

HTML

What is HTML?

- HTML(Hyper Text Markup Language)
 - is a language for describing web pages.
 - not a programming language
 - uses markup tags to describe web pages.
- Most Web documents are created using HTML.
 - Documents are saved with extension .html or .htm.
- Markup?
 - Markup Tags are strings in the language surrounded by a < and a > sign.
 - Opening tag: <html> Ending tag: </html>
 - Not case sensitive.
 - Can have attributes which provide additional information about HTML elements on your page.

Example

 - <body bgcolor="red">
 - <table border="0">

HTML

- An HTML document appears as follows:

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>Title of page</title>
  </head>
  <body>
    This is my first homepage. <b>This text is bold</b>
  </body>
</html>
```

DOCTYPE tells type, version, language of particular document.

- HTML Head Section:** contain information about the document. The browser does not display this information to the user. Following tags can be in the head section: `<base>`, `<link>`, `<meta>`, `<script>`, `<style>`, and `<title>`.
- HTML Body Section:** defines the document's body. Contains all the contents of the document (like text, images, colors, graphics, etc.).
- Each document can have at most one `<body>` element

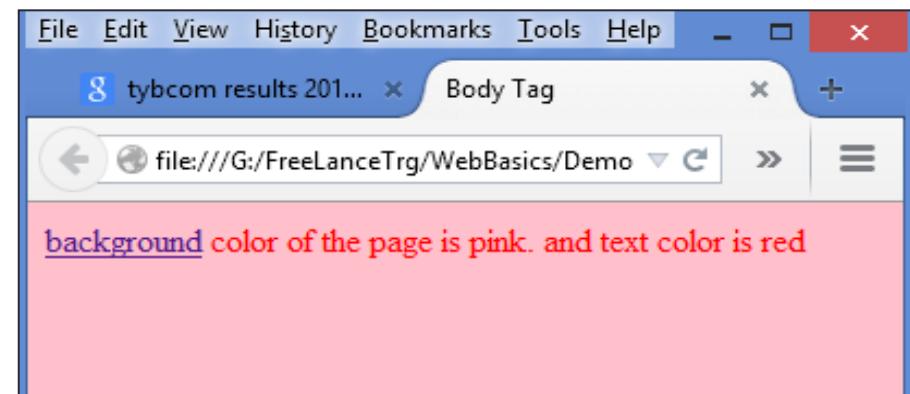
Document (Body) Contents

- Body Text
 - HTML truncates spaces in your text.
 - Use
 to insert new lines.
 - Use <p> tag to create paragraphs.
 - Use <div> to hold division or a section in an HTML document
 - Use as an inline container to mark up a part of a text, or a part of a document
- Comments in HTML Document
 - Increase code readability; Ignored by the browser.
 - Example of HTML comment: <!-- This is a Sample HTML Comment -->
- <div>...</div>: Creates divisions in Web pages. Can be used to set the alignment/class for an entire section of the page. Eg:
 - <div align="center"> This text is at the centre of the browser window </div>

Attributes

- HTML attributes are special words which provide additional information about the elements or attributes are the modifier of the HTML element.
- Eg of attributes used in <body> element :
 - BGCOLOR: Gives a background color to HTML page.
 - BACKGROUND: Use to specify images to the BACKGROUND.
 - TEXT: Specifies text color throughout the browser window
 - LINK, ALINK, VLINK: Used to specify link color, active link color & visited link color
- Examples:
 - <body text="red"> OR <body text="#FF0000">
 - <body link="red" alink="blue" vlink="purple">
 - <body bgcolor="black"> OR <body bgcolor="#000000">
 - <body background="http://www.mysite.edu/img1.gif">

```
<HTML>
<HEAD>
  <TITLE>Body Tag</TITLE>
</HEAD>
<BODY BGCOLOR="pink" text="red"
      alink="green" link="yellow">
  <a href="body.html">background</a>
  color of the page is pink. and text color is red
</BODY>
</HTML>
```



Formatting tags

- Bold Font: **...**
- Italic: *<i>...</i>*
- Underline: <u>...</u>
- Strikethrough: ~~<strike>~~ or ~~<s>~~
- Subscript: _{...}
- Superscript: ^{...}

```

<b>This is in bold font</b><br>
<i>This text is in Italics</i><br>
<u>This text is Underlined</u><br>
<small>This is small text.</small><br>
<font size=7>This text is very large</font><br>
<font color="Blue">This is Blue Text.</font><br>
<del>This is strikethrough style text</del><br>
The chemical formula of water is h<sub>2</sub>o.<br>
A simple formula for a parabola is y = x<sup>2</sup>. <br>
Applying some <mark>markup</mark> here<br>
Applying <em>emphasis</em><br>
<tt>teletype text</tt>

```

This is in bold font

This text is in Italics

This text is Underlined

This is small text.

This text is ve

This is Blue Text.

~~This is strikethrough style text~~

The chemical formula of water is h₂o.

A simple formula for a parabola is y = x².

Applying some **markup** here

Applying **emphasis**

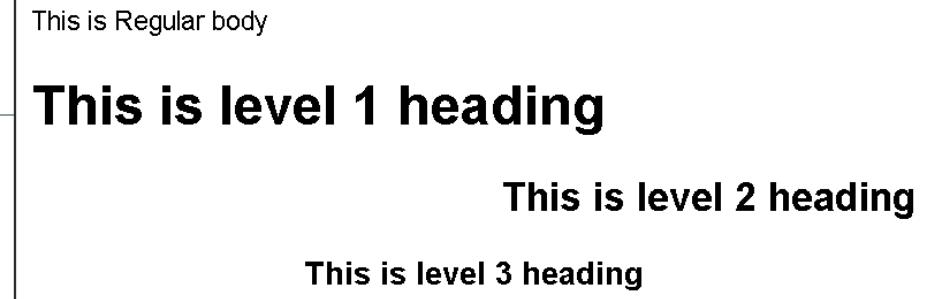
teletype text

Layout tags

- Heading Styles:

- **<hn>.....</hn>**
 - Value of n can range from **1 to 6**
- **<h1 align="center">This is level 1 heading</h1>**

