configuration file **Beans.xml**.

```
<?xml version = "1.0" encoding = "UTF-8"?>  
<beans xmlns = "http://www.springframework.org/schema/beans"  
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation = "http://www.springframework.org/schema/beans  
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">  
  
    <!-- Initialization for data source -->  
    <bean id = "dataSource"  
        class = "org.springframework.jdbc.datasource.DriverManagerDataSource">  
        <property name = "driverClassName" value = "com.mysql.cj.jdbc.Driver"/>  
        <property name = "url" value = "jdbc:mysql://localhost:3306/TEST"/>  
        <property name = "username" value = "root"/>  
        <property name = "password" value = "admin"/>  
    </bean>  
  
    <!-- Definition for studentJdbcTemplate bean -->  
    <bean id="studentJdbcTemplate"  
        class = "com.mrcet.StudentJdbcTemplate">  
        <property name = "dataSource" ref = "dataSource" />  
    </bean>  
</beans>
```

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message.

-----Listing Multiple Records-----

ID : 1, Name : Zara, Age : 11

ID : 2, Name : Nuha, Age : 2

ID : 3, Name : Ayan, Age : 15

3. Spring JDBC - Update Query

Following example will demonstrate how to update a query using Spring JDBC. We'll update the available records in Student Table.

Syntax

```
String updateQuery = "update Student set age = ? where id = ?";  
jdbcTemplateObject.update(updateQuery, age, id);
```

Where,

- **updateQuery** – Update query to update student with place holders.
- **jdbcTemplateObject** – StudentJdbcTemplate object to update student object in the database.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example which will update a query. To write our example, let us have a working Eclipse IDE in place and use the following steps to create a Spring application.

Following is the content of the Data Access Object interface file **StudentDAO.java**.

```
package com.mrcet;  
  
import java.util.List;  
import javax.sql.DataSource;  
  
public interface StudentDAO {  
    /**  
     * This is the method to be used to initialize  
     * database resources ie. connection.  
     */  
    public void setDataSource(DataSource ds);  
  
    /**  
     * This is the method to be used to update  
     * a record into the Student table.  
     */  
    public void update(Integer id, Integer age);  
  
    /**  
     * This is the method to be used to list down  
     * a record from the Student table corresponding  
     * to a passed student id.  
     */  
}
```

```
public Student getStudent(Integer id);  
}
```

Following is the content of the **Student.java** file.

```
package com.mrcet;  
  
public class Student {  
    private Integer age;  
    private String name;  
    private Integer id;  
  
    public void setAge(Integer age) {  
        this.age = age;  
    }  
    public Integer getAge() {  
        return age;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setId(Integer id) {  
        this.id = id;  
    }  
    public Integer getId() {  
        return id;  
    }  
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.mrcet;  
  
import java.sql.ResultSet;  
import java.sql.SQLException;  
import org.springframework.jdbc.core.RowMapper;  
  
public class StudentMapper implements RowMapper<Student> {  
    public Student mapRow(ResultSet rs, int rowNum) throws SQLException {  
        Student student = new Student();  
        student.setId(rs.getInt("id"));  
        student.setName(rs.getString("name"));  
        student.setAge(rs.getInt("age"));  
        return student;  
    }  
}
```

Following is the implementation class file **StudentJdbcTemplate.java** for the defined DAO interface StudentDAO.

```
package com.mrcet;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class StudentJdbcTemplate implements StudentDAO {
    private DataSource dataSource;
    private JdbcTemplate jdbcTemplateObject;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
        this.jdbcTemplateObject = new JdbcTemplate(dataSource);
    }

    public void update(Integer id, Integer age){
        String SQL = "update Student set age = ? where id = ?";
        jdbcTemplateObject.update(SQL, age, id);
        System.out.println("Updated Record with ID = " + id );
        return;
    }

    public Student getStudent(Integer id) {
        String SQL = "select * from Student where id = ?";
        Student student = jdbcTemplateObject.queryForObject(
            SQL, new Object[]{id}, new StudentMapper()
        );
        return student;
    }
}
```

