MRCET CAMPUS

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(AUTONOMOUS INSTITUTION – UGC, GOVT. OF INDIA)

Department of CSE

(Emerging Technologies)

(DATASCIENCE,CYBERSECURITY,IOT)

B.TECH(R-20 Regulation)

(IV YEAR – I SEM)

(2023-24)

FULL STACK DEVELOPMENT

(R20A0516)

LECTURE NOTES

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

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

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

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

Department of Computer Science and Engineering

EMERGING TECHNOLOGIES

FULL STACK DEVELOPMENT

(R20A0516)

LECTURE NOTES

Prepared by

V.Suneetha,Associate Professor
M.Gayatri,Associate Professor
D.Kalpana,Associate Professor

On
01.06.2023

Departmen t of Computer Science and Engineering

EMERGING TECHNOLOGIES

Vision

- "To be at the forefront of Emerging Technologies and to evolve as a Centre of Excellence in Research, Learning and Consultancy to foster the students into globally competent professionals useful to the Society."

Mission

The department of CSE (Emerging Technologies) is committed to:

- To offer highest Professional and Academic Standards in terms of Personal growth and satisfaction.
- Make the society as the hub of emerging technologies and thereby capture opportunities in new age technologies.
- To create a benchmark in the areas of Research, Education and Public Outreach.
- To provide students a platform where independent learning and scientific study are encouraged with emphasis on latest engineering techniques.

QUALITY POLICY

- To pursue continual improvement of teaching learning process of Undergraduate and Post Graduate programs in Engineering & Management vigorously.
- To provide state of art infrastructure and expertise to impart the quality education and research environment to students for a complete learning experiences.
- Developing students with a disciplined and integrated personality.
- To offer quality relevant and cost effective programmes to produce engineers as per requirements of the industry need.

For more information: www.mrcet.ac.in

MALLAREDDY COLLEGE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF CSE(Emerging Technologies)

INDEX

SNO	UNIT	TOPIC	PAGE NO
1	I	Web development Basics - HTML	2
2	I	Web servers	35
3	I	UNIX CLI Version control - Git & Github	36
4	I	HTML, CSS	40
5	II	Javascript basics OOPS Aspects of JavaScript Memory usage and Functions in JS	60
6	II	AJAX for data exchange with server jQuery Framework jQuery events	69
7	II	JSON data format.	80
8	III	Introduction to React	83
9	III	React Router and Single Page Applications React Forms	90
10	III	Flow Architecture	105
11	III	Introduction to Redux More Redux	107
12	III	Client-Server Communication	125
13	IV	Java Web Development	129
14	IV	Model View Controller (MVC) Pattern	130
15	IV	MVC Architecture using Spring RESTful API	135
16	IV	Spring Framework Building an application using Maven	146
17	V	Relational schemas	149
18	V	Normalization Structured Query Language (SQL)	153
19	V	Data persistence using Spring JDBC Agile development principles	154
20	V	Deploying application in Cloud	167

(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.

HTML Document Structure

A typical HTML document will have the following structure:

```
Document declaration tag
<html>
   <head>
       Document header related tags
   </head>

   <body>
       Document body related tags
   </body>
</html>
```

We will study all the header and body tags in subsequent chapters, but for now let's see what is document declaration tag.

The <!DOCTYPE> Declaration

The <!DOCTYPE> declaration tag is used by the web browser to understand the version of the HTML used in the document. Current version of HTML is 5 and it makes use of the following declaration:

```
<!DOCTYPE html>
```

There are many other declaration types which can be used in HTML document depending on what version of HTML is being used. We will see more details on this while discussing <!DOCTYPE...> tag along with other HTML tags.

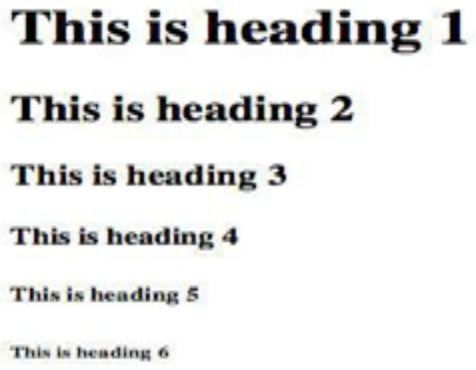
Heading Tags

Any document starts with a heading. You can use different sizes for your headings. HTML also has six levels of headings, which use the elements **<h1>, <h2>, <h3>, <h4>, <h5>, and <h6>**. While displaying any heading, browser adds one line before and one line after that heading.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Heading Example</title>
</head>
<body>
<h1>This is heading 1</h1>
<h2>This is heading 2</h2>
<h3>This is heading 3</h3>
<h4>This is heading 4</h4>
<h5>This is heading 5</h5>
<h6>This is heading 6</h6>
</body>
</html>
```

This will produce the following result:



Paragraph Tag

The ** ** tag offers a way to structure your text into different paragraphs. Each paragraph of text should go in between an opening and a closing tag as shown below in the example:

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Paragraph Example</title>
</head>
<body>
<p>Here is a first paragraph of text.</p>
<p>Here is a second paragraph of text.</p>
<p>Here is a third paragraph of text.</p>
</body>
</html>
```

This will produce the following result:

```
Here is a first paragraph of text.
Here is a second paragraph of text.
Here is a third paragraph of text.
```

Line Break Tag

Whenever you use the **
** element, anything following it starts from the next line. This tag is an example of an **empty** element, where you do not need opening and closing tags, as there is nothing to go in between them.

The
 tag has a space between the characters **br** and the forward slash. If you omit this space, older browsers will have trouble rendering the line break, while if you miss the forward slash character and just use
 it is not valid in XHTML.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Line Break  Example</title>
</head>
<body>
```

```
<p>Hello<br />
You delivered your assignment on time.<br />
Thanks<br />
Mahnaz</p>
</body>
</html>
```

This will produce the following result:

```
Hello
You delivered your assignment on time.
Thanks
Mahnaz
```

Centering Content

You can use ** ** tag to put any content in the center of the page or any table cell.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Centring Content Example</title>
</head>
<body>
<p>This text is not in the center.</p>
<center>
<p>This text is in the center.</p>
</center>
</body>
</html>
```

This will produce the following result:

This text is not in the center.

This text is in the center.

Horizontal Lines

Horizontal lines are used to visually break-up sections of a document. The **<hr>** tag creates a line from the current position in the document to the right margin and breaks the line accordingly.

For example, you may want to give a line between two paragraphs as in the given example below:

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Horizontal Line Example</title>
</head>
<body>
<p>This is paragraph one and should be on top</p>
<hr />
<p>This is paragraph two and should be at bottom</p>
</body>
</html>
```

This will produce the following result:

This is paragraph one and should be on top

This is paragraph two and should be at bottom

Again **<hr />** tag is an example of the **empty** element, where you do not need opening and closing tags, as there is nothing to go in between them.

The **<hr />** element has a space between the characters **hr** and the forward slash. If you omit this space, older browsers will have trouble rendering the horizontal line, while if you miss the forward slash character and just use **<hr>** it is not valid in XHTML

Preserve Formatting

Sometimes, you want your text to follow the exact format of how it is written in the HTML document. In these cases, you can use the preformatted tag **<pre>**.

Any text between the opening **<pre>** tag and the closing **</pre>** tag will preserve the formatting of the source document.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Preserve Formatting Example</title>
</head>
<body>
<pre>
function testFunction( strText ){
   alert (strText)
}
</pre>
</body>
</html>
```

This will produce the following result:

```
function testFunction( strText ){
  alert (strText)
}
```

Try using the same code without keeping it inside **<pre>...</pre>** tags

Nonbreaking Spaces

Suppose you want to use the phrase "12 Angry Men." Here, you would not want a browser to split the "12, Angry" and "Men" across two lines:

```
An example of this technique appears in the movie "12 Angry Men."
```

In cases, where you do not want the client browser to break text, you should use a nonbreaking space entity ** ** instead of a normal space. For example, when coding the "12 Angry Men" in a paragraph, you should use something similar to the following code:

Example

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

```
<head>
<title>Nonbreaking Spaces Example</title>
</head>
<body>
<p>An example of this technique appears in the movie "12 Angry Men."</p>
</body>
</html>
```

HTML – ELEMENTS

HTML element is defined by a starting tag. If the element contains other content, it ends with a closing tag, where the element name is preceded by a forward slash as shown below with few tags:

Start Tag	Content	End Tag
	This is paragraph content.	
<h1>	This is heading content.	</h1>
	This is division content.	

So here ** …. ** is an HTML element, **<h1>…</h1>** is another HTML element. There are some HTML elements which don't need to be closed, such as **<img…/>**, **<hr />** and **
** elements. These are known as **void elements**.

HTML documents consists of a tree of these elements and they specify how HTML documents should be built, and what kind of content should be placed in what part of an HTML document.

HTML Tag vs. Element

An HTML element is defined by a *starting tag*. If the element contains other content, it ends with a *closing tag*.

For example, ** ** is starting tag of a paragraph and ** ** is closing tag of the same paragraph but ** This is paragraph ** is a paragraph element.

Nested HTML Elements

It is very much allowed to keep one HTML element inside another HTML element:

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Nested Elements Example</title>
</head>
<body>
<h1>This is <i>italic</i> heading</h1>
<p>This is <u>underlined</u> paragraph</p>
</body>
</html>
```

This will display the following result:

This is *italic* heading

This is underlined paragraph

3. HTML – ATTRIBUTES

We have seen few HTML tags and their usage like heading tags **<h1>, <h2>,** paragraph tag **<p>** and other tags. We used them so far in their simplest form, but most of the HTML tags can also have attributes, which are extra bits of information.

An attribute is used to define the characteristics of an HTML element and is placed inside the element's opening tag. All attributes are made up of two parts: a **name** and a **value**:

- The **name** is the property you want to set. For example, the paragraph **<p>** element inthe example carries an attribute whose name is **align**, which you can use to indicate

the alignment of paragraph on the page.

-
- The **value** is what you want the value of the property to be set and always put within quotations. The below example shows three possible values of align attribute: **left, center** and **right**.

Attribute names and attribute values are case-insensitive. However, the World Wide Web Consortium (W3C) recommends lowercase attributes/attribute values in their HTML 4 recommendation.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Align Attribute  Example</title>
</head>
<body>
<p align="left">This is left aligned</p>
<p align="center">This is center aligned</p>
<p align="right">This is right aligned</p>
</body>
</html>
```

This will display the following result:

```
This is left aligned
                              This is center aligned
                                                    This is right aligned
```

Core Attributes

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

- Id
- Title
- Class
- Style

The Id Attribute

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

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

We will discuss style sheet in separate tutorial. For now, let's use the id attribute to distinguish between two paragraph elements as shown below.

Example

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

The title Attribute

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

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

Example

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


<title>The title Attribute Example</title>
```

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

This will produce the following result:

Titled Heading Tag Example

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

The class Attribute

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

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

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

The style Attribute

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

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

This will produce the following result:

Some text...

At this point of time, we are not learning CSS, so just let's proceed without bothering much about CSS. Here, you need to understand what are HTML attributes and how they can be used while formatting content.

Internationalization Attributes

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

- dir
- lang
- xml:lang

The dir Attribute

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

Value	Meaning
ltr	Left to right (the default value)
rtl	Right to left (for languages such as Hebrew or Arabic that are read right to left)

Example

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

This will produce the following result:

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

When *dir* attribute is used within the <html> tag, it determines how text will be presented within the entire document. When used within another tag, it controls the text's direction for just the content of that tag.

The lang Attribute

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

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

Example

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

The xml:lang Attribute

The *xml:lang* attribute is the XHTML replacement for the *lang* attribute. The value of the*xml:lang* attribute should be an ISO-639 country code as mentioned in previous section.

Generic Attributes

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

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

We will see related examples as we will proceed to study other HTML tags. For a complete list of HTML Tags and related attributes please check reference to **HTML Tags List**.

If you use a word processor, you must be familiar with the ability to make text bold, italicized, or underlined; these are just three of the ten options available to indicate how text can appear in HTML and XHTML.

Bold Text

Anything that appears within **...** element, is displayed in bold as shown below:

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Bold Text Example</title>
</head>
<body>
<p>The following word uses a <b>bold</b> typeface.</p>
</body>
</html>
```

This will produce the following result:

```
The following word uses a bold typeface.
```

Italic Text

Anything that appears within **...** element is displayed in italicized as shown below:

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Italic Text Example</title>
</head>
<body>
<p>The following word uses a <i>italicized</i> typeface.</p>
</body>
</html>
```

This will produce the following result:

```
The following word uses an italicized typeface.
```

Underlined Text

Anything that appears within **<u>...</u>** element, is displayed with underline as shownbelow:

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Underlined Text Example</title>
</head>
<body>
<p>The following word uses a <u>underlined</u> typeface.</p>
</body>
</html>
```

This will produce the following result:

The following word uses an underlined typeface.

Strike Text

Anything that appears within **<strike>...</strike>** element is displayed with strikethrough, which is a thin line through the text as shown below:

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Strike Text Example</title>
</head>
<body>
<p>The following word uses a <strike>strikethrough</strike> typeface.</p>
</body>
</html>
```

This will produce the following result:

The following word uses a ~~strikethrough~~ typeface.

Monospaced Font

The content of a **<tt>...</tt>** element is written in monospaced font. Most of the fonts are known as variable-width fonts because different letters are of different widths (for example, the letter 'm' is wider than the letter 'i'). In a monospaced font, however, each letter has the same width.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Monospaced Font Example</title>
</head>
<body>
<p>The following word uses a <tt>monospaced</tt> typeface.</p>
</body>
</html>
```

This will produce the following result:

```
The following word uses a monospaced typeface.
```

Superscript Text

The content of a **^{...}** element is written in superscript; the font size used is the same size as the characters surrounding it but is displayed half a character's height above the other characters.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Superscript Text Example</title>
</head>
<body>
<p>The following word uses a <sup>superscript</sup> typeface.</p>
</body>
</html>
```

This will produce the following result:

```
The following word uses a superscript typeface.
```

Subscript Text

The content of a **_{...}** element is written in subscript; the font size used is the same as the characters surrounding it, but is displayed half a character's height beneath the other characters.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Subscript Text Example</title>
</head>
<body>
<p>The following word uses a <sub>subscript</sub> typeface.</p>
</body>
</html>
```

This will produce the following result:

The following word uses a subscript typeface.

Inserted Text

Anything that appears within **<ins>...</ins>** element is displayed as inserted text.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Inserted Text Example</title>
</head>
<body>
<p>I want to drink <del>cola</del> <ins>wine</ins></p>
</body>
</html>
```

This will produce the following result:

I want to drink ~~cola~~ wine

Deleted Text

Anything that appears within **~~...~~** element, is displayed as deleted text.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Deleted Text Example</title>
</head>
<body>
<p>I want to drink <del>cola</del> <ins>wine</ins></p>
</body>
</html>
```

This will produce the following result:

I want to drink ~~cola~~ wine

Larger Text

The content of the **<big>...</big>** element is displayed one font size larger than the rest of the text surrounding it as shown below:

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Larger Text Example</title>
</head>
<body>
<p>The following word uses a <big>big</big> typeface.</p>
</body>
```

```
</html>
```

This will produce the following result:

```
The following word uses a big typeface.
```

Smaller Text

The content of the **<small>...</small>** element is displayed one font size smaller than the rest of the text surrounding it as shown below:

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Smaller Text Example</title>
</head>
<body>
<p>The following word uses a <small>small</small> typeface.</p>
</body>
</html>
```

This will produce the following result:

```
The following word uses a small typeface.
```

Grouping Content

The ** ** and **** elements allow you to group together several elements to create sections or subsections of a page.

For example, you might want to put all of the footnotes on a page within a element to indicate that all of the elements within that element relate to the footnotes. You might then attach a style to this element so that they appear using a special set of style rules.

Example

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

```
<head>
<title>Div Tag Example</title>
</head>
<body>
<div id="menu" align="middle" >
<a href="/index.htm">HOME</a> |
<a href="/about/contact_us.htm">CONTACT</a> |
<a href="/about/index.htm">ABOUT</a> </div>

<div id="content" align="left" bgcolor="white">
<h5>Content Articles</h5>
<p>Actual content goes here. ...</p>
</div>
</body>
</html>
```

This will produce the following result:

HOME | CONTACT | ABOUT

CONTENT ARTICLES

Actual content goes here.....

The element, on the other hand, can be used to group inline elements only. So, if you have a part of a sentence or paragraph which you want to group together, you could use the element as follows

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Span Tag Example</title>
</head>
<body>
<p>This is the example of <span style="color:green">span tag</span> and the <span
style="color:red">div tag</span> alongwith CSS</p>
```

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

This will produce the following result:

This is the example of span tag and the div tag along with CSS

These tags are commonly used with CSS to allow you to attach a style to a section of a page.

6. HTML – PHRASE TAGS

The phrase tags have been desicolgned for specific purposes, though they are displayed in a similar way as other basic tags like **<b>, <i>, <pre>,** and **<tt>,** you have seen in previous chapter. This chapter will take you through all the important phrase tags, so let's start seeing them one by one.

Emphasized Text

Anything that appears within **...** element is displayed as emphasized text.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Emphasized Text Example</title>
</head>
<body>
<p>The following word uses a <em>emphasized</em> typeface.</p>
</body>
</html>
```

This will produce the following result:

The following word uses an *emphasized* typeface.

Marked Text

Anything that appears with-in **<mark>...</mark>** element, is displayed as marked with yellow ink.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Marked Text Example</title>
</head>
<body>
<p>The following word has been <mark>marked</mark> with yellow</p>
</body>
</html>
```

This will produce the following result:
The following word has been marked with yellow

Strong Text

Anything that appears within **...** element is displayed as important text.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Strong Text Example</title>
</head>
<body>
<p>The following word uses a <strong>strong</strong> typeface.</p>
</body>
</html>
```

This will produce the following result:

The following word uses a **strong** typeface.

Text Abbreviation

You can abbreviate a text by putting it inside opening <abbr> and closing </abbr> tags. If present, the title attribute must contain this full description and nothing else.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Text Abbreviation</title>
</head>
```

```
<body>
<p>My best friend's name is  <abbr title="Abhishek">Abhy</abbr>.</p>
</body>
</html>
```

This will produce the following result:

My best friend's name is Abhy.

Acronym Element

The **<acronym>** element allows you to indicate that the text between <acronym> and </acronym> tags is an acronym.

At present, the major browsers do not change the appearance of the content of the <acronym> element.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Acronym Example</title>
</head>
<body>
<p>This chapter covers marking up text in <acronym>XHTML</acronym>.</p>
</body>
</html>
```

This will produce the following result:

This chapter covers marking up text in XHTML.

Text Direction

The **<bdo>...</bdo>** element stands for Bi-Directional Override and it is used to override the current text direction.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Text Direction Example</title>
</head>
<body>
<p>This text will go left to right.</p>
<p><bdo dir="rtl">This text will go right to left.</bdo></p>
</body>
</html>
```

This will produce the following result:

This text will go left to right.

This text will go right to left.

Special Terms

The **<dfn>...</dfn>** element (or HTML Definition Element) allows you to specify that you are introducing a special term. It's usage is similar to italic words in the midst of a paragraph.

Typically, you would use the <dfn> element the first time you introduce a key term. Most recent browsers render the content of a <dfn> element in an italic font.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Special Terms Example</title>
</head>
<body>
<p>The following word is a <dfn>special</dfn> term.</p>
</body>
```

```
</html>
```

This will produce the following result:

The following word is a *special* term.

Quoting Text

When you want to quote a passage from another source, you should put it in between**<blockquote>...</blockquote>** tags.