```
<html>
<head>
  <title>Header Demo</title>
</head>
<body>
  This is Regular body
  <h1 align="left">This is level 1 heading</h1>
  <h2 align="right">This is level 2 heading</h2>
  <h3 align="center">This is level 3 heading</h3>
</body>
</html>
```



Special Characters in HTML

- Character Entities

- Comprise following parts:
 - Ampersand (&),
 - Entity name or a #
 - Character code
 - Semicolon (;)
- Included in HTML page using:
 - To display “>” symbol, use character code 62 i.e. > or >
 - For space, use

| Result | Description | Entity Name | Entity Number |
|--------|----------------------|-------------|---------------|
| | non-breaking space | | |
| < | less than | < | < |
| > | greater than | > | > |
| & | ampersand | & | & |
| ¢ | cent | ¢ | ¢ |
| £ | pound | £ | £ |
| ¥ | yen | ¥ | ¥ |
| € | euro | € | € |
| © | copyright | © | © |
| ® | registered trademark | ® | ® |

Horizontal Lines in a Web Page

- <hr> - horizontal Rule. Attributes:
 - Size: Line thickness <hr size="5">
 - Width: Line width either in pixels or % of browser window
 - <hr width="100"> or <hr width="60%">
 - Align: Alignment values can be left, center or right <hr align="center">
 - Color: to display colored horizontal lines. Eg <hr color="red">

```
<body> <p>
    This paragraph contains a lot of lines
    in the source code,
    but the browser ignores it. </p>
    <hr size="2" width="50%" color="blue" align = "left" >
    <p>Notice the horizontal rule occupying 50 % of the window width.
    <p>This paragraph contains <br> line breaks in the
    source code <br>
    so this is the third line displayed within the paragraph.
</body>
```

This paragraph contains a lot of lines in the source code, but the browser ignores it.

Notice the horizontal rule occupying 50 % of the window width.

This paragraph contains
line breaks in the source code
so this is the third line displayed within the paragraph.

Numbered List (Ordered List)

- Automatically generate numbers in front of each item in the list
 - Number placed against an item depends on the location of the item in the list.
 - Example:

```
<body>
  <ol>
    <li>INDIA</li>
    <li>SRILANKA</li>
  </ol>
</body>
```

1. INDIA
2. SRILANKA

- To start an ordered list at a number other than 1, use **START** attribute:
Example : <ol start=11>

11. INDIA
12. SRILANKA

- Select the type of numbering system with the **type** attribute. Example:

- A-Uppercase letters. <OL TYPE=A>
- a-Lowercase letters. <OL TYPE=a>
- I-Uppercase Roman letters
- i-Lowercase Roman letters
- 1-Standard numbers, default

```
<ol type="a">
  <li>INDIA</li>
  <li>SRILANKA</li>
</ol>
```

a. INDIA
b. SRILANKA

Bulleted List (Unordered List)

- Example:

```
<ul>
    <li>Latte</li>
    <li>Expresso</li>
    <li>Mocha</li>
</ul>
```

- Latte
- Expresso
- Mocha

- To change the type of bullet used throughout the list, place the TYPE attribute in the **** tag at the beginning of the list.
 - DISC (default) gives bullet in disc form.
 - SQUARE gives bullet in square form.
 - CIRCLE gives bullet in circle form.

```
<ul>
    <li type='disc'>Latte</li>
    <li type='square'>Expresso</li>
    <li type='circle'>Mocha</li>
</ul>
```

- Latte
- Expresso
- Mocha

Adding Image

- Images are added into a document using tag.
- Attributes :
 - alt: holds a text description of the image; its optional, but is incredibly useful for accessibility - screen readers read this description out to their users so they know what the image means
 - align: Image horizontal alignment; and also flow of text around image. Valid values: left, right
 - width/Height: Sets the width and height of the image.
 - src: Specifies the image path or source.

```
<h1>CryptoCurrency</h1><hr>
 notice that text doesnt flow aro ...
<p>
<p>A cryptocurrency is an encrypted data string that denotes a unit of currency.
...
<p>
<p>Bitcoin (฿) is a decentralized digital curr
single administrator, that can be sent from us
```

CryptoCurrency



notice that text doesn't flow around this image. Some more blah, blah, blah, blah, blah, blah, blah, blah, blah, blah



A cryptocurrency is an encrypted data string that denotes a unit of currency. It is monitored and organized by a peer-to-peer network called a blockchain, which also serves as a secure ledger of transactions, e.g., buying, selling, and transferring. Unlike physical money, cryptocurrencies are decentralized, which means they are not issued by governments or other financial institutions.

Bitcoin (฿) is a decentralized digital currency, without a central bank or single administrator, that can be sent from user to user on the peer-to-peer bitcoin network without the need for intermediaries. Transactions are verified by network nodes through cryptography and recorded in a public distributed ledger called a blockchain. The cryptocurrency was invented in 2008 by an unknown person or group of people using the name Satoshi Nakamoto.



Table

- Use `<table>` tag to create a table.
- Table Attributes:
 - Border: applies border to table. Eg : `<table border="2">.....</table>`
 - Align: defines the horizontal alignment of the table element.
 - Values of the align attribute are right, left and center. `<table align="center">`
 - Width: defines the width of the table element.
 - `<table width="75%">.....</table>`
 - `<table width="400">.....</table>`
- Example:

```
<table border="1">
  <tr>
    <td>Row-1, cell-1</td>
    <td>Row-1, cell-2</td>
  </tr>
</table>
```

| | |
|---------------|---------------|
| Row 1, cell 1 | Row 1, cell 2 |
|---------------|---------------|

- To add a caption to a table, use the `<caption>` tag:
 - The `<caption>` tag must be inserted immediately after the `<table>` tag

```
<table style="width:100%">
  <caption>Monthly savings</caption>
  <tr><th>Month</th><th>Savings</th></tr>
  ...
</table>
```

Table Data

- An HTML table has two kinds of cells:
 - Header Cells `<th>`: Contain header information; The text is bold and centered.
 - Standard Cells `<td>`: Contain data; The text is regular and left-aligned.