Following is the content of the **MainApp.java** file.

```
package com.mrcet;

import java.util.List;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.mrcet.StudentJdbcTemplate;

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
        StudentJdbcTemplate studentJdbcTemplate =
            (StudentJdbcTemplate)context.getBean("studentJdbcTemplate");

        System.out.println("----Updating Record with ID = 2 ----" );
        studentJdbcTemplate.update(2, 20);

        System.out.println("----Listing Record with ID = 2 ----" );
        Student student = studentJdbcTemplate.getStudent(2);
        System.out.print("ID : " + student.getId() );
        System.out.print(", Name : " + student.getName() );
        System.out.println(", Age : " + student.getAge());
    }
}
```

```
}  
}
```

Following is the configuration file **Beans.xml**.

```
<?xml version = "1.0" encoding = "UTF-8"?>  
<beans xmlns = "http://www.springframework.org/schema/beans"  
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"  
  xsi:schemaLocation = "http://www.springframework.org/schema/beans  
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">  
  
  <!-- Initialization for data source -->  
  <bean id = "dataSource"  
    class = "org.springframework.jdbc.datasource.DriverManagerDataSource">  
    <property name = "driverClassName" value = "com.mysql.cj.jdbc.Driver"/>  
    <property name = "url" value = "jdbc:mysql://localhost:3306/TEST"/>  
    <property name = "username" value = "root"/>  
    <property name = "password" value = "admin"/>  
  </bean>  
  
  <!-- Definition for studentJdbcTemplate bean -->  
  <bean id = "studentJdbcTemplate"  
    class = "com.mrcet.StudentJdbcTemplate">  
    <property name = "dataSource" ref = "dataSource" />  
  </bean>  
  
</beans>
```

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message.

```
----Updating Record with ID = 2 ----  
Updated Record with ID = 2  
----Listing Record with ID = 2 ----  
ID : 2, Name : Nuha, Age : 20
```

4. Spring JDBC - Delete Query

The following example will demonstrate how to delete a query using Spring JDBC. We'll delete one of the available records in Student Table

Syntax

```
String deleteQuery = "delete from Student where id = ?";  
jdbcTemplateObject.update(deleteQuery, id);
```

Where,

- **deleteQuery** – Delete query to delete student with placeholders.
- **jdbcTemplateObject** – StudentJdbcTemplate object to delete student object in the database.

To understand the above-mentioned concepts related to Spring JDBC, let us write an example

which will delete a query. To write our example, use the following steps to create a Spring application.

Following is the content of the Data Access Object interface file **StudentDAO.java**.

```
package com.mrcet;

import java.util.List;
import javax.sql.DataSource;

public interface StudentDAO {
    /**
     * This is the method to be used to initialize
     * database resources ie. connection.
     */
    public void setDataSource(DataSource ds);

    /**
     * This is the method to be used to list down
     * all the records from the Student table.
     */
    public List<Student> listStudents();

    /**
     * This is the method to be used to delete
     * a record from the Student table corresponding
     * to a passed student id.
     */
    public void delete(Integer id);
}
```

Following is the content of the **Student.java** file.

```
package com.mrcet;

public class Student {
    private Integer age;
    private String name;
    private Integer id;

    public void setAge(Integer age) {
        this.age = age;
    }
    public Integer getAge() {
        return age;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setId(Integer id) {
```

```
    this.id = id;
}
public Integer getId() {
    return id;
}
}
```

Following is the content of the **StudentMapper.java** file.

```
package com.mrcet;

import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;

public class StudentMapper implements RowMapper<Student> {
    public Student mapRow(ResultSet rs, int rowNum) throws SQLException {
        Student student = new Student();
        student.setId(rs.getInt("id"));
        student.setName(rs.getString("name"));
        student.setAge(rs.getInt("age"));
        return student;
    }
}
```