Text inside a <blockquote> element is usually indented from the left and right edges of the surrounding text, and sometimes uses an italicized font.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Blockquote Example</title>
</head>
<body>
<p>The following description of XHTML is taken from the W3C Web site:</p>

<blockquote>XHTML 1.0 is the W3C's first Recommendation for XHTML, following on
from earlier work on HTML 4.01, HTML 4.0, HTML 3.2 and HTML 2.0.</blockquote>
</body>
</html>
```

This will produce the following result:

The following description of XHTML is taken from the W3C Web site:

```
XHTML 1.0 is the W3C's first Recommendation for XHTML, following on from earlier
work on HTML 4.01, HTML 4.0, HTML 3.2 and HTML 2.0.
```

Short Quotations

The **<q>...</q>** element is used when you want to add a double quote within a sentence.

Example

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

```
<title>Double Quote Example</title>
</head>
<body>
<p>Amit is in Spain, <q>I think I am wrong</q>.</p>
</body>
</html>
```

This will produce the following result:

Amit is in Spain, I think I am wrong.

Text Citations

If you are quoting a text, you can indicate the source placing it between an opening **<cite>**tag and closing **</cite>** tag

As you would expect in a print publication, the content of the <cite> element is rendered in italicized text by default.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Citations Example</title>
</head>
<body>
<p>This HTML tutorial is derived from <cite>W3 Standard for HTML</cite>.</p>
</body>
</html>
```

This will produce the following result:

This HTML tutorial is derived from *W3 Standard for HTML*.

Computer Code

Any programming code to appear on a Web page should be placed inside **<code>...</code>**tags. Usually the content of the <code> element is presented in a monospaced font, just like the code in most programming books.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Computer Code Example</title>
</head>
<body>
<p>Regular text. <code>This is code.</code> Regular text.</p>
</body>
</html>
```

This will produce the following result:

Regular text. `This is code`. Regular text.

Keyboard Text

When you are talking about computers, if you want to tell a reader to enter some text, you can use the **<kbd>...</kbd>** element to indicate what should be typed in, as in this example.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Keyboard Text Example</title>
</head>
<body>
<p>Regular text. <kbd>This is inside kbd element</kbd> Regular text.</p>
</body>
</html>
```

This will produce the following result:

Regular text. `This is inside kbd element` Regular text.

Programming Variables

This element is usually used in conjunction with the **<pre>** and **<code>** elements to indicate that the content of that element is a variable.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Variable Text Example</title>
</head>
<body>
<p><code>document.write("<var>user-name</var>")</code></p>
</body>
</html>
```

This will produce the following result:

`document.write("`*`user-name`*`")`

Program Output

The **<samp>...</samp>** element indicates sample output from a program, and script etc. Again, it is mainly used when documenting programming or coding concepts.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Program Output Example</title>
</head>
<body>
<p>Result produced by the program is <samp>Hello World!</samp></p>
</body>
</html>
```

This will produce the following result:

Result produced by the program is `Hello World!`

Address Text

The **<address>...</address>** element is used to contain any address.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Address Example</title>
</head>
<body>
<address>388A, Road No 22, Jubilee Hills -  Hyderabad</address>
</body>
</html>
```

This will produce the following result:

388A, Road No 22, Jubilee Hills - Hyderabad

7.HTML – META TAGS

HTML lets you specify metadata - additional important information about a document in a variety of ways. The META elements can be used to include name/value pairs describing properties of the HTML document, such as author, expiry date, a list of keywords, document author etc.

The **<meta>** tag is used to provide such additional information. This tag is an empty element and so does not have a closing tag but it carries information within its attributes.

You can include one or more meta tags in your document based on what information you want to keep in your document but in general, meta tags do not impact physical appearance of the document so from appearance point of view, it does not matter if you include them or not.

Adding Meta Tags to Your Documents

You can add metadata to your web pages by placing <meta> tags inside the header of the document which is represented by **<head>** and **</head>** tags. A meta tag can have following attributes in addition to core attributes:

Attribute	Description

Name	Name for the property. Can be anything. Examples include, keywords, description, author, revised, generator etc.
content	Specifies the property's value.
scheme	Specifies a scheme to interpret the property's value (as declared in the content attribute).
http-equiv	Used for http response message headers. For example, http-equiv can be used to refresh the page or to set a cookie. Values include content-type, expires, refresh and set-cookie.

Specifying Keywords

You can use <meta> tag to specify important keywords related to the document and later these keywords are used by the search engines while indexing your webpage for searching purpose.

Example

Following is an example, where we are adding HTML, Meta Tags, Metadata as important keywords about the document.

```
<!DOCTYPE html>
<html>
<head>
<title>Meta Tags Example</title>
<meta name="keywords" content="HTML, Meta Tags, Metadata" />
</head>
<body>
<p>Hello HTML5!</p>
</body>
</html>
```

This will produce the following result:

```
Hello HTML5!
```

Document Description

You can use <meta> tag to give a short description about the document. This again can be used by various search engines while indexing your webpage for searching purpose.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Meta Tags Example</title>
<meta name="keywords" content="HTML, Meta Tags, Metadata" />
<meta name="description" content="Learning about Meta Tags." />
</head>
<body>
<p>Hello HTML5!</p>
</body>
</html>
```

Document Revision Date

You can use <meta> tag to give information about when last time the document was updated. This information can be used by various web browsers while refreshing your webpage.

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Meta Tags Example</title>
<meta name="keywords" content="HTML, Meta Tags, Metadata" />
<meta name="description" content="Learning about Meta Tags." />
<meta name="revised" content="Tutorialspoint, 3/7/2014" />
</head>
<body>
<p>Hello HTML5!</p>
</body>
</html>
```

Document Refreshing

A <meta> tag can be used to specify a duration after which your web page will keep refreshing automatically.

Example

If you want your page keep refreshing after every 5 seconds then use the following syntax.

```
<!DOCTYPE html>
<html>
<head>
<title>Meta Tags Example</title>
<meta name="keywords" content="HTML, Meta Tags, Metadata" />
<meta name="description" content="Learning about Meta Tags." />
<meta name="revised" content="Tutorialspoint, 3/7/2014" />
<meta http-equiv="refresh" content="5" />
</head>
<body>
<p>Hello HTML5!</p>
</body>
</html>
```

Page Redirection

You can use <meta> tag to redirect your page to any other webpage. You can also specify a duration if you want to redirect the page after a certain number of seconds.

Example

Following is an example of redirecting current page to another page after 5 seconds. If you want to redirect page immediately then do not specify *content* attribute.

```
<!DOCTYPE html>
<html>
<head>
<title>Meta Tags Example</title>
<meta name="keywords" content="HTML, Meta Tags, Metadata" />
<meta name="description" content="Learning about Meta Tags." />
<meta name="revised" content="Tutorialspoint, 3/7/2014" />
<meta http-equiv="refresh" content="5; url=http://www.tutorialspoint.com" />
</head>
<body>
```

```
<p>Hello HTML5!</p>
</body>
</html>
```

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

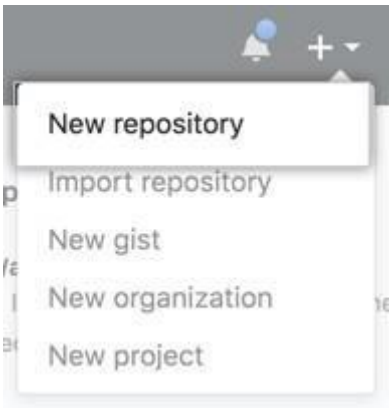
To complete this tutorial, you need a GitHub account and Internet access. You don't need to know how to code, use the command line, or install Git (the version control software that GitHub is built on). If you have a question about any of the expressions used in this guide, head on over to the glossary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

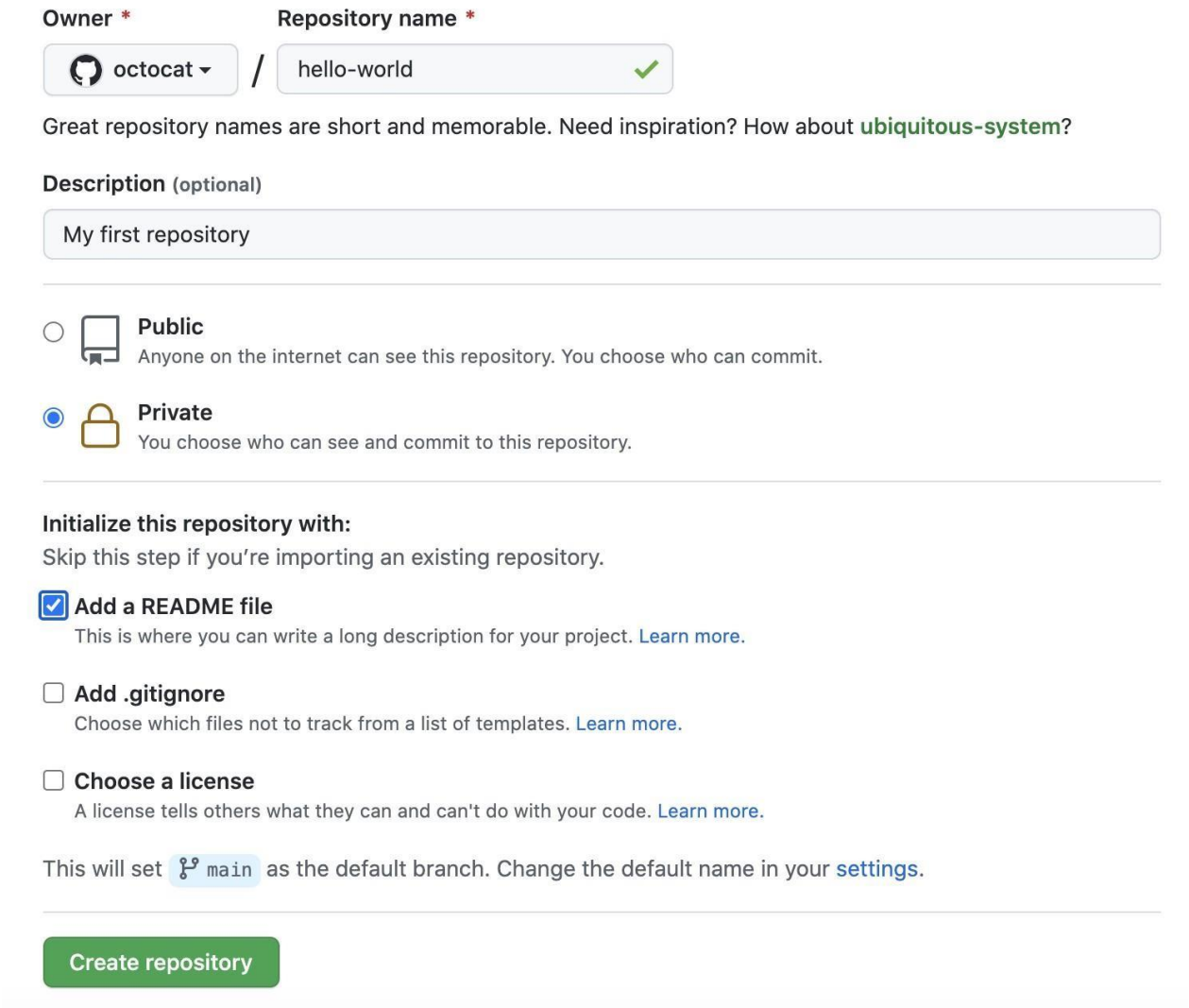
Your hello-world repository can be a place where you store ideas, resources, or even share and discuss things with others.

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.



Creating a branch

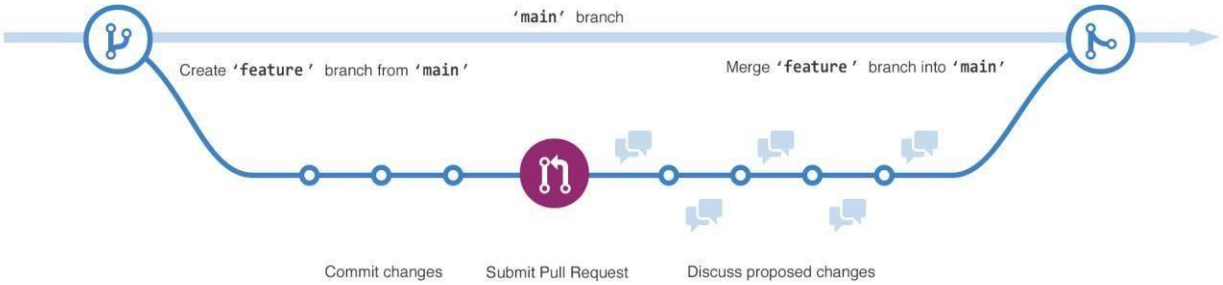
Branching lets you have different versions of a repository at one time.
By default, your repository has one branch named main that is considered to be the definitive branch. You can create additional branches off of main in your repository. You can use branches to have different versions of a project at one time. This is helpful when you want to add new features to a project without changing the main source of code. The work done on different branches will not show up on the main branch until you merge it, which we

will cover later in this guide. You can use branches to experiment and make edits before committing them to main.

When you create a branch off the main branch, you're making a copy, or snapshot, of main as it was at that point in time. If someone else made changes to the main branch while you were working on your branch, you could pull in those updates.

This diagram shows:

- The main branch
- A new branch called feature
- The journey that feature takes before it's merged into main



Have you ever saved different versions of a file? Something like:

- story.txt
- story-edit.txt
- story-edit-reviewed.txt

Branches accomplish similar goals in GitHub repositories.

Here at GitHub, our developers, writers, and designers use branches for keeping bug fixes and feature work separate from our main (production) branch. When a change is ready, they merge their branch into main.

What is CSS?

While HTML is a **markup language** used to format/structure a web page, CSS is a **design language** that you use to make your web page look nice and presentable.
CSS stands for **Cascading Style Sheets**, and you use it to improve the appearance of a web page. By adding thoughtful CSS styles, you make your page more attractive and pleasant for the end user to view and use. Imagine if human beings were just made to have skeletons and bare bones – how would that look? Not nice if you ask me. So CSS is like our skin, hair, and general physical appearance.

You can also use CSS to layout elements by positioning them in specified areas of your page.

To access these elements, you have to "select" them. You can select a single or multiple web elements and specify how you want them to look or be positioned.

The rules that govern this process are called **CSS selectors.**
With CSS you can set the colour and background of your elements, as well as the typeface, margins, spacing, padding and so much more.

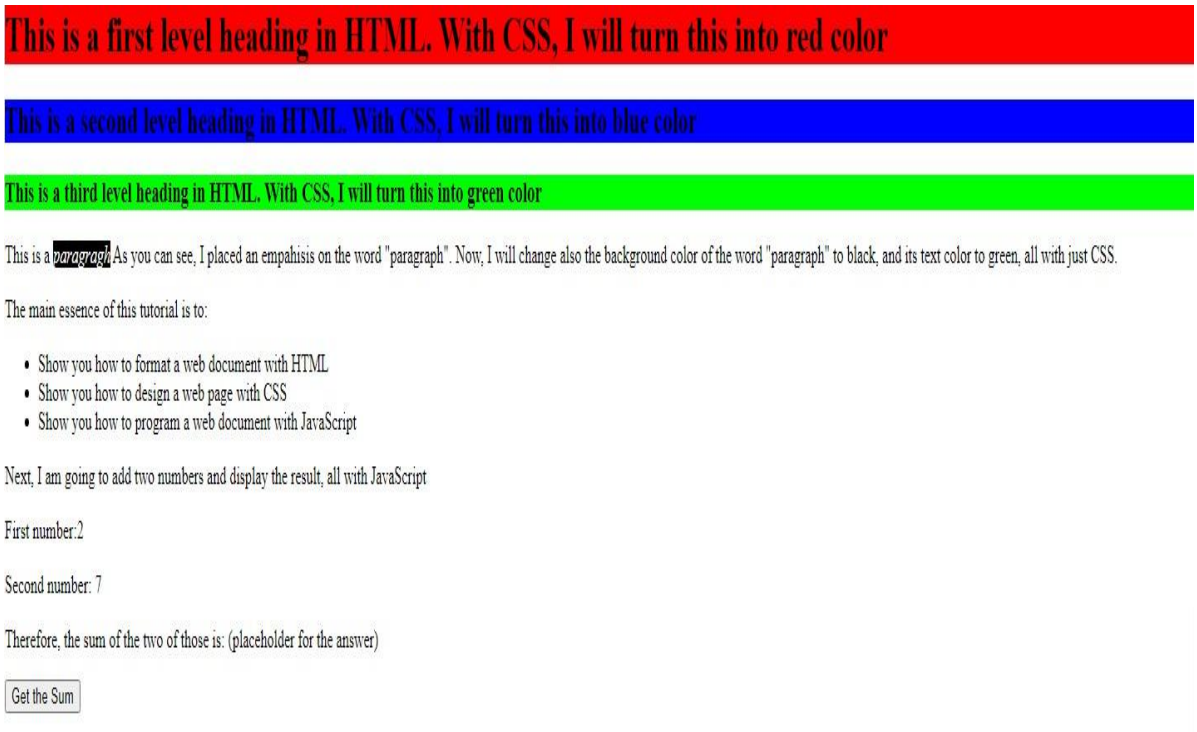
If you remember our example HTML page, we had elements which were pretty self-explanatory. For example, I stated that I would change the color of the level one heading `h1` to red.
To illustrate how CSS works, I will be sharing the code which sets the background-color of the three levels of headers to red, blue, and green respectively:

```
h1 {
   background-color: #ff0000;
}
```

```
h2 {
    background-color: #0000FF;
}

h3 {
    background-color: #00FF00;
}

em {
    background-color: #000000;
    color: #ffffff;
}
```

localhost:3000/styles.css

The above style, when applied, will change the appearance of our web page to this:



Cool, right?

We access each of the elements we want to work on by "selecting" them. The `h1` selects all level 1 headings in the page, the `h2` selects the level 2 elements, and so on. You can select any single HTML element you want and specify how you want it to look or be positioned.
Want to learn more about CSS? You can check out the second part of freeCodeCamp's Responsive Web Design certification to get started.

What is JavaScript?

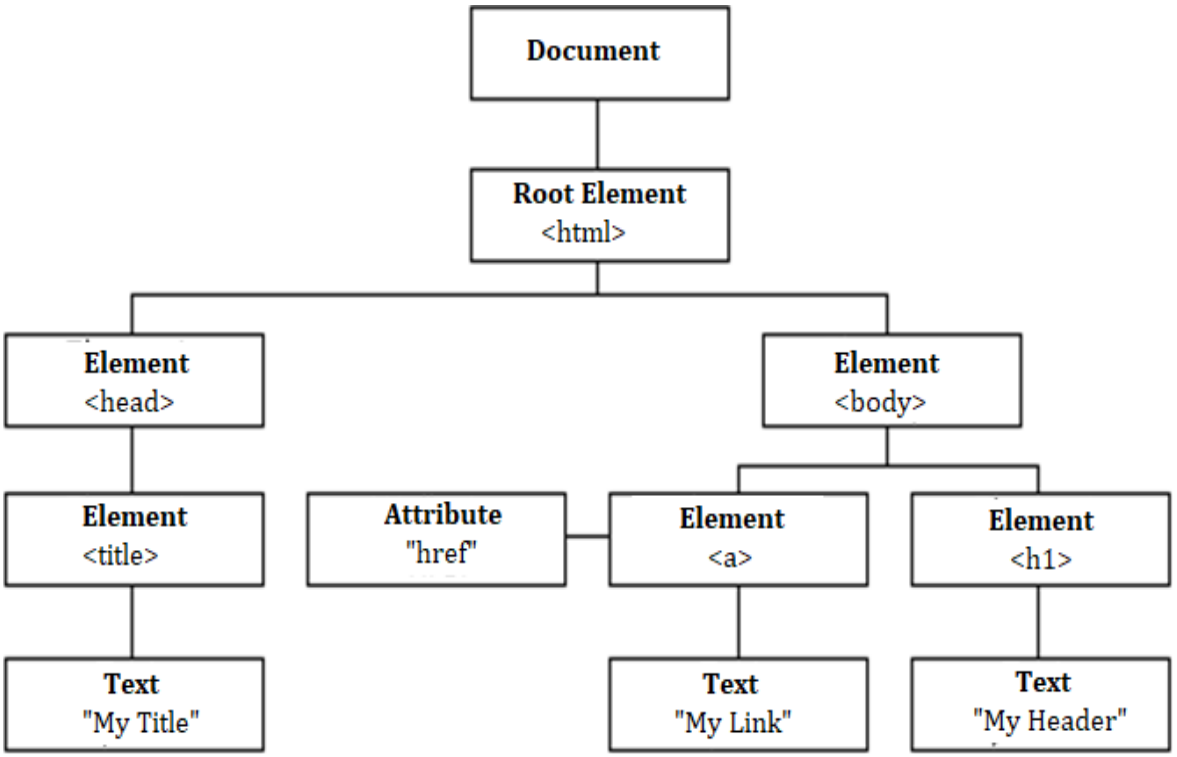
Now, if HTML is the **markup language** and CSS is the **design language**, then JavaScript is the **programming language.**
If you don't know what programming is, think of certain actions you take in your daily life:

When you sense danger, you run. When you are hungry, you eat. When you are tired, you sleep. When you are cold, you look for warmth. When crossing a busy road, you calculate the distance of vehicles away from you.

Your brain has been programmed to react in a certain way or do certain things whenever something happens. In this same way, you can program your web page or individual elements to react a certain way and to do something when something else (an event) happens.

You can program actions, conditions, calculations, network requests, concurrent tasks and many other kinds of instructions.

You can access any elements through the Document Object Model API (DOM) and make them change however you want them to.
The DOM is a tree-like representation of the web page that gets loaded into the browser.

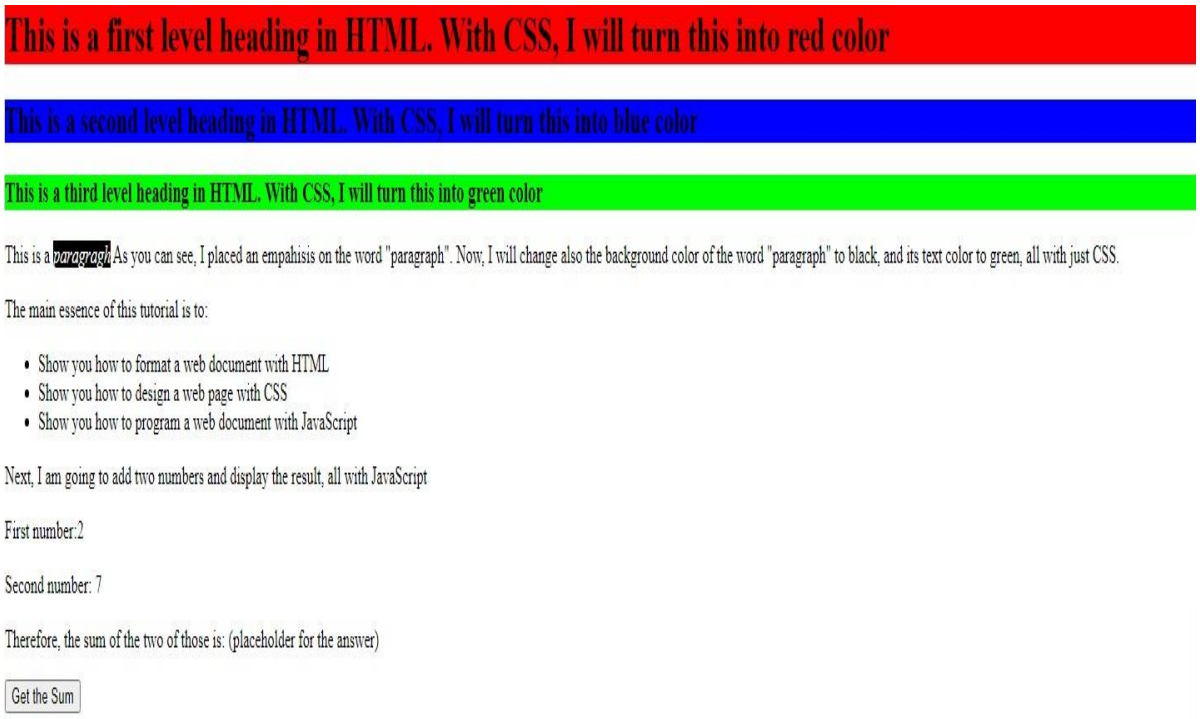


Each element on the web page is represented on the DOM

Thanks to the DOM, we can use methods like `getElementById()` to access elements from our web page.

JavaScript allows you to make your webpage **"think and act"**, which is what programming is all about.

If you remember from our example HTML page, I mentioned that I was going to sum up the two numbers displayed on the page and then display the result in the place of the placeholder text. The calculation runs once the button gets clicked.



Clicking the "Get the sum" button will display the sum of 2 and 7

This code illustrates how you can do calculations with JavaScript:

```
function displaySum() {
  let firstNum = Number(document.getElementById('firstNum').innerHTML)
  let secondNum = Number(document.getElementById('secondNum').innerHTML)

  let total = firstNum + secondNum;
  document.getElementById("answer").innerHTML = `${firstNum} + ${secondNum}, equals to ${total}`
;
}

document.getElementById('sumButton').addEventListener("click", displaySum);
```

Remember what I told you about HTML attributes and their uses? This code displays just that.

The `displaySum` is a function which gets both items from the web page, converts them to numbers (with the Number method), sums them up, and passes them in as inner values to another element.

The reason we were able to access these elements in our JavaScript was because we had set unique attributes on them, to help us identify them.

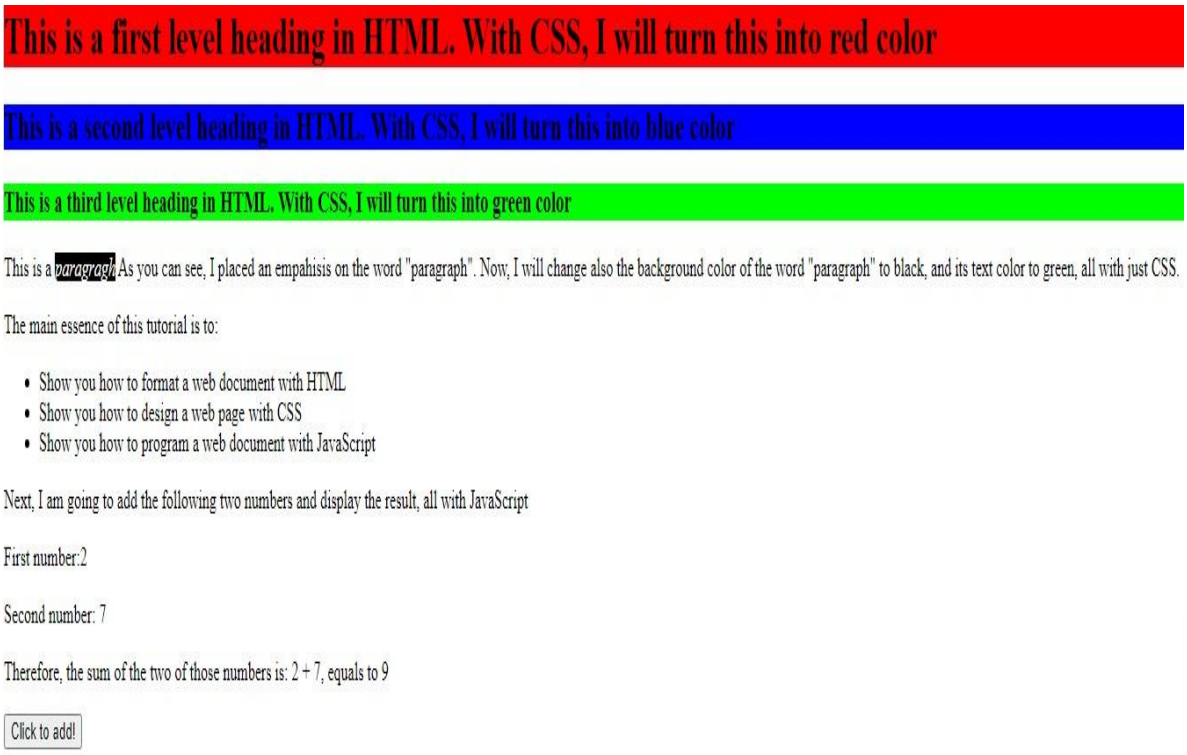
So thanks to this:

```
// id attribute has been set in all three


<span id= "firstNum">2</span> <br>
    ...<span id= "secondNum">7</span>
    ..... <span id= "answer">(placeholder for the answer)</span>
```

We were able to do this:

```
//getElementById will get all HTML elements by a specific "id" attribute
...
let firstNum = Number(document.getElementById('firstNum').innerHTML)
  let secondNum = Number(document.getElementById('secondNum').innerHTML)


  let total = firstNum + secondNum;
  document.getElementById("answer").innerHTML = `${firstNum} + ${secondNum}, equals to ${total}`
;
```

Finally, upon clicking the button, you will see the sum of the two numbers on the newly updated page:



2

plus 7 is equals to 9

If you want to get started with JavaScript, you can check out freeCodeCamp's JavaScript Algorithms and Data Structures certification. And you can use this great Intro to JS course to supplement your learning.

How to Put HTML, CSS, and JavaScript Together

Together, we use these three languages to format, design, and program web pages.

And when you link together some web pages with hyperlinks, along with all their assets like images, videos, and so on that are on the server computer, it gets rendered into a **website**.
This rendering typically happens on the front end, where the users can see what's being displayed and interact with it.

On the other hand, data, especially sensitive information like passwords, are stored and supplied from the back end part of the website. This is the part of a website which exists only on the server computer, and isn't displayed on the front-end browser. There, the user cannot see or readily access that information.

Wrapping Up

As a web developer, the three main languages we use to build websites are HTML, CSS, and JavaScript.

JavaScript is the programming language, we use HTML to structure the site, and we use CSS to design and layout the web page.

These days, CSS has become more than just a design language, though. You can actually implement animations and smooth transitions with just CSS.

In fact, you can do some basic programming with CSS too. An example of this is when you use media queries, where you define different style rules for different kinds of screens (resolutions).

JavaScript has also grown beyond being used just in the browser as well. We now use it on the server thanks to **Node.js**.
But the basic fact remains: HTML, CSS, and JavaScript are the main languages of the Web.

So that's it. The languages of the Web explained in basic terms. I really hope you got something useful from this article.

Web Servers Shell:

A web shell is a shell-like interface that enables a web server to be remotely accessed, often for the purposes of cyberattacks.[1] A web shell is unique in that a web browser is used to interact with it.[2][3]

A web shell could be programmed in any programming language that is supported on a server. Web shells are most commonly written in the PHP programming language due to the widespread usage of PHP for web applications. However, Active Server Pages, ASP.NET, Python, Perl,

Ruby, and Unix shell scripts are also used, although these languages are less commonly used.[1][2][3]

Using network monitoring tools, an attacker can find vulnerabilities that can potentially allow delivery of a web shell. These vulnerabilities are often present in applications that are run on a web server.[2]

An attacker can use a web shell to issue shell commands, perform privilege escalation on the web server, and the ability to upload, delete, download, and execute files to and from the web server.[2

UNIX CLI Version control

We have various commands that help us to find out the Unix variant, type, and machine. The most common Unix command is uname, and we will talk about it first, followed by variant-specific information.

ADVERTISEMENT

Checking Unix version

1. Open the terminal application and then type the following uname command:
   ```
   uname
   uname -a
   ```
2. Display the current release level (OS Version) of the Unix operating system.
   ```
   uname -r
   ```
3. You will see Unix OS version on screen. To see architecture of Unix, run:
   ```
   uname -m
   ```

Here is outputs from my FreeBSD Unix server:

Examples

Although uname available on all Unix variants, there are other ways to display OS versions and names. Let us look at operating system-specific information.

How to check FreeBSD unix version

Type the following command to determine the version and patch level:

```
freebsd-version
freebsd-version -k
freebsd-version -r
freebsd-version -u
```

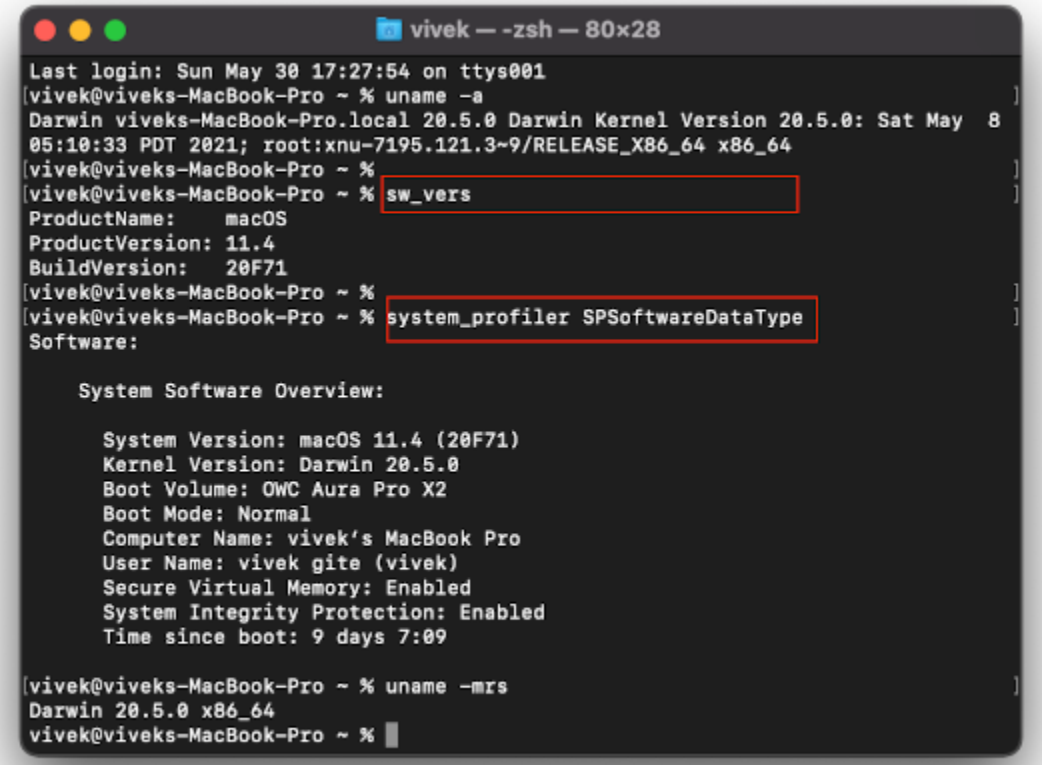
Show FreeBSD Unix Version

A note about macOS

Open the macOS Terminal app and then type any one of the following command to print macOS or Mac OS X version:

```
sw_vers
# OR #
```

```
system_profiler SPSoftwareDataType
```



HP-UX Unix

Use the swlist command as follows for determining your HP-UX Unix system version:

```
swlist
swlist | grep -i oe
swlist HPUX*OE*
swlist HPUX*OE*
```

You will see something as follows:

```
HPUX11i-OE-Ent B.11.23.0606 HP-UX Enterprise Operating Environment Component
```

OR

```
HPUX11i-TCOE B.11.23.0409 HP-UX Technical Computing OE Component
```

To see machine model from HP, type:

```
model
machinfo
getconf MACHINE_MODEL
```

Oracle or Sun Solaris OS

Verifying Operating system version on Oracle or Sun Solaris Unix is easy:

```
uname
uname -a
uname -r
# use the cat command #
cat /etc/release
```

You will get info such as `Oracle Solaris 11 (5.11)` OR `Oracle Solaris 11.1 SPARC`.

IBM AIX Unix

To view the base level of the Unix system OS from IBM, type:

```
uname
uname -a
uname -r
oslevel
prtconf
```

See oslvel AIX command man-page for more info.

Summing up

The uname and other Unix command commands can help you determine information about your Unix server or desktop, including its hardware type, machine model, operating system version. The uname and other options various. Hence, see the following man pages:

```
man uname
```

Git & Github

Introduction to Git

For installation purposes on ubuntu, you can refer to this article: How to Install, Configure and Use GIT on Ubuntu?

Git is a distributed version control system. So, What is a Version Control System?

A version Control system is a system that maintains different versions of your project when we work in a team or as an individual. (system managing changes to files) As the project progresses, new features get added to it. So, a version control system maintains all the different versions of your project for you and you can roll back to any version you want without causing any trouble to you for maintaining different versions by giving names to it like MyProject, MyProjectWithFeature1, etc.

Distributed Version control system means every collaborator(any developer working on a team project)has a local repository of the project in his/her local machine unlike central where team members should have an internet connection to every time update their work to the main central repository.

So, by distributed we mean: the project is distributed. A repository is an area that keeps all your project files, images, etc. In terms of Github: different versions of projects correspond to commits.

For more details on introduction to Github, you can refer: Introduction to Github

Git Repository Structure

It consists of 4 parts:

Working directory: This is your local directory where you make the project (write code) and make changes to it.
Staging Area (or index): this is an area where you first need to put your project before committing. This is used for code review by other team members.
Local Repository: this is your local repository where you commit changes to the project before pushing them to the central repository on Github. This is what is provided by the distributed version control system. This corresponds to the .git folder in our directory.
Central Repository: This is the main project on the central server, a copy of which is with every team member as a local repository.
All the repository structure is internal to Git and is transparent to the developer.

Some commands which relate to repository structure:

```
// transfers your project from working directory
// to staging area.
git add .

// transfers your project from staging area to
// Local Repository.
git commit -m "your message here"


// transfers project from local to central repository.
// (requires internet)
git push
```

Github

Github basically is a for-profit company owned by Microsoft, which hosts Git repositories online. It helps users share their git repository online, with other users, or access it remotely. You can also host a public repository for free on Github.

User share their repository online for various reasons including but not limited to project deployment, project sharing, open source contribution, helping out the community and many such.

Accessing Github central repository via HTTPS or SSH
Here, transfer project means transfer changes as git is very lightweight and works on changes in a project. It internally does the transfer by using Lossless Compression Techniques and transferring compressed files. Https is the default way to access Github central repository.

By git remote add origin http_url: remote means the remote central repository. Origin corresponds to your central repository which you need to define (hereby giving HTTPS URL) in order to push changes to Github.
Via SSH: connect to Linux or other servers remotely.
If you access Github by ssh you don't need to type your username and password every time you push changes to GitHub.

Terminal commands:

ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
This does the ssh key generation using RSA cryptographic algorithm.

eval "$(ssh-agent -s)" -> enable information about local login session.

ssh-add ~/.ssh/id_rsa -> add to ssh key.
cat ~/.ssh/id_rsa (use .pub file if not able to connect)
add this ssh key to github.

Now, go to github settings -> new ssh key -> create key

ssh -T git@github.com -> activate ssh key (test connection)

Refresh your github Page.
Working with git – Important Git commands
Git user configuration (First Step)

git --version (to check git version)
git config --global user.name "your name here"
git config --global user.email "your email here"
These are the information attached to commits.

Initialize directory

git init
initializes your directory to work with git and makes a local repository. .git folder is made (OR)

git clone http_url
This is done if we have an existing git repository and we want to copy its content to a new place.

Connecting to the remote repository

git remote add origin http_url/ssh_url
connect to the central repo to push/pull. pull means adopting the changes on the remote repository to your local repository. push merges the changes from your local repository to the remote repository.

git pull origin master
One should always first pull contents from the central repo before pushing so that you are updated with other team members' work. It helps prevent merge conflicts. Here, master means the master branch (in Git).

Stash Area in git

git stash
Whichever files are present in the staging area, it will move that files to stash before committing it.

git stash pop
Whenever we want files for commit from stash we should use this command.

git stash clear
By doing this, all files from stash area is been deleted.

Steps to add a file to a remote Repository:

First, your file is in your working directory, Move it to the staging area by typing:

git add -A (for all files and folders)
#To add all files only in the current directory
git add .
git status: here, untracked files mean files that you haven't added to the staging area. Changes are not staged for commit means you have staged the file earlier than you have made changes in that files in your working directory and the changes need to be staged once more. Changes ready to be committed: these are files that have been committed and are ready to be pushed to the central repository.

git commit -a -m "message for commit"
-a: commit all files and for files that have been
staged earlier need not to be git add once more
-a option does that automatically.
git push origin master -> pushes your files to

github master branch
git push origin anyOtherBranch -> pushes any
other branch to github.
git log ; to see all your commits
git checkout commitObject(first 8 bits) file.txt->
revert back to this previous commit for file file.txt
Previous commits m=ight be seen through the git log command.

HEAD -> pointer to our latest commit.
Ignoring files while committing

In many cases, the project creates a lot of logs and other irrelevant files which are to be ignored. So to ignore those files, we have to put their names in“.gitignore” file.

touch .gitignore
echo "filename.ext" >>.gitignore
#to ignore all files with .log extension
echo "*.log" > .gitignore
Now the filenames written in the .gitignore file would be ignored while pushing a new commit. To get the changes between commits, commit, and working tree.

git diff
‘git diff’ command compares the staging area with the working directory and tells us the changes made. It compares the earlier information as well as the current modified information.

Branching in Git

create branch ->
git branch myBranch
or

git checkout -b myBranch -> make and switch to the branch myBranch

Do the work in your branch. Then,

git checkout master ; to switch back to master branch

Now, merge contents with your myBranch By:

git merge myBranch (writing in master branch)

This merger makes a new commit.

Another way

git rebase myBranch

This merges the branch with the master in a serial fashion. Now,

git push origin master

Contributing to Open Source

Open Source might be considered as a way where user across the globe may share their opinions, customizations or work together to solve an issue or to complete the desired project together. Many companies host there repositories online on Github to allow access to developers to make changes to their product. Some companies(not necessarily all) rewards their contributors in different ways.

You can contribute to any open source project on Github by forking it, making desired changes to the forked repository, and then opening a pull request. The project owner will review your project and will ask to improve it or will merge it.

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.

Javascript basics OOPS:

As JavaScript is widely used in Web Development, in this article we will explore some of the **Object Oriented** mechanisms supported by **JavaScript** to get the most out of it. Some of the common interview questions in JavaScript on OOPS include:

- How is Object-Oriented Programming implemented in JavaScript?
- How does it differ from other languages?
- Can you implement Inheritance in JavaScript?

and so on…

There are certain features or mechanisms which make a Language Object-Oriented like:

OOPs Concept in JavaScript

Object	Classes	Encapsulation
Abstraction	Inheritance	Polymorphism

Let's dive into the details of each one of them and see how they are implemented in JavaScript.

Object: An Object is a **unique** entity that contains **properties** and **methods**. For example "a car" is a real-life Object, which has some characteristics like color, type, model, and horsepower and performs certain actions like driving. The characteristics

of an Object are called Properties in Object-Oriented Programming and the actions are called methods. An Object is an **instance** of a class. Objects are everywhere in JavaScript, almost every element is an Object whether it is a function, array, or string.

Note: A Method in javascript is a property of an object whose value is a function.

The object can be created in two ways in JavaScript:

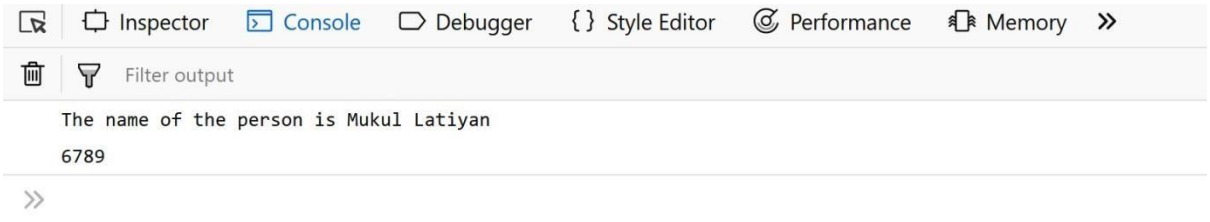
- **Object Literal**
- **Object Constructor**

Example: Using an Object Literal.

- Javascript