```
<table>
  <tr> <th>Column1 Header</th> <th>Column2 Header</th></tr>
  <tr> <td>Cell 1,1</td> <td>Cell 1,2</td> </tr>
  <tr> <td>Cell 2,1</td> <td>Cell 2,2</td> </tr>
</table>
```

| Column1 Header | Column2 Header |
|----------------|----------------|
| Cell 1,1 | Cell 1,2 |
| Cell 2,1 | Cell 2,2 |

- You can insert a `bgcolor` attribute in a `<table>`, `<td>`, `<th>` or `<tr>` tag to set the color of the particular element.

```
<table bgcolor="cyan">
  <tr bgcolor="blue">
    <th bgcolor="red">Header 1</th><th>Header 2</th>
  </tr>
  <tr>
    <td bgcolor="green">data 1</td><td>data 2</td>
  </tr>
</table>
```

| | |
|----------|----------|
| Header 1 | Header 2 |
| data 1 | data 2 |

Cell Spanning

- colspan="number of columns"
 - By default, the number of columns in a table is defined by the number of table data cells appearing in the table row that contains the most data.
 - Ideally, place the same number of data cells in each table row. If a table row does not contain the requisite number of table cells, then it will essentially be in 'error' and will be displayed with a missing cell.
- rowspan="number of rows"
 - Forces a table cell to span the number of rows specified by the given value.

```
<table border=1>
  <tr>
    <th>Column 1</th><th>Column 2</th><th>Column 3</th>
  </tr>
  <tr>
    <td rowspan=2>Column 1 Data</td>
    <td align=center colspan=2>Col 2, Row 1 Data</td>
  </tr>
  <tr>
    <td>Col 2, Row 2 Data</td><td>Col 3, Row 2 Data</td>
  </tr>
</table>
```

| Column 1 | Column 2 | Column 3 |
|---------------|-------------------|-------------------|
| Column 1 Data | Col 2, Row 1 Data | |
| | Col 2, Row 2 Data | Col 3, Row 2 Data |

Hyperlink

- Hyperlinks access resources on the internet.
- Create a link with `` (anchor)

- `Login Here`
- Hello, Welcome to `My Site`
- I have some ` older information` about this subject.

[Login Here](#)

Hello, Welcome to [My Site](#)

I have some [older information](#) about this subject.

```

<ul>
  <li><a href='home.html'>mumbai</a></li>
  <li><a href='home.html'>pune</a></li>
  <li><a href='home.html'>nasik</a></li>
</ul>

```

- [mumbai](#)
- [pune](#)
- [nasik](#)

| team | points | grade |
|------------------------------------------------------------------------------------|--------|-------|
| <a >mumbai<="" a><="" href="home.html" td=""> <td>90</td> <td>a</td> | 90 | a |
| <a >pune<="" a><="" href="home.html" td=""> <td>86</td> <td>b</td> | 86 | b |
| <a >nasik<="" a><="" href="home.html" td=""> <td>80</td> <td>c</td> | 80 | c |

| team | points | grade |
|---------------------------------------------------------------------------------|--------|-------|
| <a >mumbai<="" a><="" href="home.html" td=""><td>90</td><td>a</td> | 90 | a |
| <a >pune<="" a><="" href="home.html" td=""><td>86</td><td>b</td> | 86 | b |
| <a >nasik<="" a><="" href="home.html" td=""><td>80</td><td>c</td> | 80 | c |

Use of Image as a Hyperlink

- Images used as hyperlinks:
 -
- Images contained within a table:

```


| Product | Cost | Image                       |
|---------|------|-----------------------------|
| Pencil  | \$8  |      |
| Brush   | \$15 |  |
| Pin     | \$3  |         |


```

| Product | Cost | Image |
|---------|------|-------|
| Pencil | \$8 | |
| Brush | \$15 | |
| Pin | \$3 | |

HTML Forms for User Input

- Data Submission using a Form
 - HTML forms are used to accept of user input.
 - A form contains form elements.
 - Form elements are elements that allow users to enter information in a form.
 - Define a form with the `<form>` tag.
- `<input>` tag is used to create form input fields.
 - Type attribute of `<input>` tag specifies the field type

| | |
|----------------------------|------------------------------------------------|
| ○ Single line text box | <code><input type="text"></code> |
| ○ Password field | <code><input type="password"></code> |
| ○ Hidden field | <code><input type="hidden"></code> |
| ○ Radio button | <code><input type="radio"></code> |
| ○ Checkbox | <code><input type="checkbox"></code> |
| ○ File selector dialog box | <code><input type="file"></code> |
| ○ Button | <code><input type="button"></code> |
| ○ Submit/Reset | <code><input type="submit/reset"></code> |
- `<textarea>`
- `<select>`
- `<button>`

```
<form method="get/post" action="URL">
  Field definitions
</form>
```

Registration Form

| | |
|----------------------------------------------------------------------|-----------------------------------------|
| First Name : | <input type="text"/> |
| Last Name : | <input type="text"/> |
| Username : | <input type="text"/> |
| Password : | <input type="password"/> |
| Email : | <input type="text"/> |
| Mobile No : | <input type="text"/> |
| City: | <input type="button" value="Select ▾"/> |
| <input type="button" value="Register"/> Back to Home | |

Form elements

- Text fields: **Single Line** : used to type letters, numbers, etc. in a form.
 - <INPUT TYPE="type" NAME="name" SIZE="number" VALUE="value" maxlength=n>
 - Eg:

```
<form>
    First name: <input type="text" name="firstname" value="fname">
    Last name:<input type="text" name="lname">
</form>
```

- Text Area (Multiple Line Text Field)
 - A text area can hold an unlimited number of characters. Text renders in a fixed-width font (usually Courier).
 - You can specify text area size with cols and rows attributes.

```
<textarea name="name" rows="10" cols="50" [disabled] [readonly]>
    Default-Text
</textarea>
```

```
<textarea name="address" rows=5 cols=10>
    Please write your address
</textarea>
<textarea rows="4" cols="20">
```

- Password
 - <input type="password" name="name" size=n value="value" [disabled]>
 - Eg: Enter the password:<input type="password" name="passwd" size=20 value="abc">

Form elements

- Check box - Lets you select one or more options from a limited number of choices.

`<input type="checkbox" name="name" value="value" [checked] [disabled]> Checkbox Label`

- Content of value attribute is sent to the form's action URL.