Following is the implementation class file **StudentJdbcTemplate.java** for the defined DAO interface StudentDAO.

```
package com.mrcet;

import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class StudentJdbcTemplate implements StudentDAO {
    private DataSource dataSource;
    private JdbcTemplate jdbcTemplateObject;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
        this.jdbcTemplateObject = new JdbcTemplate(dataSource);
    }

    public List<Student> listStudents() {
        String SQL = "select * from Student";
        List<Student> students = jdbcTemplateObject.query(SQL, new StudentMapper());
        return students;
    }

    public void delete(Integer id){
        String SQL = "delete from Student where id = ?";
        jdbcTemplateObject.update(SQL, id);
        System.out.println("Deleted Record with ID = " + id );
        return;
    }
}
```



```
}
```

Following is the content of the **MainApp.java** file.

```
package com.mrcet;

import java.util.List;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.mrcet.StudentJdbcTemplate;

public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("Beans.xml");
        StudentJdbcTemplate studentJdbcTemplate =
            (StudentJdbcTemplate)context.getBean("studentJdbcTemplate");

        System.out.println("----Delete Record with ID = 2 ----" );
        studentJdbcTemplate.delete(2);

        System.out.println("-----Listing Multiple Records-----" );
        List<Student> students = studentJdbcTemplate.listStudents();

        for (Student record : students) {
            System.out.print("ID : " + record.getId() );
            System.out.print(", Name : " + record.getName() );
            System.out.println(", Age : " + record.getAge());
        }
    }
}
```

Following is the configuration file **Beans.xml**.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation = "http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">

    <!-- Initialization for data source -->
    <bean id = "dataSource"
        class = "org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name = "driverClassName" value = "com.mysql.cj.jdbc.Driver"/>
        <property name = "url" value = "jdbc:mysql://localhost:3306/TEST"/>
        <property name = "username" value = "root"/>
        <property name = "password" value = "admin"/>
    </bean>

    <!-- Definition for studentJdbcTemplate bean -->
    <bean id = "studentJdbcTemplate"
        class = "com.mrcet.StudentJdbcTemplate">
        <property name = "dataSource" ref = "dataSource" />
    </bean>
```

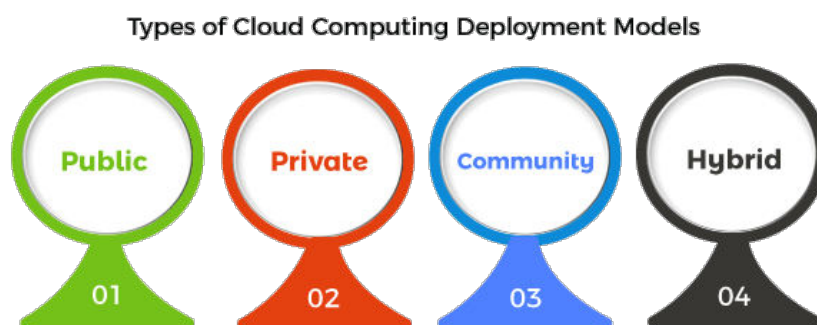
</beans>

Once you are done creating the source and bean configuration files, let us run the application. If everything is fine with your application, it will print the following message.