```
// Defining object
let person = {
    first_name:'Mukul',
    last_name: 'Latiyan',

    //method
    getFunction : function(){
        return (`The name of the person is
          ${person.first_name} ${person.last_name}`)
    },
    //object within object
    phone_number : {
        mobile:'12345',
        landline:'6789'
    }
}
console.log(person.getFunction());
console.log(person.phone_number.landline);
```

Output:

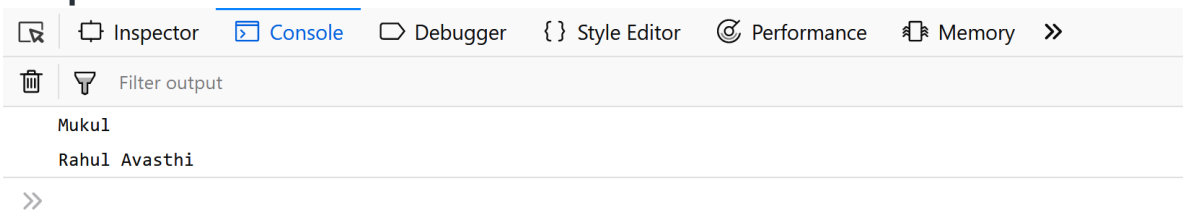


Example: Using an Object Constructor.

- Javascript

```
// Using a constructor
function person(first_name,last_name){
   this.first_name = first_name;
   this.last_name = last_name;
}
// Creating new instances of person object
let person1 = new person('Mukul','Latiyan');
let person2 = new person('Rahul','Avasthi');

console.log(person1.first_name);
console.log(`${person2.first_name} ${person2.last_name}`);
```

Output:



Note: The JavaScript Object.create() Method creates a new object, using an existing object as the prototype of the newly created object.

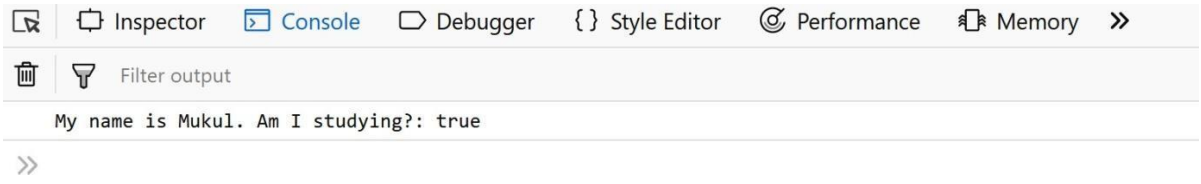
Example:

- Javascript

```
// Object.create() example a
// simple object with some properties
const coder = {
    isStudying : false,
    printIntroduction : function(){
        console.log(`My name is ${this.name}. Am I
          studying?: ${this.isStudying}.`)
    }
}
// Object.create() method
const me = Object.create(coder);

// "name" is a property set on "me", but not on "coder"
me.name = 'Mukul';

// Inherited properties can be overwritten
me.isStudying = true;
```

```
me.printIntroduction();
```

Output:



Classes: Classes are **blueprints** of an Object. A class can have many Objects because the class is a **template** while Objects are **instances** of the class or the concrete implementation.
Before we move further into implementation, we should know unlike other Object Oriented languages there are **no classes in JavaScript** we have only Object. To be more precise, JavaScript is a prototype-based Object Oriented Language, which means it doesn't have classes, rather it defines behaviors using a constructor function and then reuses it using the prototype.
Note: Even the classes provided by ECMA2015 are objects.
JavaScript classes, introduced in ECMAScript 2015, are primarily syntactical sugar over JavaScript's existing prototype-based inheritance. The class syntax is not introducing a new object-oriented inheritance model to JavaScript. JavaScript classes provide a much simpler and clearer syntax to create objects and deal with inheritance.

-Mozilla Developer Network

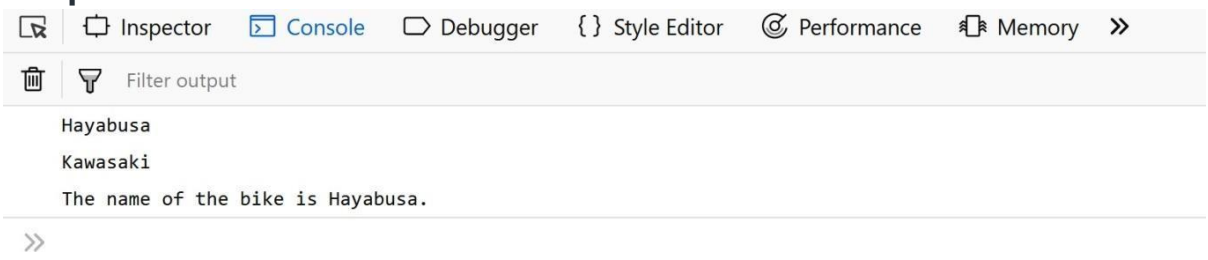
Example: Let's use ES6 classes then we will look at the traditional way of defining an Object and simulate them as classes.

- Javascript

```
// Defining class using es6
class Vehicle {
  constructor(name, maker, engine) {
    this.name = name;
    this.maker = maker;
    this.engine = engine;
  }
  getDetails(){
      return (`The name of the bike is ${this.name}.`)
```

```
  }
}
// Making object with the help of the constructor
let bike1 = new Vehicle('Hayabusa', 'Suzuki', '1340cc');
let bike2 = new Vehicle('Ninja', 'Kawasaki', '998cc');

console.log(bike1.name);   //   Hayabusa
console.log(bike2.maker);  //  Kawasaki
console.log(bike1.getDetails());
```

Output:



Example: Traditional Way of defining an Object and simulating them as classes.

- Javascript

```
// Defining class in a Traditional Way.
function Vehicle(name,maker,engine){
    this.name = name,
    this.maker = maker,
    this.engine = engine
};

Vehicle.prototype.getDetails = function(){
    console.log('The name of the bike is '+ this.name);
}

let bike1 = new Vehicle('Hayabusa','Suzuki','1340cc');
let bike2 = new Vehicle('Ninja','Kawasaki','998cc');

console.log(bike1.name);
console.log(bike2.maker);
console.log(bike1.getDetails());
```

Output:

As seen in the above example it is much simpler to define and reuse objects in ES6. Hence, we would be using ES6 in all of our examples.

Abstraction: Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

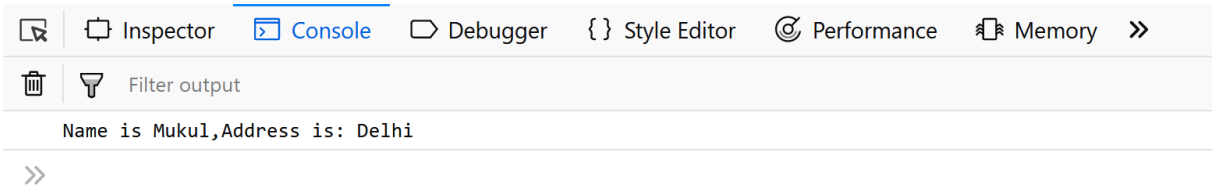
Encapsulation: The process of **wrapping properties and functions** within a **single unit** is known as encapsulation.

Example: Let's understand encapsulation with an example.

- Javascript

```
// Encapsulation example
class person{
    constructor(name,id){
        this.name = name;
        this.id = id;
    }
    add_Address(add){
        this.add = add;
    }
    getDetails(){
        console.log(`Name is ${this.name},
        Address is: ${this.add}`);
    }
}

let person1 = new person('Mukul',21);
person1.add_Address('Delhi');
person1.getDetails();
```

Output: In this example, we simply create a person Object using the constructor, Initialize its properties and use its functions. We are not bothered by the implementation details. We are working with an Object's interface without considering the implementation details.



S

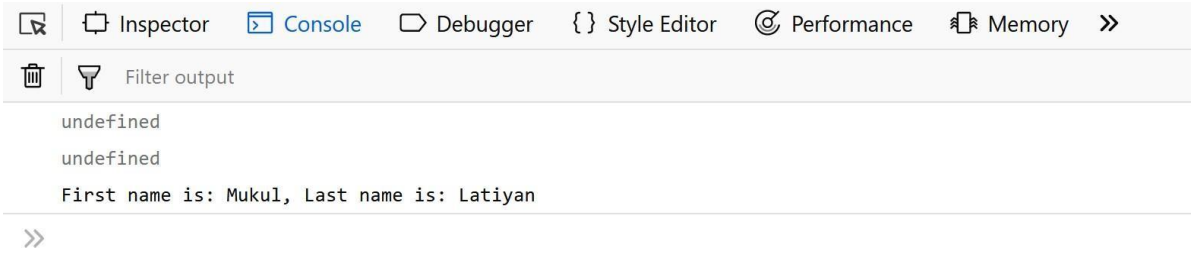
ometimes encapsulation refers to the **hiding of data** or **data Abstraction** which means representing essential features hiding the background detail. Most of the OOP languages provide access modifiers to restrict the scope of a variable, but there are no such access modifiers in JavaScript, there are certain ways by which we can restrict the scope of variables within the Class/Object.

Example:

- Javascript

```
// Abstraction example
function person(fname,lname){
    let firstname = fname;
    let lastname = lname;

    let getDetails_noaccess = function(){
        return (`First name is: ${firstname} Last
            name is: ${lastname}`);
    }

    this.getDetails_access = function(){
        return (`First name is: ${firstname}, Last
            name is: ${lastname}`);
    }
}
let person1 = new person('Mukul','Latiyan');
console.log(person1.firstname);
console.log(person1.getDetails_noaccess);
console.log(person1.getDetails_access());
```

Output: In this example, we try to access some property(person1.firstname) and functions(person1.getDetails_noaccess) but it returns undefined while there is a method that we can access from the person object(person1.getDetails_access()). By changing the way we define a function we can restrict its scope.



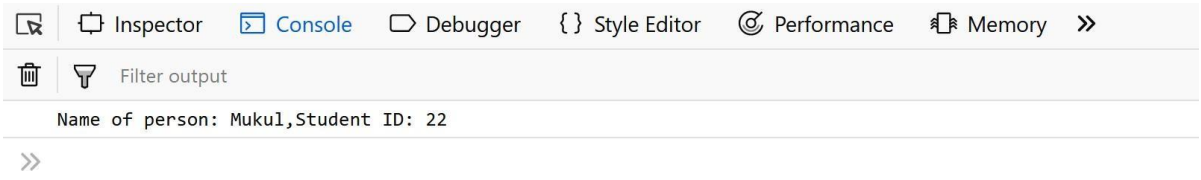
Inheritance: It is a concept in which some properties and methods of an Object are being used by another Object. Unlike most of the OOP languages where classes inherit classes, JavaScript Objects inherit Objects i.e. certain features (property and methods) of one object can be reused by other Objects.
Example: Let's understand inheritance and polymorphism with an example.

- Javascript

```
// Inheritance example
class person{
    constructor(name){
        this.name = name;
    }
    // method to return the string
    toString(){
        return (`Name of person: ${this.name}`);
    }
}
class student extends person{
    constructor(name,id){
        // super keyword for calling the above
        // class constructor
        super(name);
        this.id = id;
    }
    toString(){
        return (`${super.toString()},
        Student ID: ${this.id}`);
    }
}
let student1 = new student('Mukul',22);
console.log(student1.toString());
```

Output: In this example, we define a Person Object with certain properties and methods and then we inherit the Person Object in the

Student Object and use all the properties and methods of the person Object as well as define certain properties and methods for the Student Object.



Note: The Person and Student objects both have the same method (i.e toString()), this is called **Method Overriding**. Method Overriding allows a method in a child class to have the same name(polymorphism) and method signature as that of a parent class.
In the above code, the super keyword is used to refer to the immediate parent class's instance variable.

Polymorphism: Polymorphism is one of the core concepts of object-oriented programming languages. Polymorphism means the same function with different signatures is called many times. In real life, for example, a boy at the same time may be a student, a class monitor, etc. So a boy can perform different operations at the same time. Polymorphism can be achieved by method overriding and method overloading

Functions in JS AJAX

What is Ajax?

Ajax stands for Asynchronous Javascript And Xml. Ajax is just a means of loading data from the server and selectively updating parts of a web page without reloading the whole page.

Basically, what Ajax does is make use of the browser's built-in XMLHttpRequest (XHR) object to send and receive information to and from a web server asynchronously, in the background, without blocking the page or interfering with the user's experience.

Ajax has become so popular that you hardly find an application that doesn't use Ajax to some extent. The example of some large-scale Ajax-driven online applications are: Gmail, Google Maps, Google Docs, YouTube, Facebook, Flickr, and so many other applications.

Note: Ajax is not a new technology, in fact, Ajax is not even really a technology at all. Ajax is just a term to describe the process of exchanging data from a web server asynchronously through JavaScript, without refreshing the page.

Tip: Don't get confused by the term X (i.e. XML) in AJAX. It is only there for historical reasons. Other data exchange format such as JSON, HTML, or plain text can be used instead of XML.

Understanding How Ajax Works

To perform Ajax communication JavaScript uses a special object built into the browser–an XMLHttpRequest (XHR) object–to make HTTP requests to the server and receive data in response.

All modern browsers (Chrome, Firefox, IE7+, Safari, Opera) support the XMLHttpRequest object.

The following illustrations demonstrate how Ajax communication works:

Ajax Illustration

Since Ajax requests are usually asynchronous, execution of the script continues as soon as the Ajax request is sent, i.e. the browser will not halt the script execution until the server response comes back.

In the following section we'll discuss each step involved in this process one by one:

Sending Request and Retrieving the Response

Before you perform Ajax communication between client and server, the first thing you must do is to instantiate an XMLHttpRequest object, as shown below:

```
var request = new XMLHttpRequest();
```

Now, the next step in sending the request to the server is to instantiating the newly-created request object using the open() method of the XMLHttpRequest object.

The open() method typically accepts two parameters– the HTTP request method to use, such as "GET", "POST", etc., and the URL to send the request to, like this:

```
request.open("GET", "info.txt"); -Or- request.open("POST", "add-user.php");
```

Tip: The file can be of any kind, like .txt or .xml, or server-side scripting files, like .php or .asp, which can perform some actions on the server (e.g. inserting or reading data from database) before sending the response back to the client.

And finally send the request to the server using the send() method of the XMLHttpRequest object.

request.send(); -Or- request.send(body);

Note: The send() method accepts an optional body parameter which allow us to specify the request's body. This is primarily used for HTTP POST requests, since the HTTP GET request doesn't have a request body, just request headers.

The GET method is generally used to send small amount of data to the server. Whereas, the POST method is used to send large amount of data, such as form data.

In GET method, the data is sent as URL parameters. But, in POST method, the data is sent to the server as a part of the HTTP request body. Data sent through POST method will not visible in the URL.

See the chapter on HTTP GET vs. POST for a detailed comparison of these two methods.

In the following section we'll take a closer look at how Ajax requests actually works.

Performing an Ajax GET Request

The GET request is typically used to get or retrieve some kind of information from the server that doesn't require any manipulation or change in database, for example, fetching search results based on a term, fetching user details based on their id or name, and so on.

The following example will show you how to make an Ajax GET request in JavaScript.

ExampleTry this code »

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>JavaScript Ajax GET Demo</title>
<script>
function displayFullName() {
    // Creating the XMLHttpRequest object
    var request = new XMLHttpRequest();

    // Instantiating the request object
    request.open("GET", "greet.php?fname=John&lname=Clark");

    // Defining event listener for readystatechange event
    request.onreadystatechange = function() {
        // Check if the request is compete and was successful
        if(this.readyState === 4 && this.status === 200) {
```

```
            // Inserting the response from server into an HTML element

            document.getElementById("result").innerHTML =
this.responseText;

        }

    };



    // Sending the request to the server

    request.send();

}

</script>

</head>

<body>

    <div id="result">

        <p>Content of the result DIV box will be replaced by the server
response</p>

    </div>

    <button type="button" onclick="displayFullName()">Display Full
Name</button>

</body>

</html>
```

When the request is asynchronous, the send() method returns immediately after sending the request. Therefore you must check where the response currently stands in its lifecycle before processing it using the readyState property of the XMLHttpRequest object.

The readyState is an integer that specifies the status of an HTTP request. Also, the function assigned to the onreadystatechange event handler called every time the readyState property changes. The possible values of the readyState property are summarized below.

Value State Description

0 UNSENT An XMLHttpRequest object has been created, but the open() method hasn't been called yet (i.e. request not initialized).

1 OPENED The open() method has been called (i.e. server connection established).

2 HEADERS_RECEIVED The send() method has been called (i.e. server has received the request).

3 LOADING The server is processing the request.

4 DONE The request has been processed and the response is ready.

Note: Theoretically, the readystatechange event should be triggered every time the readyState property changes. But, most browsers do not fire this event when readyState changes to 0 or 1. However, all browsers fire this event when readyState changes to 4 .

The status property returns the numerical HTTP status code of the XMLHttpRequest's response. Some of the common HTTP status codes are listed below:

200 – OK. The server successfully processed the request.

404 – Not Found. The server can't find the requested page.

503 – Service Unavailable. The server is temporarily unavailable.

Please check out the HTTP status codes reference for a complete list of response codes.

Here's the code from our "greet.php" file that simply creates the full name of a person by joining their first name and last name and outputs a greeting message.

ExampleDownload

```
<?php
if(isset($_GET["fname"]) && isset($_GET["lname"])) {
    $fname = htmlspecialchars($_GET["fname"]);
    $lname = htmlspecialchars($_GET["lname"]);

    // Creating full name by joining first and last name
    $fullname = $fname . " " . $lname;

    // Displaying a welcome message
    echo "Hello, $fullname! Welcome to our website.";
} else {
    echo "Hi there! Welcome to our website.";
}
?>
```

Performing an Ajax POST Request

The POST method is mainly used to submit a form data to the web server.