```
<input type="checkbox" name="color1" value="0"/>Red  
<input type="checkbox" name="color3" value="1" checked>Green
```

- Radio Buttons

- `<input type="radio" name="name" value="value" [checked] [disabled]> Radio Button Label`
- Content of the value attribute is sent to the form's action URL.*

```
<form>  
<input type="radio" name="gender" value="male"/>Male<br>  
<input type="radio" name="gender" value="female"/> Female  
</form>
```

- Hidden form field - Allows to pass information between forms:

- `<input type="hidden" name="name" size="n" value="value"/>`
- `<input type="hidden" name="valid_user" size="5" value="yes"/>`

Form elements

- File Selector Dialog Box
 - `<input type="file" name="name" size="width of field" value="value">`

```
<FORM>
Select file to upload:
<INPUT name="file" type="file"> <BR>
<INPUT type="submit" >
</FORM>
```

Select file to upload: No file chosen

- Buttons:
 - To add a button to a form use:
 - `<input type="button" name="btnCalculate" value="Calculate"/>`
 - To submit the contents of the form use:
 - `<input type="submit" name="btnSubmit" value="Submit"/>`
 - To reset the field contents use:
 - `<input type="reset" name="btnReset" value="Reset"/>`
 - You can also create button using `<button>` tag ; defines a push button. Inside this element you can put content, such as text or images.
 - `<button type="button">Click Me!</button>`

Drop-Down List

```
<select name="name" multiple="true/false" size=n [disabled]>
    <option [selected] [disabled] [value]>Option 1</option>...
</select>
```

- Multiple: States if multiple element selection is allowed.
- Size: Number of visible elements.
- Disabled: States if the option is to be disabled after it first loads.

```
<form>
<select multiple size="3" name="pref">
    <option value="ih" selected>Internet-HTML</option>
    <option value="js">Javascript</option>
    <option value="vbs">VBscript</option>
    <option value="as">ASP</option>
</select>
</form>
```



- Forms with labels

```
<form>
    <label for="uname">User name : </label>
    <input id="uname" name="username">
    <button>Submit</button>
</form>
```

User name :

<fieldset> and <legend>

- Group related elements in a form

```
<form>
<fieldset>
  <legend>Personal Details:</legend>
  Name: <input type="text"><br><br>
  Email: <input type="text"><br>
</fieldset>
</form>
```

Personal Details:

Name:

Email:

Your details

1. Name
2. Email
3. Phone

Delivery address

1. Address
2. Post code
3. Country

Card details

1. Card type
 1. VISA
 2. AmEx
 3. Mastercard

2. Card number

3. Security code

4. Name on card

iFrame

- An HTML iframe is used to display a web page within a web page.
 - specifies an inline frame.
 - Use the height and width attributes to specify the size of the iframe
 - With CSS, you can also change the size, style and color of the iframe's border
Eg <iframe src="" style="border:none;">
 - Alternatively use the frameborder attribute : 1 (yes) or 0 (no)

```
<html>
  <body>
    <p>Some HTML text before iframe display</p>
    <iframe src="numberedList.html"
            width="20%" height="200">
      Your browser doesn't support inline frames.
    </iframe>
    <p> Some text after iframe display </p>
  </body>
</html>
```

Some HTML text before iframe display

My favorite cricket teams

- INDIA
- SRILANKA
- PAKISTAN
- AUSTRALIA
- SOUTH AFRICA

Some text after iframe display

Formatting tags

| Tag | Example | Results |
|----------|------------------------------|--------------------------------------|
| | Bold Text | An example of Bold Text |
| <big> | <big>Big Text</big> | An example of Big Text |
| <center> | <center>Center Text</center> | An example of Center Text |
| | Emphasized Text | An example of <i>Emphasized Text</i> |
| <i> | <i>Italic Text</i> | An example of <i>Italic Text</i> |
| <small> | <small>Small Text</small> | An example of Small Text |
| | Strong Text | An example of Strong Text |
| <sub> | _{Subscript Text} | An example of Subscript Text |
| <sup> | ^{Superscript Text} | An example of Superscript Text |
| | Delete Text | An example of Delete Text |
| <s> | <s>Strike Text</s> | An example of Strike Text |
| <strike> | <strike>Strike Text</strike> | An example of Strike Text |
| <u> | <u>Underline Text</u> | An example of <u>Underline Text</u> |
| <tt> | <tt>Teletype Text</tt> | An example of Teletype Text |

HTML5

What's new in HTML5?

- HTML5 offers new enhanced set of tags
 - New Content Tags : <nav>, <section>, <header>, <article>, <aside>, <summary>
 - New Media Tags : <video>, <audio>
 - New Dynamic drawing : <canvas> graphic tag
 - New form controls, like calendar, date, time, email, url, search
- Support for JavaScript APIs
 - Canvas element for 2D drawing API
 - Video and audio APIs
 - APIs to support offline storages
 - The Drag & Drop APIs
 - The Geolocation API
 - Web workers, WebSQL etc
- The DOCTYPE tells the browser which type and version of document to expect.
 - The DOCTYPE announcement is not a HTML label; it is a guideline to the web program about what variant of HTML the page is composed in.