```
----Updating Record with ID = 2 ----  
Updated Record with ID = 2  
----Listing Record with ID = 2 ----  
ID : 2, Name : Nuha, Age : 20
```

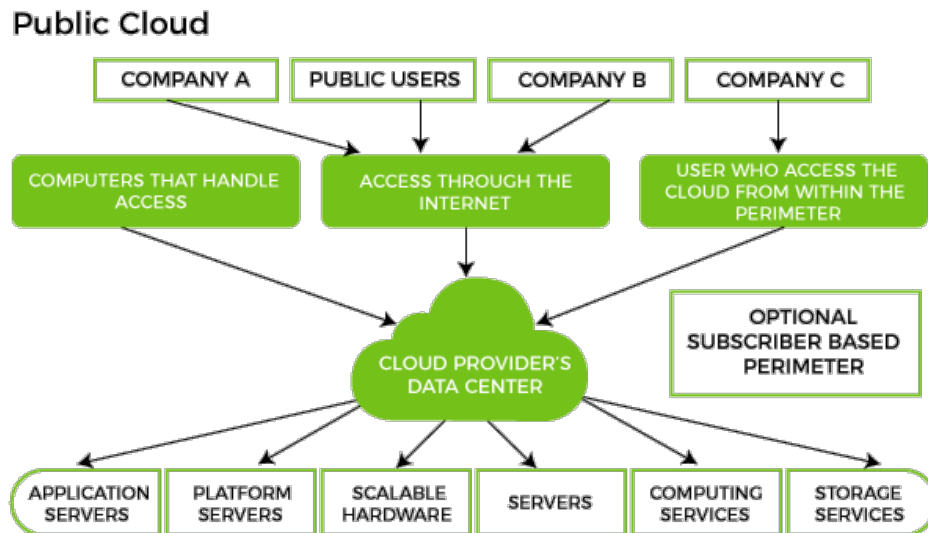
Different Types of Cloud Computing Deployment Models

Most cloud hubs have tens of thousands of servers and storage devices to enable fast loading. It is often possible to choose a geographic area to put the data "closer" to users. Thus, deployment models for cloud computing are categorized based on their location. To know which model would best fit the requirements of your organization, let us first learn about the various types.



Public Cloud

The name says it all. It is accessible to the public. Public deployment models in the cloud are perfect for organizations with growing and fluctuating demands. It also makes a great choice for companies with low-security concerns. Thus, you pay a cloud service provider for networking services, compute virtualization & storage available on the public internet. It is also a great delivery model for the teams with development and testing. Its configuration and deployment are quick and easy, making it an ideal choice for test environments.



Benefits of Public Cloud

- Minimal Investment - As a pay-per-use service, there is no large upfront cost and is ideal for businesses who need quick access to resources
- No Hardware Setup - The cloud service providers fully fund the entire Infrastructure
- No Infrastructure Management - This does not require an in-house team to utilize the public cloud.

Limitations of Public Cloud

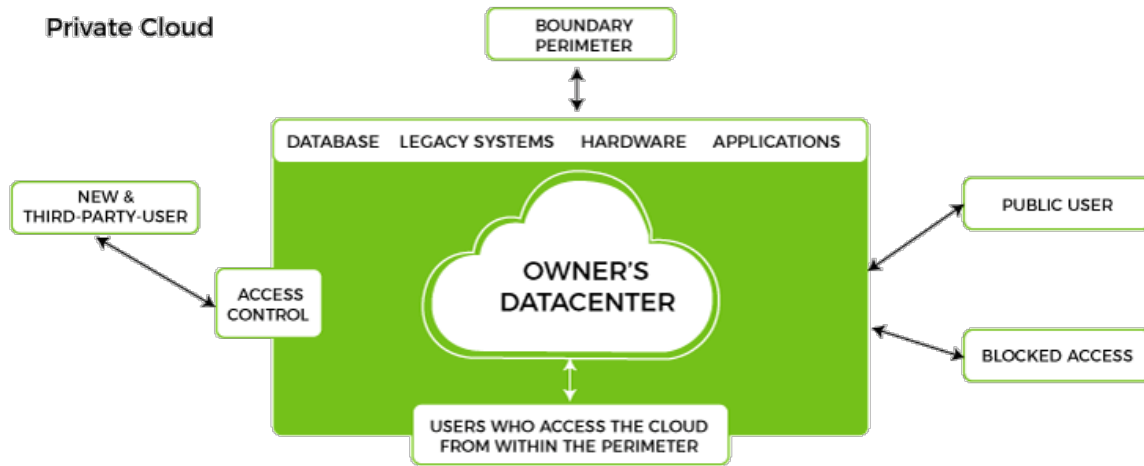
- Data Security and Privacy Concerns - Since it is accessible to all, it does not fully protect against cyber-attacks and could lead to vulnerabilities.
- Reliability Issues - Since the same server network is open to a wide range of users, it can lead to malfunction and outages
- Service/License Limitation - While there are many resources you can exchange with tenants, there is a usage cap.

Private Cloud

Now that you understand what the public cloud could offer you, of course, you are keen to know what a private cloud can do. Companies that look for cost efficiency and greater control over data & resources will find the private cloud a more suitable choice.

It means that it will be integrated with your data center and managed by your IT

team. Alternatively, you can also choose to host it externally. The private cloud offers bigger opportunities that help meet specific organizations' requirements when it comes to customization. It's also a wise choice for mission-critical processes that may have frequently changing requirements.



Benefits of Private Cloud

- Data Privacy - It is ideal for storing corporate data where only authorized personnel gets access
- Security - Segmentation of resources within the same Infrastructure can help with better access and higher levels of security.
- Supports Legacy Systems - This model supports legacy systems that cannot access the public cloud.

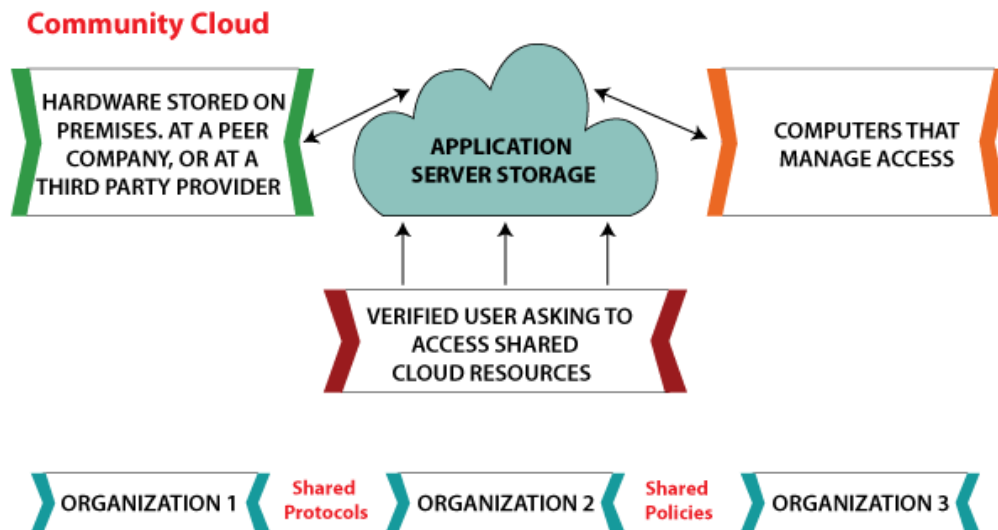
Limitations of Private Cloud

- Higher Cost - With the benefits you get, the investment will also be larger than the public cloud. Here, you will pay for software, hardware, and resources for staff and training.
- Fixed Scalability - The hardware you choose will accordingly help you scale in a certain direction
- High Maintenance - Since it is managed in-house, the maintenance costs also increase.

Community Cloud