The following example will show you how to submit form data to the server using Ajax.

ExampleTry this code »

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>JavaScript Ajax POST Demo</title>
<script>
function postComment() {
    // Creating the XMLHttpRequest object
    var request = new XMLHttpRequest();

    // Instantiating the request object
    request.open("POST", "confirmation.php");

    // Defining event listener for readystatechange event
    request.onreadystatechange = function() {
```

```
        // Check if the request is compete and was successful
        if(this.readyState === 4 && this.status === 200) {
            // Inserting the response from server into an HTML element
            document.getElementById("result").innerHTML =
this.responseText;
        }
    };

    // Retrieving the form data
    var myForm = document.getElementById("myForm");
    var formData = new FormData(myForm);

    // Sending the request to the server
    request.send(formData);
}
</script>
</head>
<body>
    <form id="myForm">
        <label>Name:</label>
        <div><input type="text" name="name"></div>
        <br>
```

```
        <label>Comment:</label>

        <div><textarea name="comment"></textarea></div>

        <p><button type="button" onclick="postComment()">Post
Comment</button></p>

    </form>

    <div id="result">

        <p>Content of the result DIV box will be replaced by the server
response</p>

    </div>

</body>

</html>
```

If you are not using the FormData object to send form data, for example, if you're sending the form data to the server in the query string format, i.e. request.send(key1=value1&key2=value2) then you need to explicitly set the request header using setRequestHeader() method, like this:

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

The setRequestHeader() method, must be called after calling open(), but before calling send().

Some common request headers are: application/x-www-form-urlencoded, multipart/form-data, application/json, application/xml, text/plain, text/html, and so on.

Note: The FormData object provides an easy way to construct a set of key/value pairs representing form fields and their values which can be sent using XMLHttpRequest.send() method. The transmitted data is in the same format that the form's submit() method would use to send the data if the form's encoding type were set to multipart/form-data.

Here's the code of our "confirmation.php" file that simply outputs the values submitted by the user.

ExampleDownload

```
<?php
if($_SERVER["REQUEST_METHOD"] == "POST") {
    $name = htmlspecialchars(trim($_POST["name"]));
    $comment = htmlspecialchars(trim($_POST["comment"]));

    // Check if form fields values are empty
    if(!empty($name) && !empty($comment)) {
        echo "<p>Hi, <b>$name</b>. Your comment has been received successfully.<p>";
        echo "<p>Here's the comment that you've entered: <b>$comment</b></p>";
    } else {
        echo "<p>Please fill all the fields in the form!</p>";
    }
} else {
```

```
    echo "<p>Something went wrong. Please try again.</p>";
}
?>
```

For security reasons, browsers do not allow you to make cross-domain Ajax requests. This means you can only make Ajax requests to URLs from the same domain as the original page, for example, if your application is running on the domain "mysite.com", you cannot make Ajax request to "othersite.com" or any other domain. This is commonly known as same origin policy.

JSON data format.

What is JSON?

- JSON stands for **J**ava**S**cript **O**bject **N**otation
- JSON is a lightweight data-interchange format
- JSON is plain text written in JavaScript object notation
- JSON is used to send data between computers
- JSON is language independent *

*
The JSON syntax is derived from JavaScript object notation, but the JSON format is text only.

Code for reading and generating JSON exists in many programming languages.

The JSON format was originally specified by Douglas Crockford.

Why Use JSON?

The JSON format is syntactically similar to the code for creating JavaScript objects. Because of this, a JavaScript program can easily convert JSON data into JavaScript objects.

Since the format is text only, JSON data can easily be sent between computers, and used by any programming language.

JavaScript has a built in function for converting JSON strings into JavaScript objects:

```
JSON.parse()
```

JavaScript also has a built in function for converting an object into a JSON string:

```
JSON.stringify()
```

You can receive pure text from a server and use it as a JavaScript object.

You can send a JavaScript object to a server in pure text format.

You can work with data as JavaScript objects, with no complicated parsing and translations.

Storing Data

When storing data, the data has to be a certain format, and regardless of where you choose to store it, *text* is always one of the legal formats.

JSON makes it possible to store JavaScript objects as text.

JSON Example

This example is a JSON string:

```
'{"name":"John", "age":30, "car":null}'
```

It defines an object with 3 properties:

- name
- age
- car

Each property has a value.

If you parse the JSON string with a JavaScript program, you can access the data as an object:

```
let personName = obj.name;
let personAge = obj.age;
```

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

Introduction to ReactJS

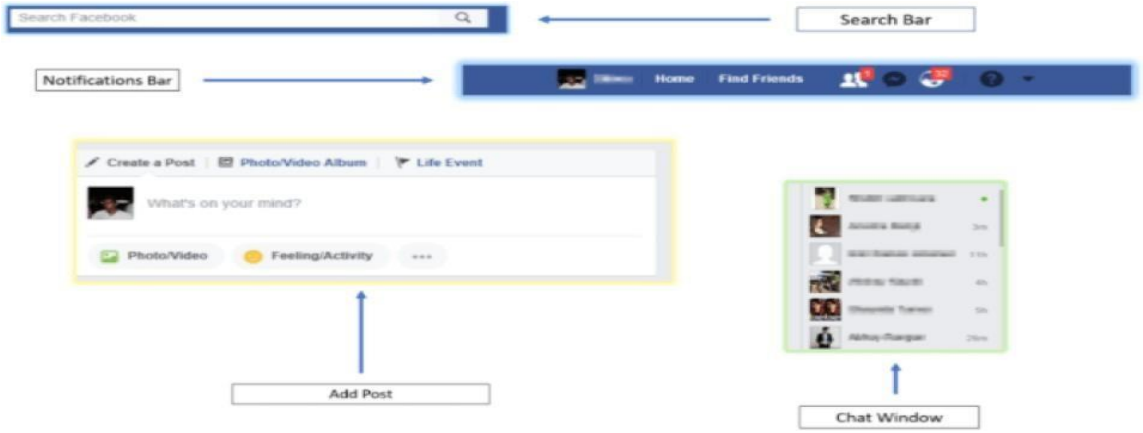
React is a popular JavaScript library used for web development. React.js or ReactJS or React are different ways to represent ReactJS. Today's many large-scale companies (Netflix, Instagram, to name a few) also use React JS. There are many advantages of using this framework over other frameworks, and It's ranking under the top 10 programming languages for the last few years under various language ranking indices.

What is ReactJS?

React.js is a front-end JavaScript framework developed by Facebook. To build composable user interfaces predictably and efficiently using declarative code, we use React. It's an open-source and component-based framework responsible for creating the application's view layer.

- ReactJs follows the Model View Controller (MVC) architecture, and the view layer is accountable for handling mobile and web apps.
- React is famous for building single-page applications and mobile apps.

Let's take an example: Look at the Facebook page, which is entirely built on React, to understand how react does works.



As the figure shows, ReactJS divides the UI into multiple components, making the code easier to debug. This way, each function is assigned to a specific component, and it produces some HTML which is rendered as output by the DOM.

ReactJS History:

Jordan Walke created React, who worked as a software engineer in Facebook has first released an early React prototype called "FaxJS."

In 2011, React was first deployed on Facebook's News Feed, and later in 2012 on Instagram.

As the figure shows, ReactJS divides the UI into multiple components, making the code easier to debug. This way, each function is assigned to a specific component, and it produces some HTML which is rendered as output by the DOM.

ReactJS History:

Jordan Walke created React, who worked as a software engineer in Facebook has first released an early React prototype called "FaxJS."

In 2011, React was first deployed on Facebook's News Feed, and later in 2012 on Instagram.

The current version of React.JS is V17.0.1.

Why do people choose to program with React?

There are various reasons why you should choose ReactJS as a primary tool for website UI development. Here, we highlight the most notable ones and explain why these specifics are so important:

- **Fast** - Feel quick and responsive through the Apps made in React can handle complex updates.
- **Modular** - Allow you to write many smaller, reusable files instead of writing large, dense files of code. The modularity of React is an attractive solution for JavaScript's visibility issues.
- **Scalable** - React performs best in the case of large programs that display a lot of data changes.
- **Flexible** - React approaches differently by breaking them into components while building user interfaces. This is incredibly important in large applications.
- **Popular** - ReactJS gives better performance than other JavaScript languages due to t's implementation of a virtual DOM.
- **Easy to learn** - Since it requires minimal understanding of HTML and JavaScript, the learning curve is low.
- **Server-side rendering and SEO friendly** - ReactJS websites are famous for their server-side rendering feature. It makes apps faster and much better for search engine ranking in comparison to products with client-side rendering. React even produces more opportunities for website SEO and can occupy higher positions on the search result's page.
- **Reusable UI components** - React improves development and debugging processes.
- **Community** - The number of tools and extensions available for ReactJS developers is tremendous. Along with impressive out-of-box functionalities, more opportunities emerge once you discover how giant the React galaxy is. React has a vibrant community and is supported by Facebook. Hence, it's a reliable tool for website development.

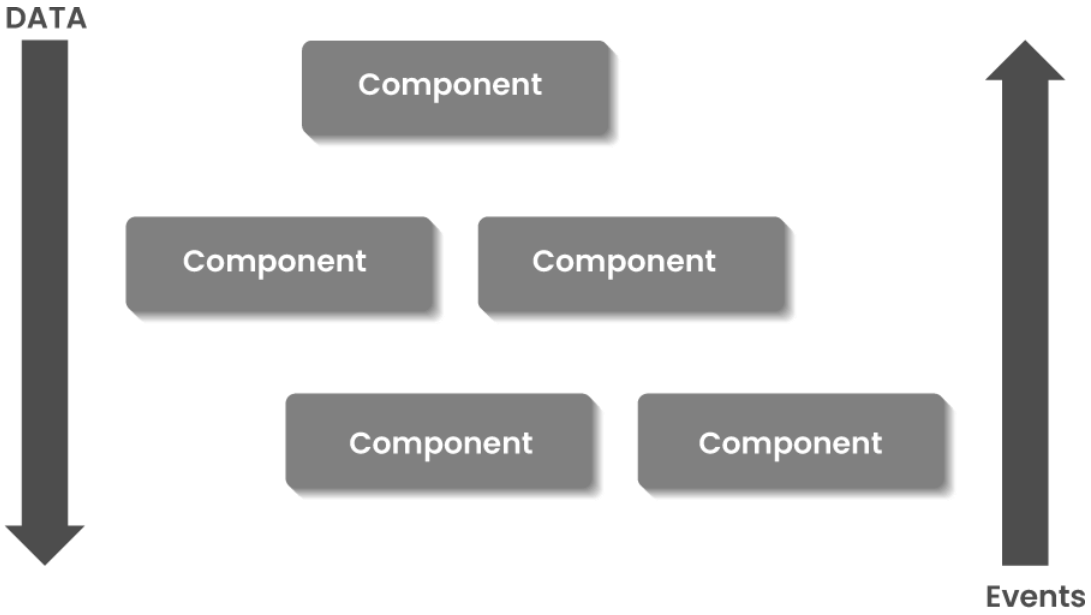
ReactJS Features:

1. JSX - JavaScript Syntax Extension

JSX is a preferable choice for many web developers. It isn't necessary to use JSX in React development, but there is a massive difference between writing react.js documents in JSX and JavaScript. JSX is a syntax extension to JavaScript. By using that, we can write HTML structures in the same file that contains JavaScript code.

2. Unidirectional Data Flow and Flux

React.js is designed so that it will only support data that is flowing downstream, in one direction. If the data has to flow in another direction, you will need additional features.

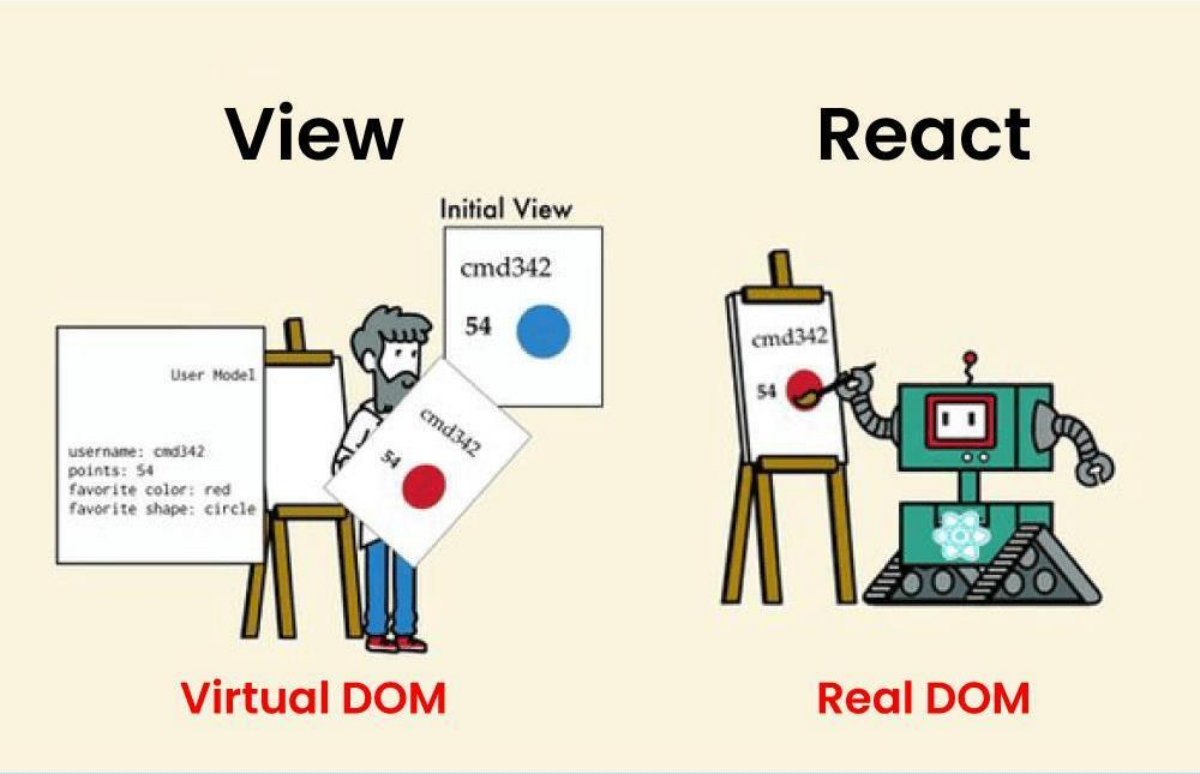


React contains a set of immutable values passed to the component renderer as properties in HTML tags. The components cannot modify any properties directly but support a call back function to do modifications.

3. Virtual Document Object Model (VDOM)

React contains a lightweight representation of real DOM in the memory called Virtual DOM. Manipulating real DOM is much slower compared to VDOM as nothing gets drawn on the screen. When any object's state changes, VDOM modifies only that object in real DOM instead of updating whole objects.

That makes things move fast, particularly compared with other front-end technologies that have to update each object even if only a single object changes in the web application.



4. Extensions

React supports various extensions for application architecture. It supports server-side rendering, Flux, and Redux extensively in web app development. React Native is a popular framework developed from React for creating cross-compatible mobile apps.

5. Debugging

Testing React apps is easy due to large community support. Even Facebook provides a small browser extension that makes React debugging easier and faster.

Next, let's understand some essential concepts of ReactJS.

Building Components of React - Components, State, Props, and Keys.

1. ReactJS Components

Components are the heart and soul of React. Components (like JavaScript functions) let you split the UI into independent, reusable pieces and think about each piece in isolation.

Components are building blocks of any React application. Every component has its structures, APIs, and methods.

In React, there are two types of components, namely stateless functional and stateful class.

- Functional Components - These components have no state of their own and contain only a render method. They are simply Javascript functions that may or may not receive data as parameters.

Stateless functional components may derive data from other components as properties (props).

An example of representing functional component is shown below:

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

- Class Components - These components are more complex than functional components. They can manage their state and to return JSX on the screen have a separate render method. You can pass data from one class to other class components.

An example of representing class component is shown below:

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

2. React State

A state is a place from where the data comes. The state in a component can change over time, and whenever it changes, the component re-renders.

A change in a state can happen as a response to system-generated events or user action, and these changes define the component's behavior and how it will render.

```
class Greetings extends React.Component {
  state = {
    name: "World"
  };
  updateName() {
    this.setState({ name: "Mindmajix" });
  }
  render() {
      return(
          <div>
              {this.state.name}
          </div>
      )
  }
}
```

The state object is initialized in the constructor, and it can store multiple properties.

For changing the state object value, use this.setState() function.

To perform a merge between the new and the previous state, use the setState() function.

3. React Props

Props stand for properties, and they are read-only components.

Both Props and State are plain JavaScript objects and hold data that influence the output of render. And they are different in one way: State is managed within the component (like variable declaration within a function), whereas props get passed to the component (like function parameters).

Props are immutable, and this is why the component of a container should describe the state that can be changed and updated, while the child components should only send data from the state using properties.

4. React Keys

In React, Keys are useful when working with dynamically created components. Setting the key value will keep your component uniquely identified after the change.

They help React in identifying the items which have changed, are removed, or added.

In summary, State, Props, keys, and components are the few fundamental React concepts that you need to be familiar with before working on it.

React Router

Routing is a process in which a user is directed to different pages based on their action or request. ReactJS Router is mainly used for developing Single Page Web Applications. React Router is used to define multiple routes in the application. When a user types a specific URL into the browser, and if this URL path matches any 'route' inside the router file, the user will be redirected to that particular route.

React Router is a standard library system built on top of the React and used to create routing in the React application using React Router Package. It provides the synchronous URL on the browser with data that will be displayed on the web page. It maintains the standard structure and behavior of the application and mainly used for developing single page web applications.

Need of React Router

React Router plays an important role to display multiple views in a single page application. Without React Router, it is not possible to display multiple views in React applications. Most of the social media websites like Facebook, Instagram uses React Router for rendering multiple views.

React Router Installation

React contains three different packages for routing. These are:

1. **react-router:** It provides the core routing components and functions for the React Router applications.
2. **react-router-native:** It is used for mobile applications.
3. **react-router-dom:** It is used for web applications design.

It is not possible to install react-router directly in your application. To use react routing, first, you need to install react-router-dom modules in your application. The below command is used to install react router dom.

1. $ npm install react-router-dom --save

Components in React Router

There are two types of router components:

- **<BrowserRouter>:** It is used for handling the dynamic URL.
- **<HashRouter>:** It is used for handling the static request.

Example

Step-1: In our project, we will create two more components along with **App.js**, which is already present.

About.js

```
import React from 'react'
class About extends React.Component {
  render() {
    return <h1>About</h1>
  }
}
export default About
```

```
import React from 'react'

class About extends React.Component {

  render() {

    return <h1>About</h1>

  }

}

export default About
```

Contact.js

```
import React from 'react'

class Contact extends React.Component {

  render() {

    return <h1>Contact</h1>

  }

}

export default Contact
```

App.js

```
import React from 'react'

class App extends React.Component {

  render() {

    return (

      <div>

        <h1>Home</h1>

      </div>

    )

  }

}

export default App
```

Step-2: For Routing, open the index.js file and import all the three component files in it. Here, you need to import line: **import { Route, Link, BrowserRouter as Router } from 'react-router-dom'** which helps us to implement the Routing. Now, our index.js file looks like below.

What is Route?

It is used to define and render component based on the specified path. It will accept components and render to define what should be rendered.

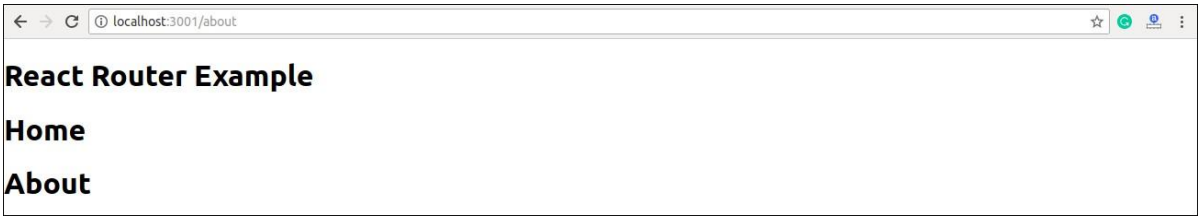
Index.js