```
<!DOCTYPE html>
```

HTML5 Attributes for <input>

- A Form is one of the most basic and essential feature of any web site
 - HTML5 brings 13 new input types and 14 new attributes
 - HTML5 introduces these data types via the <input type="_NEW_TYPE_HERE_" /> format
- **Placeholder** - A placeholder is a textbox that hold a text in lighter shade when there is no value and not focused
 - <input id="first_name" placeholder="This is a placeholder">
 - Once the textbox gets focus, the text goes off and you shall input your own text
- **AutoFocus** - Autofocus is a Boolean attribute of form field that make browser set focus on it when a page is loaded
 - <input id ="Text2" type="text" autofocus/>
- **Required** - A "Required Field" is a field that must be filled in with value before submission of a form
 - <input name="name" type="text" required />

Enter Name Please fill out this field.

New Form elements

- **Email** - Checks whether the string entered by the user is valid email id or not.
 - `<input id="email" name="email" type="email" />`
- **Search** - used for search fields (behaves like a regular text field).
 - `<input id="mysearch" type="search" />`
- **Tel** - used for input fields that should contain a telephone number.
 - `<input type="tel" name="usrtel">`
 - `<input type="tel" name="phone" pattern="[2-9][0-9]{2}-[0-9]{3}-[0-9]{4}" title="North American format: XXX-XXX-XXXX">`
- **url** - is used for input fields that should contain a URL address.
 - Depending on browser support, the url field can be automatically validated when submitted.
- **color** – displays a color palette
- **Number** - used for input fields that should contain a numeric value.
 - Min and max parameters provided to limit the values.
 - Browser will treat it as simple textfield if it doesn't support this type.
 - `<input id="movie" type="number" value="0"/>`
 - `<input type="number" min="0" max="50" step="2" value="6">`

New Form elements

- **Range** - used for input fields that should contain a value within a range

- Browser will treat it as simple textfield if it doesn't support this type
- <input id="test" type="range"/>
- <input type="range" min="1" max="20" value="0">

- **Date** - used for input fields that should contain a date.

- Depending on browser support, a date picker can show up in the input field.
- <input id="meeting" type="date" />

- **month** - Selects month and year
- **week** - Selects week and year
- **time** - Selects time (hour and minute)
- **datetime** - Selects time, date, month and year
- **datetime-local** - Selects time, date, month and year (local time)



- <input type="button">
- <input type="checkbox">
- <input type="color">
- <input type="date">
- <input type="datetime-local">
- <input type="email">
- <input type="file">
- <input type="hidden">
- <input type="image">
- <input type="month">
- <input type="number">
- <input type="password">
- <input type="radio">
- <input type="range">
- <input type="reset">
- <input type="search">
- <input type="submit">
- <input type="tel">
- <input type="text">
- <input type="time">
- <input type="url">
- <input type="week">

Built-in form validation

- **required**: Specifies whether a form field needs to be filled in before the form can be submitted.
- **minlength** and **maxlength**: Specifies the minimum and maximum length of textual data (strings).
- **min** and **max**: Specifies the minimum and maximum values of numerical input types.
- **type**: Specifies whether the data needs to be a number, an email address, or some other specific preset type.
- **pattern**: Specifies a regular expression that defines a pattern the entered data needs to follow.

```
<form>
  <label for="uname">User Name</label>
  <input id="uname" name="uname" required pattern="[a-zA-Z]*">
  <button>Submit</button>
</form>
```

HTML5 Attributes for <input>

- **Pattern** : specifies a JavaScript regular expression for the field's value to be checked against. pattern makes it easy for us to implement specific validation for product codes, invoice numbers, and so on.

```
<label>Product Number:</label>
<input type="text"
       name="product"
       pattern="[0-9][A-Z]{3}"
       title="Single digit followed by three uppercase letters."/>
/>
```

- The title attribute **specifies extra information about an element**. The information is most often shown as a tooltip text when the mouse moves over the element.

The screenshot shows a web form with several input fields:

- A text input field containing "Shrilata".
- A text input field containing "shrilata@gmail.com".
- A text input field containing ".....".
- An input field labeled "Product Number:" containing "aa".
- A "submit" button.

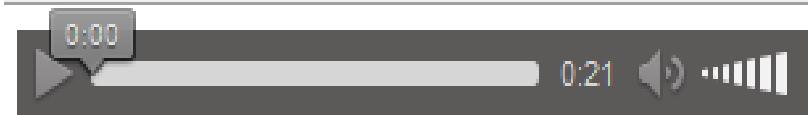
A tooltip is displayed below the "Product Number:" field, indicating a validation error:

Please match the requested format.
Single digit followed by three uppercase letters.

Audio and video

- Until now, there has never been a standard for playing audio on a web page.
 - Today, most audio is played through a audio plugin (like Microsoft Windows Media player, Microsoft Silverlight ,Apple QuickTime and the famous Adobe Flash).
 - However, not all browsers have the same plugins.
 - HTML5 the audio element to play sound files, or an audio stream.
 - Other properties like auto play, loop, preload area also available

```
<audio controls>
  <source src="vincent.mp3" type="audio/mpeg"/>
  <source src="vincent.ogg" type="audio/ogg"/>
</audio>
```



- Today, most videos are shown through a plugin (like Flash). However, not all browsers have the same plugins.
 - HTML5 provides `<video>` element to include video
 - Supported video formats for the video element : Ogg, MP4, WebM, .flv, .avi
 - Attributes : width, height, poster, autoplay, controls, loop, src

```
<video controls="controls" width="640" height="480" src="bunny.mp4" />
Your browser does not support the video element.
</video>
```

Canvas

- A canvas is a rectangle in your web page within which you can use JavaScript to draw shapes
 - Canvas can be used to represent something visually in your browser like Simple Diagrams, Fancy user interfaces, Animations, Charts and graphs, Embedded drawing applications, Working around CSS limitations
 - The canvas element has several methods for drawing paths, boxes, circles, characters, and adding images.
 - The canvas element has no drawing abilities of its own. All drawing must be done inside a JavaScript

```
<canvas id="myCanvas" width="200" height="100">  
</canvas>
```

```
<canvas id="myCanvas"></canvas>  
<script type="text/javascript">  
    var canvas=document.getElementById('myCanvas');  
    var ctx=canvas.getContext('2d');  
    ctx.fillStyle='#FF0000';  
    ctx.fillRect(0,0,80,100);  
</script>
```



HTML5 Training

HTML5
Canvas

Canvas examples

```
<script>
function draw(){
    var canvas=document.getElementById('myCanvas');
    var context=canvas.getContext('2d');
    context.strokeStyle = "red";
    context.fillStyle = "blue";
    context.fillRect(10,10,100,100);
    context.strokeRect(10,10,100,100);
}
</script>
```