The community cloud operates in a way that is similar to the public cloud. There's just one difference - it allows access to only a specific set of users who share

common objectives and use cases. This type of deployment model of cloud computing is managed and hosted internally or by a third-party vendor. However, you can also choose a combination of all three.



Benefits of Community Cloud

- Smaller Investment - A community cloud is much cheaper than the private & public cloud and provides great performance
- Setup Benefits - The protocols and configuration of a community cloud must align with industry standards, allowing customers to work much more efficiently.

Limitations of Community Cloud

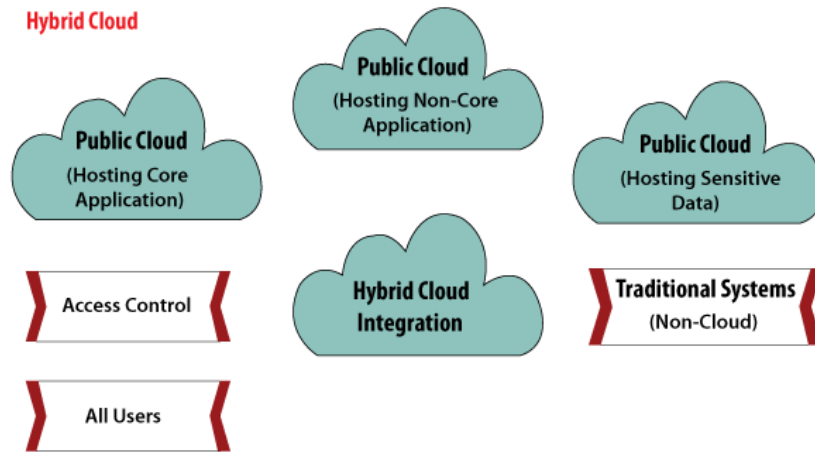
- Shared Resources - Due to restricted bandwidth and storage capacity, community resources often pose challenges.
- Not as Popular - Since this is a recently introduced model, it is not that popular or available across industries

Hybrid Cloud

As the name suggests, a hybrid cloud is a combination of two or more cloud architectures. While each model in the hybrid cloud functions differently, it is all part of the same architecture. Further, as part of this deployment of the cloud computing model, the internal or external providers can offer resources.

Let's understand the hybrid model better. A company with critical data will prefer

storing on a private cloud, while less sensitive data can be stored on a public cloud. The hybrid cloud is also frequently used for 'cloud bursting'. It means, supposes an organization runs an application on-premises, but due to heavy load, it can burst into the public cloud.



Benefits of Hybrid Cloud

- Cost-Effectiveness - The overall cost of a hybrid solution decreases since it majorly uses the public cloud to store data.
- Security - Since data is properly segmented, the chances of data theft from attackers are significantly reduced.
- Flexibility - With higher levels of flexibility, businesses can create custom solutions that fit their exact requirements

Limitations of Hybrid Cloud

- Complexity - It is complex setting up a hybrid cloud since it needs to integrate two or more cloud architectures
- Specific Use Case - This model makes more sense for organizations that have multiple use cases or need to separate critical and sensitive data

Real World Applications of Cloud Computing

In simple Cloud Computing refers to the on-demand availability of IT resources over internet. It delivers different types of services to the customer over the internet. There are three basic types of services models are available in cloud computing i.e., Infrastructure As A Service (IAAS), Platform As A Service (PAAS), Software As A Service (SAAS). On the basis of accessing and availing cloud computing services, they are divided mainly into four types of cloud i.e Public cloud, Private Cloud, Hybrid Cloud, and

Community cloud which is called Cloud deployment model. The demand for cloud services is increasing so fast and the global cloud computing market is growing at that rate. A large number of organizations and different business sectors are preferring cloud services nowadays as they are getting a list of benefits from cloud computing. Different organizations using cloud computing for different purposes and with respect to that Cloud Service Providers are providing various applications in different fields. **Applications of Cloud Computing in real-world** : Cloud Service Providers (CSP) are providing many types of cloud services and now if we will cloud computing has touched every sector by providing various cloud applications. Sharing and managing resources is easy in cloud computing that's why it is one of the dominant fields of computing. These properties have made it an active component in many fields. Now let's know some of the real-world applications of cloud computing.