```
import React from 'react';

import ReactDOM from 'react-dom';

import { Route, Link, BrowserRouter as Router } from 'react-router-dom'

import './index.css';
```

```
import App from './App';

import About from './about'

import Contact from './contact'


const routing = (

  <Router>

    <div>

      <h1>React Router Example</h1>

      <Route path="/" component={App} />

      <Route path="/about" component={About} />

      <Route path="/contact" component={Contact} />

    </div>

  </Router>

)

ReactDOM.render(routing, document.getElementById('root'));
```

Step-3: Open **command prompt**, go to your project location, and then type **npm start**. You will get the following screen.

Now, if you enter **manually** in the browser: **localhost:3000/about**, you will see **About** component is rendered on the screen.



Step-4: In the above screen, you can see that **Home** component is still rendered. It is because the home path is '/' and about path is '**/about**', so you can observe that **slash** is common in both paths which render both components. To stop this behavior, you need to use the **exact** prop. It can be seen in the below example.

Index.js

```
import React from 'react';

import ReactDOM from 'react-dom';

import { Route, Link, BrowserRouter as Router } from 'react-router-dom'

import './index.css';

import App from './App';

import About from './about'

import Contact from './contact'


const routing = (

  <Router>

    <div>

      <h1>React Router Example</h1>

      <Route exact path="/" component={App} />

      <Route path="/about" component={About} />

      <Route path="/contact" component={Contact} />
```

```
    </div>
  </Router>
)

ReactDOM.render(routing, document.getElementById('root'));
```

Output

Adding Navigation using Link component

Sometimes, we want to need **multiple** links on a single page. When we click on any of that particular **Link**, it should load that page which is associated with that path without **reloading** the web page. To do this, we need to import **<Link>** component in the **index.js** file.

What is < Link> component?

This component is used to create links which allow to **navigate** on different **URLs** and render its content without reloading the webpage.

Example

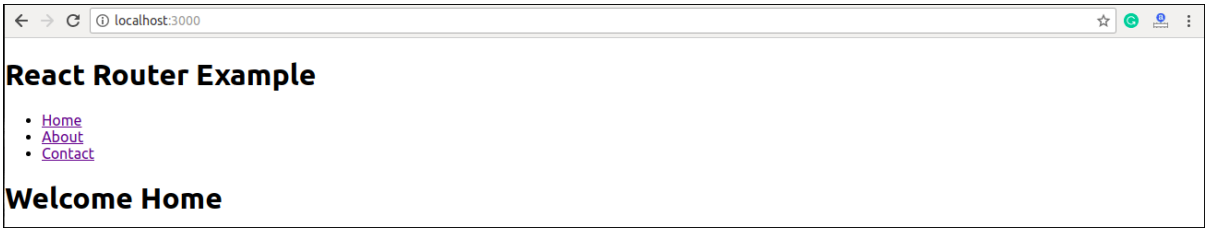
Index.js

```
import React from 'react';

import ReactDOM from 'react-dom';

import { Route, Link, BrowserRouter as Router } from 'react-router-dom'

import './index.css';

import App from './App';

import About from './about'

import Contact from './contact'
```

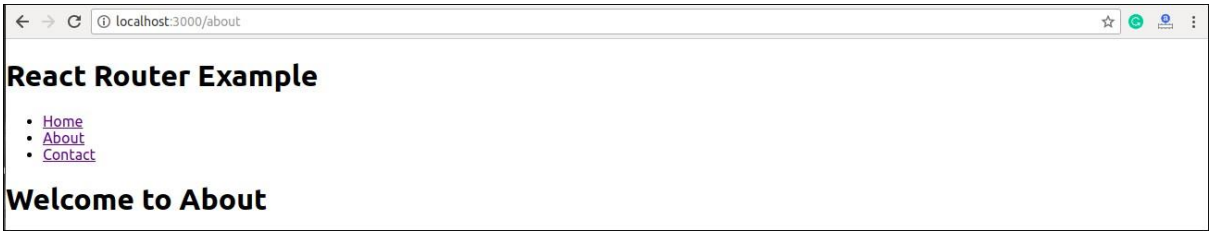
```
const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <ul>
        <li>
          <Link to="/">Home</Link>
        </li>
        <li>
          <Link to="/about">About</Link>
        </li>
        <li>
          <Link to="/contact">Contact</Link>
        </li>
      </ul>
      <Route exact path="/" component={App} />
      <Route path="/about" component={About} />
      <Route path="/contact" component={Contact} />
    </div>
  </Router>
```

```
)

ReactDOM.render(routing, document.getElementById('root'));
```

Output



After adding Link, you can see that the routes are rendered on the screen. Now, if you click on the **About**, you will see URL is changing and About component is rendered.



Now, we need to add some **styles** to the Link. So that when we click on any particular link, it can be easily **identified** which Link is **active**. To do this react router provides a new trick **NavLink** instead of **Link**. Now, in the index.js file, replace Link from Navlink and add properties **activeStyle**. The activeStyle properties mean when we click on the Link, it should have a specific style so that we can differentiate which one is currently active.

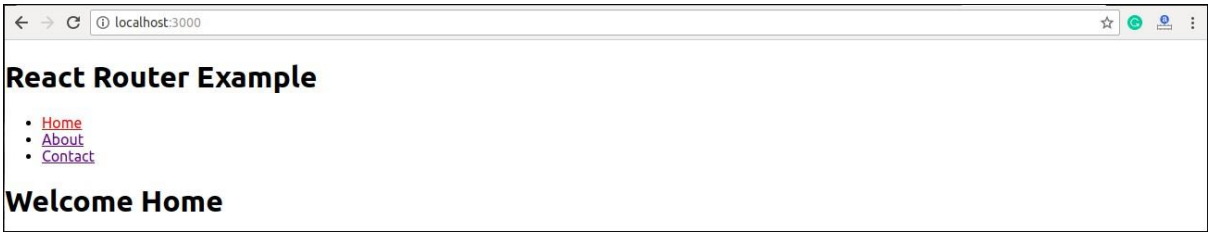
```
import React from 'react';

import ReactDOM from 'react-dom';

import { BrowserRouter as Router, Route, Link, NavLink } from 'react-router-dom'

import './index.css';

import App from './App';

import About from './about'

import Contact from './contact'
```

```
const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <ul>
        <li>
          <NavLink to="/" exact activeStyle={
            {color:'red'}
          }>Home</NavLink>
        </li>
        <li>
          <NavLink to="/about" exact activeStyle={
            {color:'green'}
          }>About</NavLink>
        </li>
        <li>
          <NavLink to="/contact" exact activeStyle={
            {color:'magenta'}
          }>Contact</NavLink>
        </li>
      </ul>
```

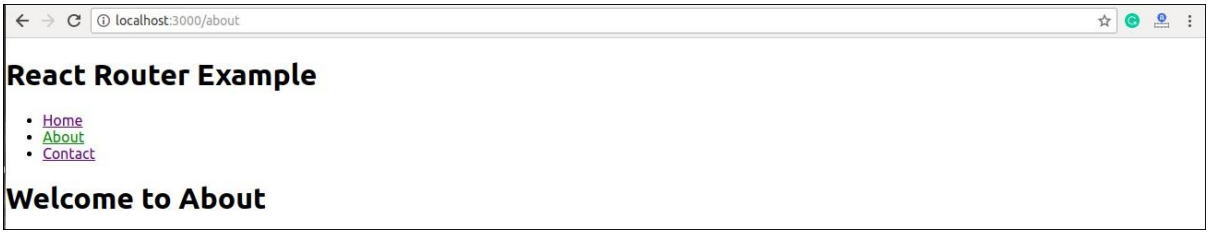
```
      <Route exact path="/" component={App} />
      <Route path="/about" component={About} />
      <Route path="/contact" component={Contact} />
    </div>
  </Router>
)
ReactDOM.render(routing, document.getElementById('root'));
```

Output

When we execute the above program, we will get the following screen in which we can see that **Home** link is of color **Red** and is the only currently **active** link.



Now, when we click on **About** link, its color shown **green** that is the currently **active** link.



<Link> vs <NavLink>

The Link component allows navigating the different routes on the websites, whereas NavLink component is used to add styles to the active routes.

Benefits Of React Router

The benefits of React Router is given below:

- In this, it is not necessary to set the browser history manually.
- Link uses to navigate the internal links in the application. It is similar to the anchor tag.
- It uses Switch feature for rendering.
- The Router needs only a Single Child element.
- In this, every component is specified in .

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.

```
1. import React, { Component } from 'react';
2. class App extends React.Component {
3.  constructor(props) {
4.     super(props);
5.     this.updateSubmit = this.updateSubmit.bind(this);
6.  this.input = React.createRef();7.
    }
8.  updateSubmit(event) {
9.     alert('You have entered the UserName and CompanyName successfully.');
10.    event.preventDefault();
11. }
12.  render() {
13.   return (
14.    <form onSubmit={this.updateSubmit}>
15.     <h1>Uncontrolled Form Example</h1>
16.     <label>Name:
17.        <input type="text" ref={this.input} />
18.     </label>
19.     <label>
20.        CompanyName:
21.        <input type="text" ref={this.input} />
22.     </label>
23.     <input type="submit" value="Submit" />
24.    </form>
25.   );
26. }
```

```
27. }
28. export default App;
```

Output

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

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

Controlled Component

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

Controlled components have functions that govern the data passing into them on every **onChange event**, rather than grabbing the data only once, e.g., when you click a **submit button**. This data is then saved to state and updated with setState() method. This makes component have better control over the form elements and data.

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

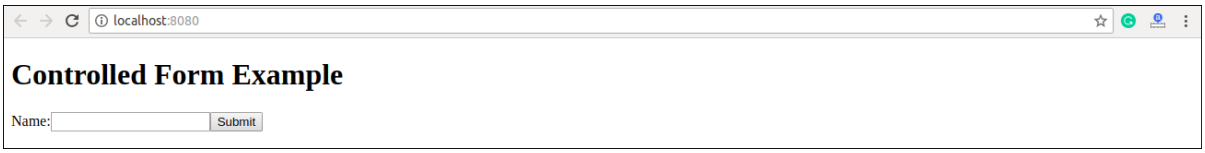
Example

```
1.  import React, { Component } from 'react';
2.  class App extends React.Component {
3.    constructor(props) {
4.       super(props);
5.       this.state = {value: ''};
6.       this.handleChange = this.handleChange.bind(this);
7.    this.handleSubmit = this.handleSubmit.bind(this);8.
      }
9.    handleChange(event) {
10.      this.setState({value: event.target.value});
11. }
12.   handleSubmit(event) {
13.      alert('You have submitted the input successfully: ' + this.state.value);
14.      event.preventDefault();
15. }
16.   render() {
17.      return (
18.          <form onSubmit={this.handleSubmit}>
19.            <h1>Controlled Form Example</h1>
20.            <label>
21.               Name:
22. <input type="text" value={this.state.value}
    onChange={this.handleChange} />
23.            </label>
24.            <input type="submit" value="Submit" />
25.         </form>
26.      );
27. }
28. }
```

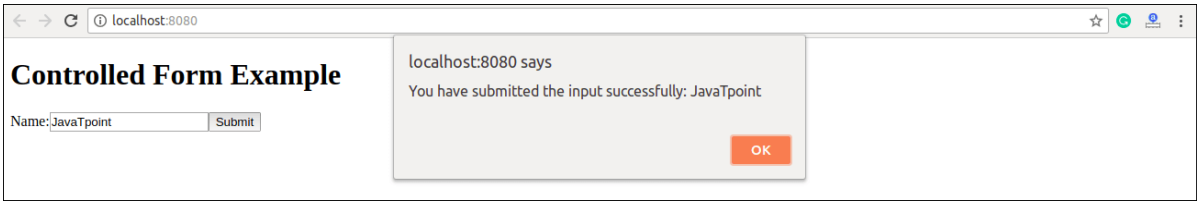
```
29. export default App;
```

Output

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

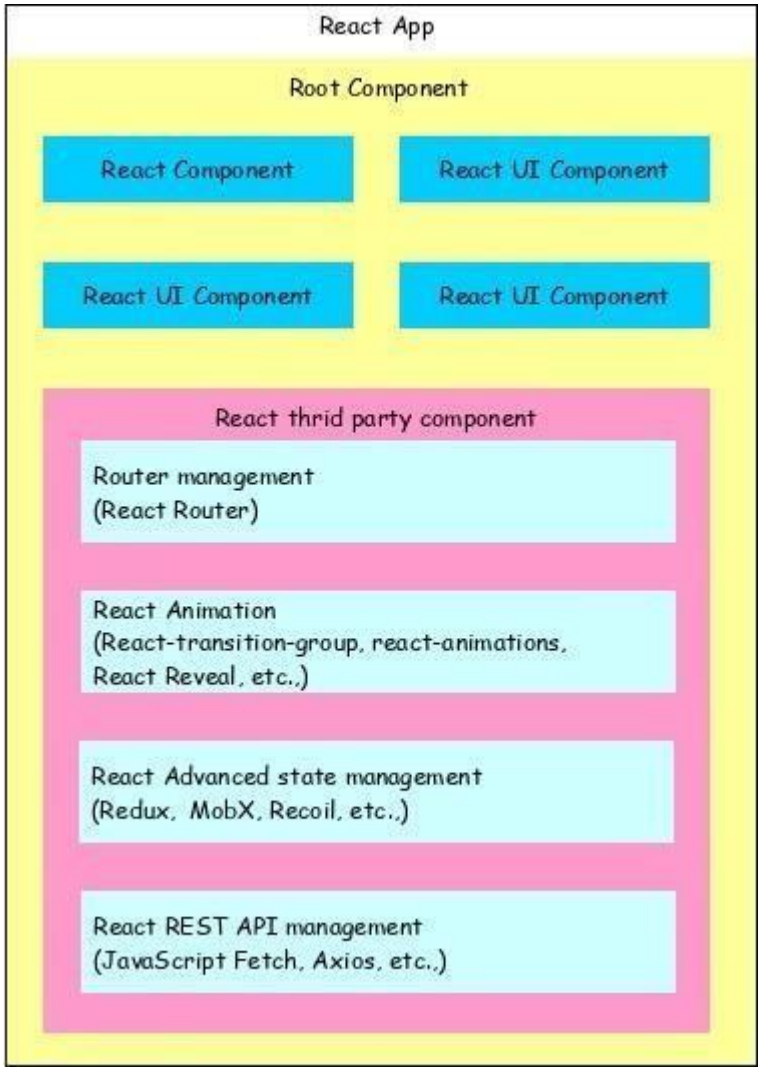


After filling the data in the field, you get the message that can be seen in the below screen.



Architecture of the React Application

React library is just UI library and it does not enforce any particular pattern to write a complex application. Developers are free to choose the design pattern of their choice. React community advocates certain design pattern. One of the patterns is Flux pattern. React library also provides lot of concepts like Higher Order component, Context, Render props, Refs etc., to write better code. React Hooks is evolving concept to do state management in big projects. Let us try to understand the high level architecture of a React application.



- React app starts with a single root component.
- Root component is build using one or more component.
- Each component can be nested with other component to any level.
- Composition is one of the core concepts of React library. So, each component is build by composing smaller components instead of inheriting one component from another component.
- Most of the components are user interface components.
- React app can include third party component for specific purpose such as routing, animation, state management, etc.

React Redux

Redux is an open-source JavaScript library used to manage application state. React uses Redux for building the user interface. It was first introduced by **Dan Abramov** and **Andrew Clark** in **2015**.

React Redux is the official React binding for Redux. It allows React components to read data from a Redux Store, and dispatch **Actions** to the **Store** to update data. Redux helps apps to scale by providing a sensible way to manage state through a unidirectional data flow model. React Redux is conceptually simple. It subscribes to the Redux store, checks to see if the data which your component wants have changed, and re-renders your component.

Redux was inspired by Flux. Redux studied the Flux architecture and omitted unnecessary complexity.

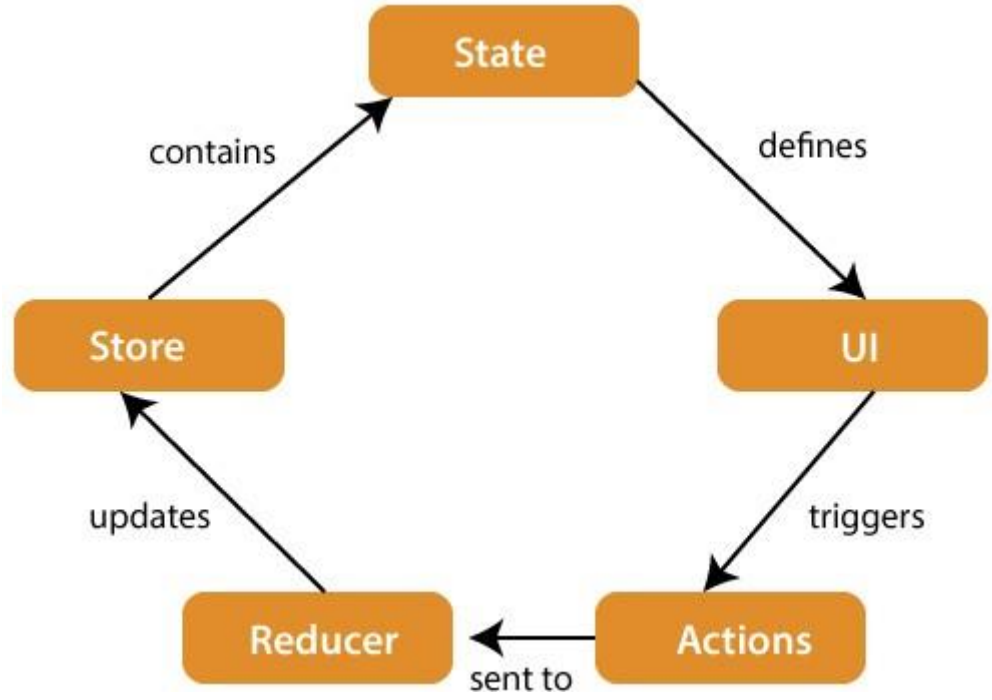
- Redux does not have Dispatcher concept.
- Redux has an only Store whereas Flux has many Stores.
- The Action objects will be received and handled directly by Store.

Why use React Redux?

The main reason to use React Redux are:

- React Redux is the official **UI bindings** for react Application. It is kept up-to-date with any API changes to ensure that your React components behave as expected.
- It encourages good 'React' architecture.
- It implements many performance optimizations internally, which allows to components re-render only when it actually needs.

Redux Architecture



The components of Redux architecture are explained below.

STORE: A Store is a place where the entire state of your application lists. It manages the status of the application and has a dispatch(action) function. It is like a brain responsible for all moving parts in Redux.

ACTION: Action is sent or dispatched from the view which are payloads that can be read by Reducers. It is a pure object created to store the information of the user's event. It includes information such as type of action, time of occurrence, location of occurrence, its coordinates, and which state it aims to change.

REDUCER: Reducer read the payloads from the actions and then updates the store via the state accordingly. It is a pure function to return a new state from the initial state.

React Redux Example

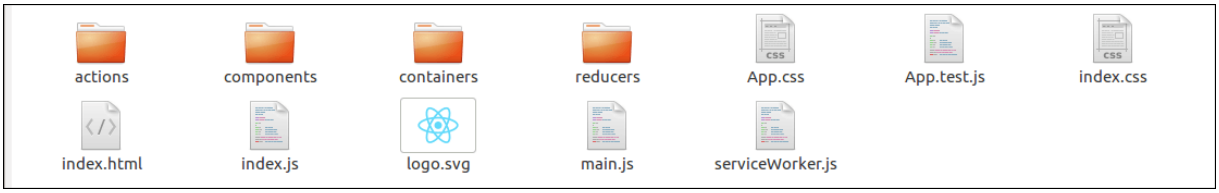
In this section, we will learn how to implements Redux in React application. Here, we provide a simple example to connect Redux and React.

Step-1 Create a new react project using **create-react-app** command. I choose the project name: "**reactproject**." Now, install **Redux** and **React-Redux**.