```
var ctx=c.getContext("2d");
ctx.moveTo(10,10);
ctx.lineTo(150,50);
ctx.lineTo(10,50);
ctx.stroke();
```



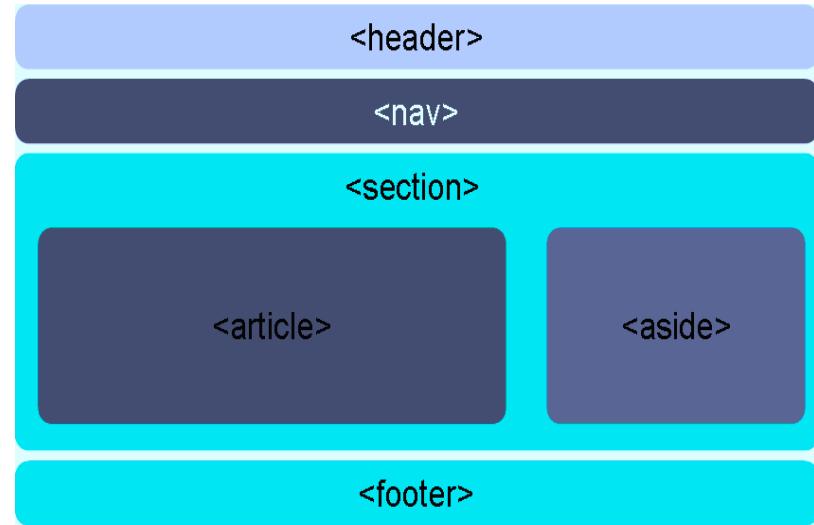
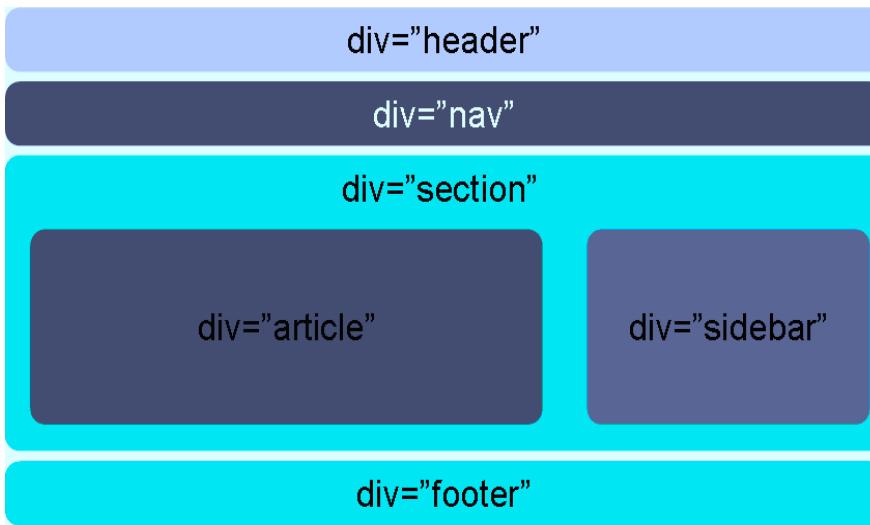
```
var ctx=c.getContext("2d");
var grd=ctx.createLinearGradient(0,0,175,50);
grd.addColorStop(0,"#FF0000");
grd.addColorStop(1,"#00FF00");
ctx.fillStyle=grd;
ctx.fillRect(0,0,175,50);
```