1. **Online Data Storage**: Cloud computing allows storing data like files, images, audios, and videos, etc on the cloud storage. The organization need not set physical storage systems to store a huge volume of business data which costs so high nowadays. As they are growing technologically, data generation is also growing with respect to time, and storing that becoming problem. In that situation, Cloud storage is providing this service to store and access data any time as per requirement.
2. **Backup and Recovery** : Cloud vendors provide security from their side by storing safe to the data as well as providing a backup facility to the data. They offer various recovery application for retrieving the lost data. In the traditional way backup of data is a very complex problem and also it is very difficult sometimes impossible to recover the lost data. But cloud computing has made backup and recovery applications very easy where there is no fear of running out of backup media or loss of data.
3. **Bigdata Analysis** : We know the volume of big data is so high where storing that in traditional data management system for an organization is impossible. But cloud computing has resolved that problem by allowing the organizations to store their large volume of data in cloud storage without worrying about physical storage. Next comes analyzing the raw data and finding out insights or useful information from it is a big challenge as it requires high-quality tools for data analytics. Cloud computing provides the biggest facility to organizations in terms of storing and analyzing big data.
4. **Testing and development** : Setting up the platform for development and finally performing different types of testing to check the readiness of the product before delivery requires different types of IT resources and infrastructure. But Cloud computing provides the easiest approach for development as well as testing even if deployment by using their IT resources with minimal expenses. Organizations find it more helpful as they got scalable and flexible cloud services for product development, testing, and deployment.

5. **Anti-Virus Applications** : Previously, organizations were installing antivirus software within their system even if we will see we personally also keep antivirus software in our system for safety from outside cyber threats. But nowadays cloud computing provides cloud antivirus software which means the software is stored in the cloud and monitors your system/organization's system remotely. This antivirus software identifies the security risks and fixes them. Sometimes also they give a feature to download the software.
6. **E-commerce Application** : Cloud-based e-commerce allows responding quickly to the opportunities which are emerging. Users respond quickly to the market opportunities as well as the traditional e-commerce responds to the challenges quickly. Cloud-based e-commerce gives a new approach to doing business with the minimum amount as well as minimum time possible. Customer data, product data, and other operational systems are managed in cloud environments.