```
1. javatpoint@root:~/Desktop$ npx create-react-app reactproject
2. javatpoint@root:~/Desktop/reactproject$ npm install redux react-redux --save
```

Step-2 Create Files and Folders

In this step, we need to create folders and files for actions, reducers, components, and containers. After creating folders and files, our project looks like as below image.



Step-3 Actions

It uses '**type**' property to inform about data that should be sent to the **Store**. In this folder, we will create two files: **index.js** and **index.spec.js**. Here, we have created an **action creator** that returns our action and sets an **id** for every created item.

Index.js

```
1. let nextTodoId = 0
2. export const addTodo = text => ({
3.   type: 'ADD_TODO',
4.   id: nextTodoId++,
5.   text
6. })
7.
```

```
export const setVisibilityFilter = filter => ({
  type: 'SET_VISIBILITY_FILTER',
  filter
})

export const toggleTodo = id => ({
  type: 'TOGGLE_TODO',
  id
})

export const VisibilityFilters = {
  SHOW_ALL: 'SHOW_ALL',
  SHOW_COMPLETED: 'SHOW_COMPLETED',
  SHOW_ACTIVE: 'SHOW_ACTIVE'
}
```

Index.spec.js

```
import * as actions from './index'

describe('todo actions', () => {
  it('addTodo should create ADD_TODO action', () => {
    expect(actions.addTodo('Use Redux')).toEqual({
      type: 'ADD_TODO',
      id: 0,
      text: 'Use Redux'
    })
})

it('setVisibilityFilter should create SET_VISIBILITY_FILTER action', () => {
  expect(actions.setVisibilityFilter('active')).toEqual({
```

```
    type: 'SET_VISIBILITY_FILTER',
    filter: 'active'
  })
})

 it('toggleTodo should create TOGGLE_TODO action', () => {
  expect(actions.toggleTodo(1)).toEqual({
    type: 'TOGGLE_TODO',
    id: 1
  })
})
})
```

Step-4 Reducers

As we know, Actions only trigger changes in the app, and the Reducers specify those changes. The Reducer is a function which takes two parameters 'Action' and 'State' to calculate and return an updated State. It read the payloads from the 'Actions' and then updates the 'Store' via the State accordingly.

In the given files, each Reducer managing its own part of the global State. The State parameter is different for every Reducer and corresponds to the part of the 'State' it manages. When the app becomes larger, we can split the Reducers into separate files and keep them completely independent and managing different data domains.

Here, we are using 'combineReducers' helper function to add any new Reducers we might use in the future.

index.js

```
import { combineReducers } from 'redux'
import todos from './todos'
import visibilityFilter from './visibilityFilter'
```

```
4.
5.  export default combineReducers({
6.    todos,
7.    visibilityFilter
8. })
```

Todos.js

```
1.  const todos = (state = [], action) => {
2.    switch (action.type) {
3.      case 'ADD_TODO':
4.        return [
5.state,
6.          {
7.            id: action.id,
8.            text: action.text,
9.            completed: false
10.         }
11.       ]
12.     case 'TOGGLE_TODO':
13.       return state.map(todo =>
14.         (todo.id === action.id)
15.           ? {.  todo, completed: !todo.completed}
16.           : todo
17.       )
18.     default:
19.       return state
20. }
21. }
22. export default todos
```

Todos.spec.js

```
1. import todos from './todos'
2.
3.  describe('todos reducer', () => {
4.    it('should handle initial state', () => {
5.      expect(
6.        todos(undefined, {})
7.      ).toEqual([])
8.    })
9.
10.   it('should handle ADD_TODO', () => {
11.     expect(
12.       todos([], {
13.         type: 'ADD_TODO',
14.         text: 'Run the tests',
15.         id: 0
16.       })
17.     ).toEqual([
18.       {
19.         text: 'Run the tests',
20.         completed: false,
21.         id: 0
22.       }
23.     ])
24.
25.     expect(
26.       todos([
27.         {
28.           text: 'Run the tests',
29.           completed: false,
```

```
30.         id: 0
31.       }
32.     ], {
33.       type: 'ADD_TODO',
34.       text: 'Use Redux',
35.       id: 1
36.     })
37.   ).toEqual([
38.     {
39.       text: 'Run the tests',
40.       completed: false,
41.       id: 0
42.     }, {
43.       text: 'Use Redux',
44.       completed: false,
45.       id: 1
46.     }
47.   ])
48.
49.   expect(
50.     todos([
51.       {
52.         text: 'Run the tests',
53.         completed: false,
54.         id: 0
55.       }, {
56.         text: 'Use Redux',
57.         completed: false,
58.         id: 1
59.       }
```

```
60.       ], {
61.         type: 'ADD_TODO',
62.         text: 'Fix the tests',
63.         id: 2
64.       })
65.     ).toEqual([
66.       {
67.         text: 'Run the tests',
68.         completed: false,
69.         id: 0
70.       }, {
71.         text: 'Use Redux',
72.         completed: false,
73.         id: 1
74.       }, {
75.         text: 'Fix the tests',
76.         completed: false,
77.         id: 2
78.       }
79.     ])
80. })
81.
82.   it('should handle TOGGLE_TODO', () => {
83.     expect(
84.       todos([
85.         {
86.           text: 'Run the tests',
87.           completed: false,
88.           id: 1
89.         }, {
```

```
90.         text: 'Use Redux',
91.         completed: false,
92.         id: 0
93.       }
94.     ], {
95.       type: 'TOGGLE_TODO',
96.       id: 1
97.     })
98.   ).toEqual([
99.     {
100.               text: 'Run the tests',
101.               completed: true,
102.               id: 1
103.             }, {
104.               text: 'Use Redux',
105.               completed: false,
106.               id: 0
107.             }
108.           ])
109.         })
110.       })
```

VisibilityFilter.js

```
1. import { VisibilityFilters } from '../actions'
2.
3.  const visibilityFilter = (state = VisibilityFilters.SHOW_ALL, action) => {
4.    switch (action.type) {
5.      case 'SET_VISIBILITY_FILTER':
6.        return action.filter
7.      default:
```

```
8.    return state9.
      }
10. }
11. export default visibilityFilter
```

Step-5 Components

It is a Presentational Component, which concerned with how things look such as markup, styles. It receives data and invokes callbacks exclusively via props. It does not know where the data comes from or how to change it. It only renders what is given to them.

App.js

It is the root component which renders everything in the UI.

```
1.  import React from 'react'
2.  import Footer from './Footer'
3.  import AddTodo from '../containers/AddTodo'
4.  import VisibleTodoList from '../containers/VisibleTodoList'
5.
6.  const App = () => (
7.    <div>
8.      <AddTodo />
9.      <VisibleTodoList />
10.     <Footer />
11.   </div>
12. )
13. export default App
```

Footer.js

It tells where the user changes currently visible **todos**.

```
1.  import React from 'react'
```

```
2. import FilterLink from '../containers/FilterLink'
3. import { VisibilityFilters } from '../actions'
4.
5. const Footer = () => (
6.   <p>
7.     Show: <FilterLink filter={VisibilityFilters.SHOW_ALL}>All</FilterLink>
8.     {', '}
9.     <FilterLink filter={VisibilityFilters.SHOW_ACTIVE}>Active</FilterLink>
10.    {', '}
11.    <FilterLink
    filter={VisibilityFilters.SHOW_COMPLETED}>Completed</FilterLink>
12. </p>
13. )
14. export default Footer
```

Link.js

It is a link with a callback.

```
1. import React from 'react'
2. import PropTypes from 'prop-types'
3.
4. const Link = ({ active, children, onClick }) => {
5.   if (active) {
6.   return <span>{children}</span>
7.   }
8.
9.   return (
10.    <a
11.      href=""
12.      onClick={e => {
```

```
13.         e.preventDefault()
14.         onClick()
15.       }}
16.     >
17.       {children}
18.     </a>
19. )
20. }
21.
22. Link.propTypes = {
23.   active: PropTypes.bool.isRequired,
24.   children: PropTypes.node.isRequired,
25.   onClick: PropTypes.func.isRequired
26. }
27.
28. export default Link
```

Todo.js

It represents a single todo item which shows **text**.

```
1.  import React from 'react'
2.  import PropTypes from 'prop-types'
3.
4.  const Todo = ({ onClick, completed, text }) => (
5.    <li
6.      onClick={onClick}
7.      style={{
8.      textDecoration: completed ? 'line-through' : 'none'
9.      }}
10. >
```

```
    {text}
  </li>
)

Todo.propTypes = {
  onClick: PropTypes.func.isRequired,
  completed: PropTypes.bool.isRequired,
  text: PropTypes.string.isRequired
}

export default Todo
```

TodoList.js

It is a list to show visible todos{ id, text, completed }.

```
import React from 'react'
import PropTypes from 'prop-types'
import Todo from './Todo'

const TodoList = ({ todos, onTodoClick }) => (
  <ul>
    {todos.map((todo, index) => (
    <Todo key={index} {...todo} onClick={() => onTodoClick(index)} />
    ))}
  </ul>
)

TodoList.propTypes = {
  todos: PropTypes.arrayOf(
    PropTypes.shape({
```

```
16.       id: PropTypes.number.isRequired,
17.       completed: PropTypes.bool.isRequired,
18.       text: PropTypes.string.isRequired
19.     }).isRequired
20.   ).isRequired,
21.   onTodoClick: PropTypes.func.isRequired
22. }
23. export default TodoList
```

Step-6 Containers

It is a Container Component which concerned with how things work such as data fetching, updates State. It provides data and behavior to presentational components or other container components. It uses Redux State to read data and dispatch Redux Action for updating data.

AddTodo.js

It contains the input field with an ADD (submit) button.

```
1.  import React from 'react'
2.  import { connect } from 'react-redux'
3.  import { addTodo } from '../actions'
4.
5.  const AddTodo = ({ dispatch }) => {
6.    let input
7.
8.    return (
9.      <div>
10.       <form onSubmit={e => {
11.         e.preventDefault()
12.         if (!input.value.trim()) {
13.           return
14.         }
```

```
15.         dispatch(addTodo(input.value))
16.         input.value = ''
17.       }}>
18.         <input ref={node => input = node} />
19.         <button type="submit">
20.           Add Todo
21.         </button>
22.       </form>
23.     </div>
24.   )
25. }
26. export default connect()(AddTodo)
```

FilterLink.js

It represents the current visibility filter and renders a link.

```
1.  import { connect } from 'react-redux'
2.  import { setVisibilityFilter } from '../actions'
3.  import Link from '../components/Link'
4.
5.  const mapStateToProps = (state, ownProps) => ({
6.    active: ownProps.filter === state.visibilityFilter
7.  })
8.
9.  const mapDispatchToProps = (dispatch, ownProps) => ({
10.   onClick: () => dispatch(setVisibilityFilter(ownProps.filter))
11. })
12.
13. export default connect(
14.   mapStateToProps,
```

```
15.  mapDispatchToProps
16. )(Link)
```

VisibleTodoList.js

It filters the todos and renders a TodoList.

```
1.  import { connect } from 'react-redux'
2.  import { toggleTodo } from '../actions'
3.  import TodoList from '../components/TodoList'
4.  import { VisibilityFilters } from '../actions'
5.
6.  const getVisibleTodos = (todos, filter) => {
7.    switch (filter) {
8.      case VisibilityFilters.SHOW_ALL:
9.        return todos
10.     case VisibilityFilters.SHOW_COMPLETED:
11.       return todos.filter(t => t.completed)
12.     case VisibilityFilters.SHOW_ACTIVE:
13.       return todos.filter(t => !t.completed)
14.     default:
15.       throw new Error('Unknown filter: ' + filter)
16. }
17. }
18.
19. const mapStateToProps = state => ({
20.   todos: getVisibleTodos(state.todos, state.visibilityFilter)
21. })
22.
23. const mapDispatchToProps = dispatch => ({
24.   toggleTodo: id => dispatch(toggleTodo(id))
```

```
25. })
26.
27. export default connect(
28.   mapStateToProps,
29.   mapDispatchToProps
30. )(TodoList)
```

Step-7 Store

All container components need access to the Redux Store to subscribe to it. For this, we need to pass it(store) as a prop to every container component. However, it gets tedious. So we recommend using special React Redux component called which make the store available to all container components without passing it explicitly. It used once when you render the root component.

index.js

```
1.  import React from 'react'
2.  import { render } from 'react-dom'
3.  import { createStore } from 'redux'
4.  import { Provider } from 'react-redux'
5.  import App from './components/App'
6.  import rootReducer from './reducers'
7.
8.  const store = createStore(rootReducer)
9.
10. render(
11.   <Provider store={store}>
12.     <App />
13.   </Provider>,
14.   document.getElementById('root')
15. )
```

Output

When we execute the application, it gives the output as below screen.

Now, we will be able to add items in the list.

Client-Server Communication

Client and server applications communicate by sending individual messages on an as-needed basis, rather than an ongoing stream of communication.

These communications are almost always initiated by clients in the form of requests. These requests are fulfilled by the server application which sends back a response containing the resource you requested, among other things.

The Anatomy of an HTTP Request

An HTTP request must have the following:

- An HTTP method (like `GET`)
- A host URL (like `https://api.spotify.com/`)

- An endpoint path(like `v1/artists/{id}/related-artists`)

A request can also optionally have:

- Body
- Headers
- Query strings
- HTTP version

The Anatomy of an HTTP Response

A response must have the following:

- Protocol version (like `HTTP/1.1`)
- Status code (like `200`)
- Status text (`OK`)
- Headers

A response may also optionally have:

- Body

HTTP Methods Explained

Now that we know what HTTP is and why it's used, let's talk about the different methods we have available to us.

In the weather app example above, we wanted to retrieve weather information about a city. But what if we wanted to submit weather information for a city?

In real life, you probably wouldn't have permissions to alter someone else's data, but let's imagine that we are contributors to a community-run weather app. And in addition to getting the weather information from an API, members in that city could update this information to display more accurate data.

Or what if we wanted to add a new city altogether that, for some reason, doesn't already exist in our database of cities? These are all different functions – retrieve data, update data, create new data – and there are HTTP methods for all of these.

HTTP POST request

We use `POST` to create a new resource. A `POST` request requires a body in which you define the data of the entity to be created.

A successful POST request would be a 200 response code. In our weather app, we could use a POST method to add weather data about a new city.

HTTP GET request

We use GET to read or retrieve a resource. A successful GET returns a response containing the information you requested.

In our weather app, we could use a GET to retrieve the current weather for a specific city.

HTTP PUT request

We use PUT to modify a resource. PUT updates the entire resource with data that is passed in the body payload. If there is no resource that matches the request, it will create a new resource.

In our weather app, we could use PUT to update all weather data about a specific city.

HTTP PATCH request

We use PATCH to modify a part of a resource. With PATCH, you only need to pass in the data that you want to update.

In our weather app, we could use PATCH to update the rainfall for a specified day in a specified city.

HTTP DELETE request

We use DELETE to delete a resource. In our weather app, we could use DELETE to delete a city we no longer wanted to track for some reason.

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

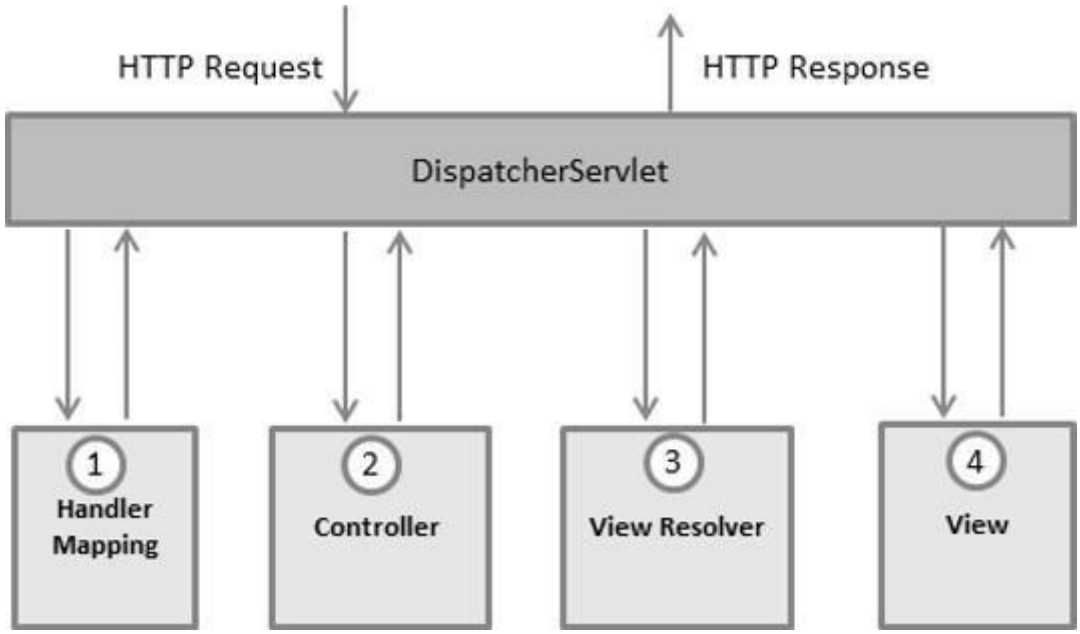
The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the

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	Spring MVC Hello World Example This example will explain how to write a simple Spring Web Hello World application.
2	Spring MVC Form Handling Example 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	Spring Page Redirection Example

	Learn how to use page redirection functionality in Spring MVC Framework.
4	Spring Static Pages Example Learn how to access static pages along with dynamic pages in Spring MVC Framework.
5	Spring Exception Handling Example 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.

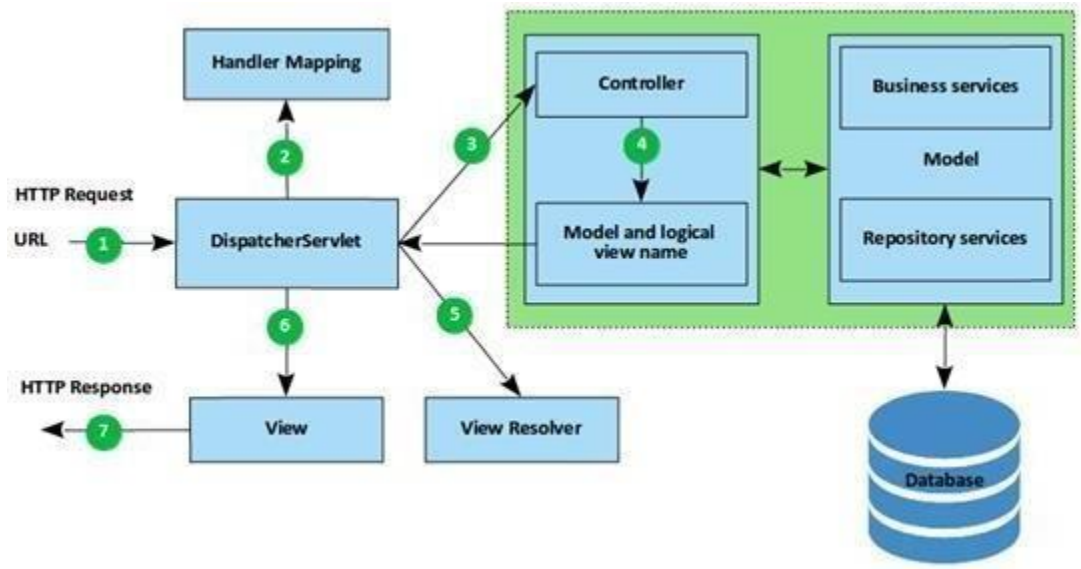


Figure 1: The Spring framework

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

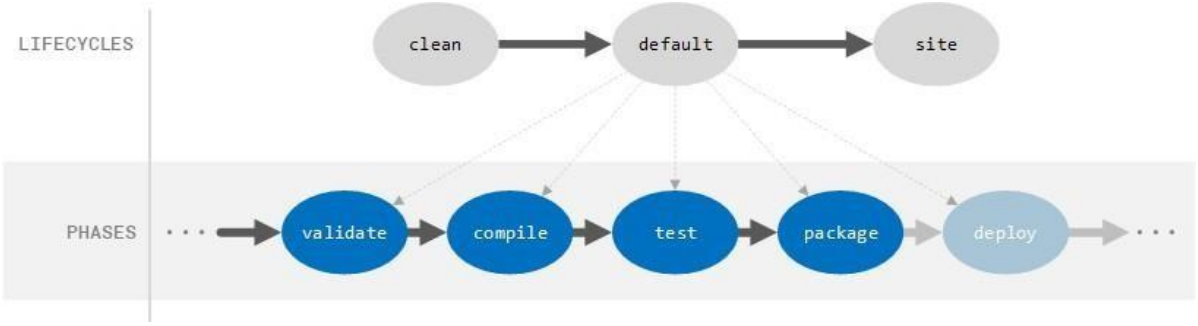
Phases

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 with 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
```

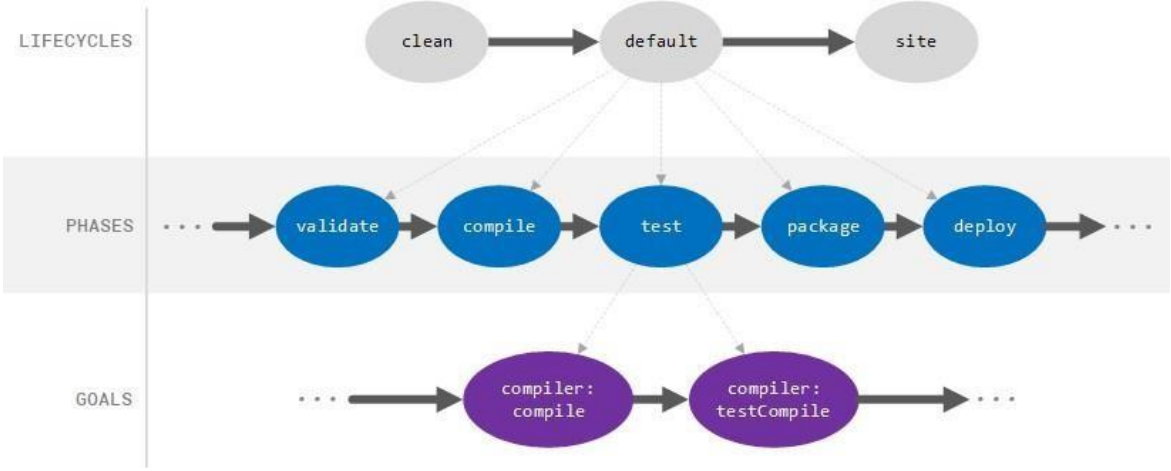
Goals & Plugins

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:

```
1
mvn foo:bar
```

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

Relation Schema

Relation schema defines the design and structure of the relation like it consists of the relation name, set of attributes/field names/column names. every attribute would have an associated domain.