```
var ctx=c.getContext("2d");
ctx.fillStyle="#FF0000";
ctx.beginPath();
ctx.arc(70,18,15,0,Math.PI*2,true);
ctx.closePath();
ctx.fill();
```



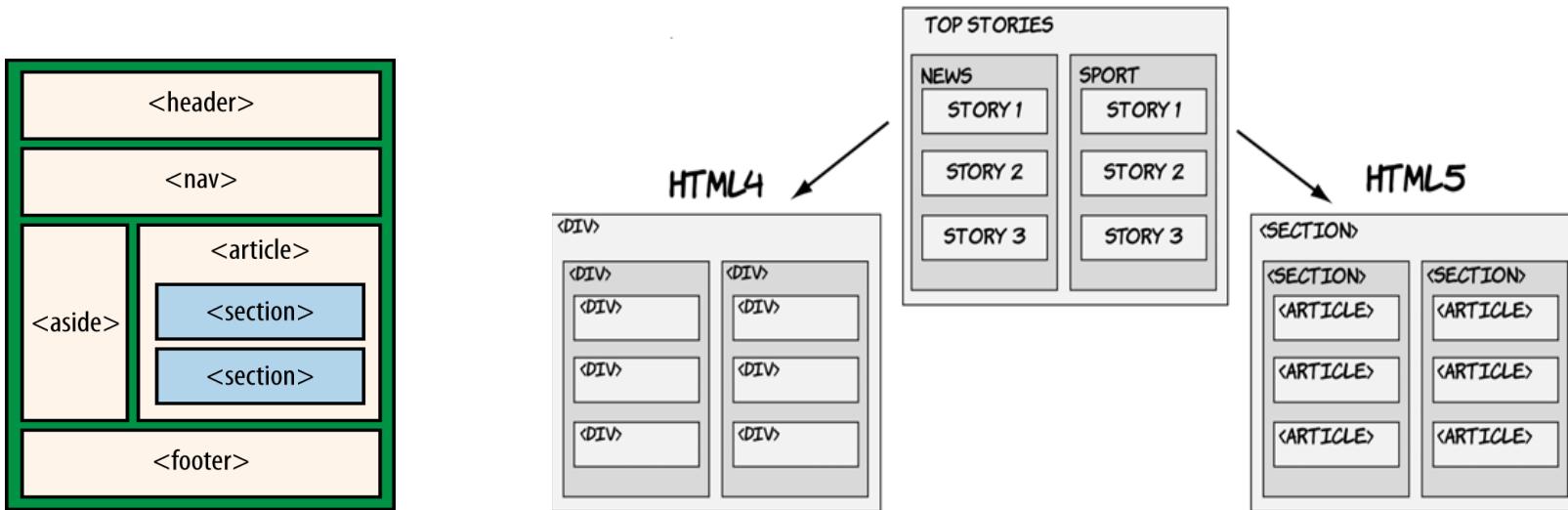
Laying out a page with HTML5

- Most HTML 4 pages include a variety of common structures, such as headers, footers and columns
- It's common to mark them up using div elements, giving each a descriptive id or class
- HTML 5 addresses this issue by introducing new elements for representing each of these different sections
- Elements that make it much easier to structure pages



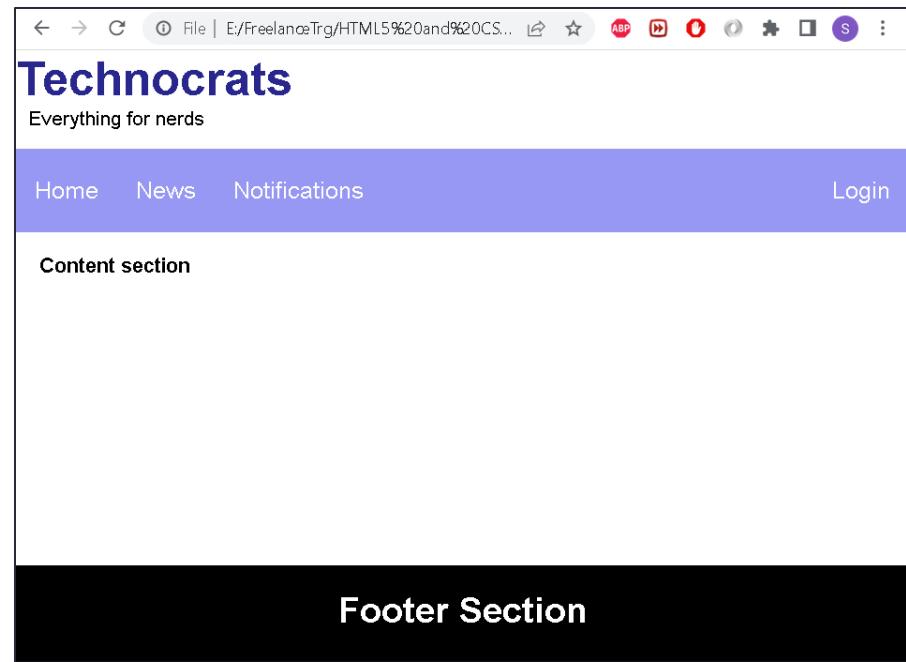
New Semantic Elements

- **<section>** : can be used to thematically group content, typically with a heading.



- **<article>**: element represents a self-contained composition in a document, page, application, or site that is intended to be independently distributable or reusable
 - Eg a forum post, a magazine or newspaper article, a blog entry, a user-submitted comment
- **<nav>**: Represents a major navigation block. It groups links to other pages or to parts of the current page.

```
<body>
  <!-- Header Section -->
  <header>
    <div class="head1">
      Technocrats
    </div>
    <div class="head2">
      Everything for nerds
    </div>
  </header>
  <!-- Menu Navigation Bar -->
  <nav class="menu">
    <a href="#home">Home</a>
    <a href="#news">News</a>
    <a href="#notification">
      Notifications
    </a>
    <div class="menu-log">
      <a href="#login">Login</a>
    </div>
  </nav>
  <!-- Body section -->
  <main class="body_sec">
    <section id="Content">
      <h3>Content section</h3>
    </section>
  </main>
  <!-- Footer Section -->
  <footer>Footer Section</footer>
</body>
```



blocking elements2.html

New Semantic Elements

- **<Header>**: tag specifies a header for a document or section. Can also be used as a heading of an blog entry or news article as every article has its title and published date and time
- **<aside>**: The "aside" element is a section that somehow related to main content, but it can be separate from that content header and footer element in an article.
- **<footer>**: Similarly to "header" element, "footer" element is often referred to the footer of a web page.
 - However, you can have a footer in every section, or every article too
- **<figure>**: The <figure> tag specifies self-contained content, like illustrations, diagrams, photos, code listings, etc.
 - This element can optionally contain a figcaption element to denote a caption for the figure.

```
<figure>
  
  <figcaption>Picture of the fave fruits</figcaption>
</figure>
```



Picture of the fave fruits

```
<body>
  <header><h1>The Times Today</h1>
  <nav>
    <ul>
      <li><a href="#">City</a></li>
      <li><a href="#">India</a></li>
      <li><a href="#">World</a></li>
      <li><a href="#">Business</a></li>
      <li><a href="#">Entertainment</a></li>
    </ul>
  </nav>
</header>
```

```
<section>
  <header style = "background-color: #607d8b70;">
    <h1 style = "color: #2a13dbf6;">Top Stories</h1>
  </header>

  <article>
    <h4 style = "color: #a80f0ff6;">Telcos companies de...
      NEW DELHI: Telecom companies have gone into a nea...
    </article>

  <article>
    <h4 style = "color: #a80f0ff6;">Fresh push for Isra...
      NEW DELHI: India is readying a slew of military d...
      The pacts include the acquisition of 164 laser-de...
    </article>
  <br>
  <footer class="footer"> Copyright 2016</footer>
</section>
```

sectionarticle-eg