7. **Cloud computing in education** : Cloud computing in the education sector brings an unbelievable change in learning by providing e-learning, online distance learning platforms, and student information portals to the students. It is a new trend in education that provides an attractive environment for learning, teaching, experimenting, etc to students, faculty members, and researchers. Everyone associated with the field can connect to the cloud of their organization and access data and information from there.
8. **E-Governance Application** : Cloud computing can provide its services to multiple activities conducted by the government. It can support the government to move from the traditional ways of management and service providers to an advanced way of everything by expanding the availability of the environment, making the environment more scalable and customized. It can help the government to reduce the unnecessary cost in managing, installing, and upgrading applications and doing all these with help of cloud computing and utilizing that money public service.
9. **Cloud Computing in Medical Fields** : In the medical field also nowadays cloud computing is used for storing and accessing the data as it allows to store data and access it through the internet without worrying about any physical setup. It facilitates easier access and distribution of information among the various medical professional and the individual patients. Similarly, with help of cloud computing offsite buildings and treatment facilities like labs, doctors making emergency house calls and ambulances information, etc can be easily accessed and updated remotely instead of having to wait until they can access a hospital computer.
10. **Entertainment Applications** : Many people get entertainment from the internet, in that case, cloud computing is the perfect place for reaching to a varied consumer base. Therefore different types of entertainment industries reach near the target audience by adopting a multi-cloud strategy. Cloud-based entertainment provides various

entertainment applications such as online music/video, online games and video conferencing, streaming services, etc and it can reach any device be it TV, mobile, set-top box, or any other form. It is a new form of entertainment called On-Demand Entertainment (ODE). With respect to this as a cloud, the market is growing rapidly and it is providing various services day by day. So other application of cloud computing includes social applications, management application, business applications, art application, and many more. So in the future cloud computing is going to touch many more sectors by providing more applications and services.

1. What are the different commands used in MySQL?
2. Describe Different types of cloud computing deployment models
3. a) Write a program to insert a record using Spring JDBC
b) Write short note on DDL query with example
4. a) Write a program to delete a record using Spring JDBC
b) Explain about DML Query with example
5. a) Write a program to update a record using Spring JDBC[8M]
b) Describe DCL and TCL in detail with example
6. Write a program to read records using Spring JDBC
7. Describe Different types of cloud computing deployment models
8. Explain Real World Applications of Cloud Computing.
9. Write a program to perform any DML operation using Spring JDBC

Command Prompt