There is a student named Geeks, she is pursuing B.Tech, in the 4th year, and belongs to IT department (department no. 1) and has roll number 1601347 She is proctored by Mrs. S Mohanty. If we want to represent this using databases we would have to create a student table with name, sex, degree, year, department, department number, roll number and proctor (adviser) as the attributes.

```
student (rollNo, name, degree, year, sex, deptNo, advisor)
```

Note –
If we create a database, details of other students can also be recorded. Similarly, we have the IT Department, with department Id 1, having Mrs. Sujata Chakravarty as the head of department. And we can call the department on the number 0657 228662 .

This and other departments can be represented by the department table, having department ID, name, hod and phone as attributes.

```
department (deptId, name, hod, phone)
```

The course that a student has selected has a courseid, course name, credit and department number.

```
course (coursId, ename, credits, deptNo)
```

The professor would have an employee Id, name, sex, department no. and phone number.

```
professor (empId, name, sex, startYear, deptNo, phone)
```

We can have another table named enrollment, which has roll no, courseId, semester, year and grade as the attributes.

```
enrollment (rollNo, coursId, sem, year, grade)
```

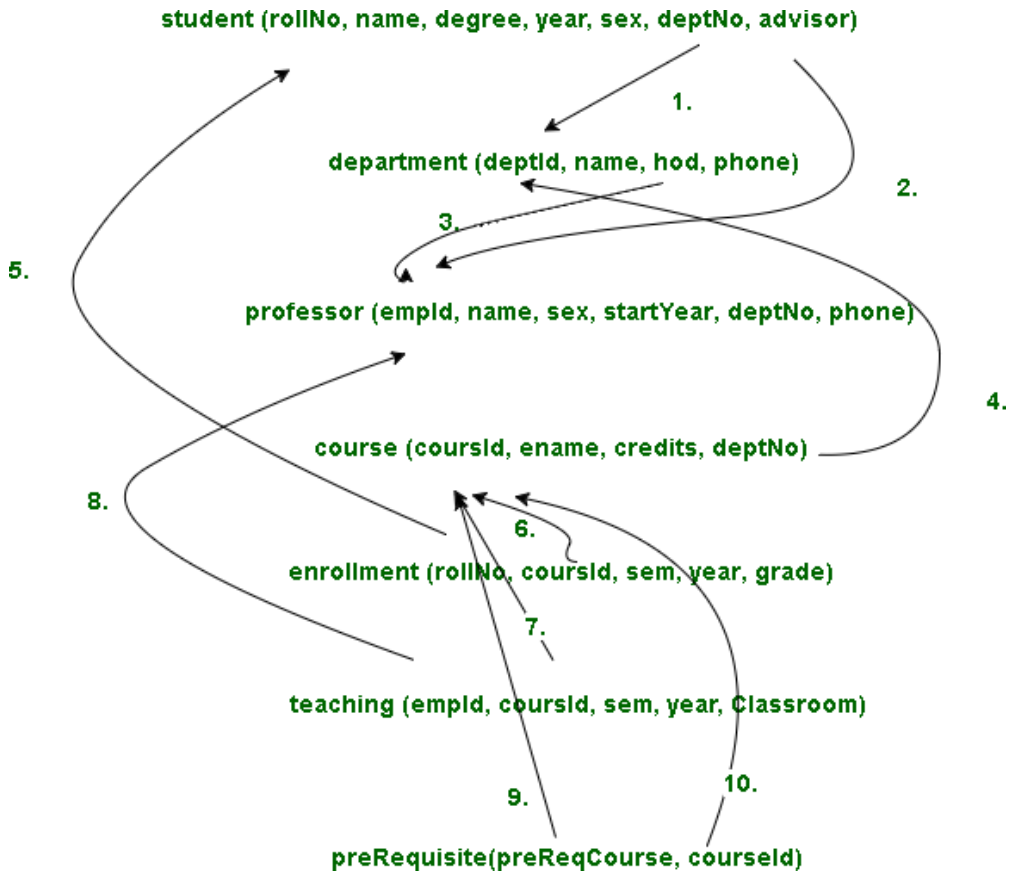
Teaching can be another table, having employee id, course id, semester, year and classroom as attributes.

```
teaching (empId, coursed, sem, year, Classroom)
```

When we start courses, there are some courses which another course that needs to be completed before starting the current course, so this can be represented by the Prerequisite table having prerequisite course and course id attributes.

```
prerequisite (preReqCourse, courseId)
```

The relations between them is represented through arrows in the following **Relation diagram**,



1. This represents that the deptNo in student table table is same as deptId used in department table. deptNo in student table is a foreign key. It refers to deptId in department table.

2. This represents that the advisor in student table is a foreign key. It refers to empId in professor table.
3. This represents that the hod in department table is a foreign key. It refers to empId in professor table.
4. This represents that the deptNo in course table table is same as deptId used in department table. deptNo in student table is a foreign key. It refers to deptId in department table.
5. This represents that the rollNo in enrollment table is same as rollNo used in student table.
6. This represents that the courseId in enrollment table is same as courseId used in course table.
7. This represents that the courseId in teaching table is same as courseId used in course table.
8. This represents that the empId in teaching table is same as empId used in professor table.
9. This represents that preReqCourse in prerequisite table is a foreign key. It refers to courseId in course table.
10. This represents that the deptNo in student table is same as deptId used in department table.

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 (liked 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:

1. **SELECT** [**DISTINCT**] Attribute_List **FROM** R1,R2….RM
2. [**WHERE** condition]
3. [**GROUP BY** (Attributes)[**HAVING** condition]]
4. [**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	9156253131	20
4	SURESH	DELHI	9156768971	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:

5. **COUNT:** Count function is used to count the number of rows in a relation. e.g;

```
SELECT COUNT (PHONE) FROM STUDENT;
```

COUNT(PHONE)
4

6. **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 tupples. It is also defined as sum divided by count values.
Syntax:AVG(attributename)
OR
Syntax: SUM(attributename)/COUNT(attributename)
The above mentioned syntax also retrieves the average value of tupples.

MAXIMUM: It extracts the maximum value among the set of tupples.
Syntax:MAX(attributename)

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

Java Database Connectivity (JDBC) is an application programming interface **(API)** that defines how a client may access a database. It is a data access technology used for Java database connectivity. It provides methods to

query and update data in a database and is oriented toward relational databases. JDBC offers a natural Java interface for working with **SQL**. JDBC is needed to provide a "pure Java" solution for application development. JDBC API uses JDBC drivers to connect with the database.

There are 4 Types of JDBC Drivers:

1. JDBC-ODBC Bridge Driver
2. Native API Driver (partially java driver)
3. Network Protocol Driver (fully java driver)
4. Thin Driver (fully java driver)

The advantages of JDBC API is as follows:

1. Automatically creates the **XML** format of data from the database.
2. It supports query and stored procedures.
3. Almost any database for which **ODBC** driver is installed can be accessed.

The disadvantages of JDBC API is as follows:

1. Writing a lot of codes before and after executing the query, such as creating connection, creating a statement, closing result-set, closing connection, etc.
2. Writing exception handling code on the database logic.
3. Repetition of these codes from one to another database logic is time-consuming.

These problems of **JDBC API** are eliminated by **Spring JDBC-Template**. It provides methods to write the queries directly that saves a lot of time and effort.

Data Access using JDBC Template

There are a number of options for selecting an approach to form the basis for your **JDBC** database access. Spring framework provides the following approaches for **JDBC** database access:

- JdbcTemplate
- NamedParameterJdbcTemplate
- SimpleJdbcTemplate
- SimpleJdbcInsert and SimpleJdbcCall

JDBC Template

JdbcTemplate is a central class in the **JDBC** core package that simplifies the use of **JDBC** and helps to avoid common errors. It internally uses **JDBC API** and eliminates a lot of problems with **JDBC API**. It executes SQL queries or updates, initiating iteration over ResultSets and catching **JDBC** exceptions

and translating them to the generic. It executes core **JDBC** workflow, leaving application code to provide SQL and extract results. It handles the exception and provides the informative exception messages with the help of exception classes defined in the **org.springframework.dao** package.
The common methods of spring JdbcTemplate class.

Methods	Description
public int update(String query)	Used to insert, update and delete records.
public int update(String query, Object… args)	Used to insert, update and delete records using **PreparedStatement** using given arguments.
public T execute(String sql, PreparedStatementCallback action)	Executes the query by using **PreparedStatementCallback.**
public void execute(String query)	Used to execute **DDL** query.
public T query(String sql, ResultSetExtractor result)	Used to fetch records using **ResultSetExtractor**.

JDBC Template Queries

Basic query to count students stored in the database using **JdbcTemplate**.

```
int result = jdbcTemplate.queryForObject(
"SELECT COUNT(*) FROM STUDENT", Integer.class);
```

And here's a simple **INSERT**:

```
public int addStudent(int id)
{
  return jdbcTemplate.update("INSERT INTO STUDENT VALUES (?, ?, ?)",
id, "megan", "India");
}
```

The standard syntax of providing parameters is using the **"?"** character.
Implementation: Spring JdbcTemplate
We start with some simple configurations of the data source. We'll use a MySQL database
Example:

- Java

```
/*package whatever //do not write package name here */

@Configuration

@ComponentScan("com.exploit.jdbc")

public class SpringJdbcConfig {

    @Bean public DataSource mysqlDataSource()

    {

        DriverManagerDataSource dataSource

            = new DriverManagerDataSource();

        dataSource.setDriverClassName(

            "com.mysql.jdbc.Driver");

        dataSource.setUrl(

            "jdbc:mysql://localhost:8800/springjdbc");

        dataSource.setUsername("user");

        dataSource.setPassword("password");



        return dataSource;

    }

}
```

A. File: Student.java

- Java

```
// Java Program to Illustrate Student Class


package com.exploit.org;


// Class

public class Student {


    // Class data members

    private Integer age;

    private String name;

    private Integer id;


    // Constructor

    public Student() {}


    // Setters and Getters

    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; }

}
```

B. File: StudentDAO.java
Below is the implementation of the Data Access Object interface file StudentDAO.java.

Example:

- Java

```
// Java Program to Illustrate StudentDAO Class


package com.exploit.org;


// importing required classes

import java.util.List;

import javax.sql.DataSource;


// Class

public interface StudentDAO {
```

```
    // Used to initialize database resources
    // ie. connection
    public void setDataSource(DataSource ds);

    // Used to list down all the records
    // from the Student table
    public List<Student> listStudents();
}
```

C. File: Maven Dependency

Dependency is used in the **pom.xml** file.

Example:

- XML

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>

<dependency>
   <groupId>mysql</groupId>
```

```
    <artifactId>mysql-connector-java</artifactId>

    <scope>runtime</scope>

</dependency>
```

D. File: StudentJDBCTemplate.java
Below is the implementation class file **StudentJDBCTemplate.java** for the defined DAO interface **StudentDAO**

Example:

- Java

```
// Java Program Illustrating Implementation

// of StudentDAO Class



package com.exploit.org;



// Importing required classes

import java.util.List;
```

```
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;


// Class
// Implementing StudentDAO Class
public class StudentJDBCTemp implements StudentDAO {


    // Class data members
    private DataSource dataSource;
    private JdbcTemplate jdbcTemplateObject;


    // Method 1
    public void setDataSource(DataSource dataSource)
    {
        // This keyword refers to current instance itself
        this.dataSource = dataSource;
        this.jdbcTemplateObject
            = new JdbcTemplate(dataSource);
    }
```

```
    // Method 2
    public List<Student> listStudents()
    {
        // Custom SQL query
        String SQL = "select * from Student";
        List<Student> students = jdbcTemplateObject.query(
            SQL, new StudentMapper());

        return students;
    }
}
```

Cloud Deployment Models

In cloud computing, we have access to a shared pool of computer resources (servers, storage, programs, and so on) in the cloud. You simply need to request additional resources when you require them. Getting resources up and running quickly is a breeze thanks to the clouds. It is possible to release resources that are no longer necessary. This method allows you to just pay for what you use. Your cloud provider is in charge of all upkeep. It functions as a virtual computing environment with a deployment architecture that varies depending on the amount of data you want to store and who has access to the infrastructure.

Deployment Models

The cloud deployment model identifies the specific type of cloud environment based on ownership, scale, and access, as well as the cloud's nature and

purpose. The location of the servers you're utilizing and who controls them are defined by a cloud deployment model. It specifies how your cloud infrastructure will look, what you can change, and whether you will be given services or will have to create everything yourself. Relationships between the infrastructure and your users are also defined by cloud deployment types.

Different types of cloud computing deployment models are:

1. Public cloud
2. Private cloud
3. Hybrid cloud
4. Community cloud
5. Multi-cloud

Public Cloud

The public cloud makes it possible for anybody to access systems and services. The public cloud may be less secure as it is open to everyone. The public cloud is one in which cloud infrastructure services are provided over the internet to the general people or major industry groups. The infrastructure in this cloud model is owned by the entity that delivers the cloud services, not by the consumer. It is a type of cloud hosting that allows customers and users to easily access systems and services. This form of cloud computing is an excellent example of cloud hosting, in which service providers supply services to a variety of customers. In this arrangement, storage backup and retrieval services are given for free, as a subscription, or on a per-user basis. Example: Google App Engine etc.

Advantages of Public Cloud Model:

- **Minimal Investment:** Because it is a pay-per-use service, there is no substantial upfront fee, making it excellent for enterprises that require immediate access to resources.
- **No setup cost:** The entire infrastructure is fully subsidized by the cloud service providers, thus there is no need to set up any hardware.
- **Infrastructure Management is not required:** Using the public cloud does not necessitate infrastructure management.

- **No maintenance:** The maintenance work is done by the service provider (Not users).

- **Dynamic Scalability:** To fulfill your company's needs, on-demand resources are accessible.

Disadvantages of Public Cloud Model:

- **Less secure:** Public cloud is less secure as resources are public so there is no guarantee of high-level security.

- **Low customization:** It is accessed by many public so it can't be customized according to personal requirements.

Private Cloud

The private cloud deployment model is the exact opposite of the public cloud deployment model. It's a one-on-one environment for a single user (customer). There is no need to share your hardware with anyone else. The distinction between private and public clouds is in how you handle all of the hardware. It is also called the "internal cloud" & it refers to the ability to access systems and services within a given border or organization. The cloud platform is implemented in a cloud-based secure environment that is protected by powerful firewalls and under the supervision of an organization's IT department. The private cloud gives greater flexibility of control over cloud resources.

Advantages of Private Cloud Model:

- **Better Control:** You are the sole owner of the property. You gain complete command over service integration, IT operations, policies, and user behavior.

- **Data Security and Privacy:** It's suitable for storing corporate information to which only authorized staff have access. By segmenting resources within the same infrastructure, improved access and security can be achieved.

- **Supports Legacy Systems:** This approach is designed to work with legacy systems that are unable to access the public cloud.
- **Customization:** Unlike a public cloud deployment, a private cloud allows a company to tailor its solution to meet its specific needs.

Disadvantages of Private Cloud Model:

- **Less scalable:** Private clouds are scaled within a certain range as there is less number of clients.
- **Costly:** Private clouds are more costly as they provide personalized facilities.

Hybrid Cloud

By bridging the public and private worlds with a layer of proprietary software, hybrid cloud computing gives the best of both worlds. With a hybrid solution, you may host the app in a safe environment while taking advantage of the public cloud's cost savings. Organizations can move data and applications between different clouds using a combination of two or more cloud deployment methods, depending on their needs.

Advantages of Hybrid Cloud Model:

- **Flexibility and control:** Businesses with more flexibility can design personalized solutions that meet their particular needs.
- **Cost:** Because public clouds provide scalability, you'll only be responsible for paying for the extra capacity if you require it.
- **Security:** Because data is properly separated, the chances of data theft by attackers are considerably reduced

.

Disadvantages of Hybrid Cloud Model:

- **Difficult to manage:** Hybrid clouds are difficult to manage as it is a combination of both public and private cloud. So, it is complex.
- **Slow data transmission:** Data transmission in the hybrid cloud takes place through the public cloud so latency occurs.

Community Cloud

It allows systems and services to be accessible by a group of organizations. It is a distributed system that is created by integrating the services of different clouds to address the specific needs of a community, industry, or business. The infrastructure of the community could be shared between the organization which has shared concerns or tasks. It is generally managed by a third party or by the combination of one or more organizations in the community.

Advantages of Community Cloud Model:

- **Cost Effective:** It is cost-effective because the cloud is shared by multiple organizations or communities.
- **Security:** Community cloud provides better security.
- **Shared resources:** It allows you to share resources, infrastructure, etc. with multiple organizations.
- **Collaboration and data sharing:** It is suitable for both collaboration and data sharing.

Disadvantages of Community Cloud Model:

- **Limited Scalability:** Community cloud is relatively less scalable as many organizations share the same resources according to their collaborative interests.
- **Rigid in customization:** As the data and resources are shared among different organizations according to their mutual interests if an organization wants some changes according to their needs they cannot do so because it will have an impact on other organizations.

Multi-cloud

We're talking about employing multiple cloud providers at the same time under this paradigm, as the name implies. It's similar to the hybrid cloud deployment approach, which combines public and private cloud resources. Instead of merging private and public clouds, multi-cloud uses many public clouds. Although public cloud providers provide numerous tools to improve the reliability of their services, mishaps still occur. It's quite rare that two distinct clouds would have an incident at the same moment. As a result, multi-cloud deployment improves the high availability of your services even more.

Advantages of a Multi-Cloud Model:

- You can mix and match the best features of each cloud provider's services to suit the demands of your apps, workloads, and business by choosing different cloud providers.
- **Reduced Latency:** To reduce latency and improve user experience, you can choose cloud regions and zones that are close to your clients.
- **High availability of service:** It's quite rare that two distinct clouds would have an incident at the same moment. So, the multi-cloud deployment improves the high availability of your services.
-

Disadvantages of Multi-Cloud Model:

- **Complex:** The combination of many clouds makes the system complex and bottlenecks may occur.
- **Security issue:** Due to the complex structure, there may be loopholes to which a hacker can take advantage hence, makes the data insecure.

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 could computing and utilizing that money public service.
1. **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.

2. **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.