```
<style>
  li{
    display: inline-flex;
    padding: 10px
  }
  .footer{
    height: 30px;
    color: #rgb(118, 216, 219);
    background-color: #455e64;
    text-align: center;
    padding-top: 10px;
  }
</style>
```

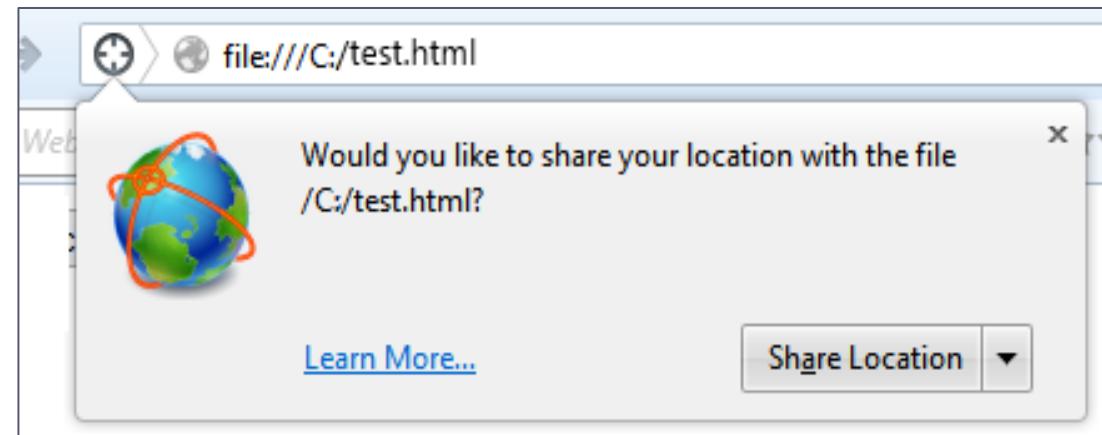
The screenshot shows a web browser window displaying the 'The Times Today' website. The header features a dark grey bar with the site's name in white. Below it is a navigation menu with links for City, India, World, Business, and Entertainment. A large, semi-transparent overlay covers the main content area, featuring a yellow-to-orange gradient starburst shape in the top right corner with the word 'DEMO' in white. The main content area has a light grey background and displays two news articles. The first article is about 'Telcos companies descend into chaos over Trai order' and includes a snippet of text: 'NEW DELHI: Telecom companies have gone into a nearchaos after Trai's regulation ordering equality on data prices. They will need to eliminate nearly hundreds of tariff plans that offer free or subsidised access to numerous popular websites and applications such as Facebook, WhatsApp and Twitter as these fall against Trai's new order.' The second article is about 'Fresh push for Israeli missile system' and includes a snippet: 'NEW DELHI: India is readying a slew of military deals with Israel worth \$3 billion before Prime Minister Narendra Modi visits Tel Aviv this year. The pacts include the acquisition of 164 laser-designation pods or 'Litening-4' for IAF fighters like Sukhoi-30MKIs and Jaguars as well as 250 advanced 'Spice' precision stand-off bombs capable of taking out fortified enemy underground command centres.' At the bottom of the page is a dark footer bar with the text 'Copyright 2016'.

Geo Location API

- geolocation is best described as the determination of the geographic position of a person, place, or thing
- Geolocation API is used to locate a user's position
 - It also keeps the track of as they move around, always with the user's consent
 - The API is device-agnostic; it doesn't care how the browser determines location, so long as clients can request and receive location data in a standard way
 - The underlying mechanism might be via GPS, wifi, or simply asking the user to enter their location manually
 - Since any of these lookups is going to take some time, the API is asynchronous; you pass it a callback method whenever you request a location



- Since the nature of the API exposes the user's location, it could compromise their privacy.
- So the user's permission to attempt to obtain the geolocation information must be sought before proceeding



Using the Geolocation API

- Test for the presence of the geolocation object:

```
if (navigator.geolocation) // check for Geolocation support  
    console.log('Geolocation is supported!');  
else  console.log('Geolocation is not supported for this Browser/OS version yet.');
```

- Obtain the geolocation object
- Geolocation Methods

```
var geolocation = navigator.geolocation;
```

- `getCurrentPosition()` : retrieves the current geographic location of the user.
- `watchPosition()` : retrieves periodic updates about the current geographic location of the device.
- `clearWatch()` : cancels an ongoing `watchPosition` call.

```
function getLocation() {  
    var geolocation = navigator.geolocation;  
    geolocation.getCurrentPosition(showLocation, errorHandler);  
}
```

showLocation and errorHandler are callback methods which would be used to get actual position

Using the Geolocation API

- `getCurrentPosition()` method is called asynchronously with an object **Position** which stores the complete location information.
 - The **Position** object specifies the current geographic location of the device

```
function showLocation( position ) {  
    var latitude = position.coords.latitude;  
    var longitude = position.coords.longitude;  
    ...  
}
```

```
function errorHandler( err ) {  
    if (err.code == 1) {  
        // access is denied  
    }  
    ...  
}
```

We need to catch any error and handle it gracefully

```
<body>  
<p id="demo">Click the button to get your coordinates:</p>  
<button onclick="getLocation()">Try It</button>  
<script>  
var x=document.getElementById("demo");  
function getLocation() {  
    if (navigator.geolocation)  
        navigator.geolocation.getCurrentPosition(showPosition);  
    else{  
        x.innerHTML="Geolocation is not supported by this browser.";  
    }  
    function showPosition(position){  
        x.innerHTML = "Latitude: " + position.coords.latitude +  
                    "<br />Longitude: " + position.coords.longitude;  
    }  
</script>
```

Click the button to get your coordinates:

Try It

Latitude: 18.5680291
Longitude: 73.8041752

Try It

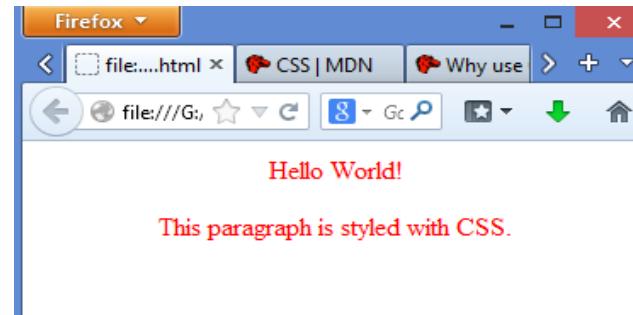
(CSS) Cascading Style Sheets

What is CSS?

- Cascading Styles Sheets - a way to style and present HTML.
 - HTML deals with content & structure, stylesheet deals with formatting & presentation of that document.
 - Allows to control the style and layout of multiple Web pages all at once.
- Why CSS?
 - saves time
 - Pages load faster
 - Easy maintenance
 - Superior styles to HTML
- A CSS rule has two parts: a selector, and one or more declarations:
- "HTML tag" { "CSS Property" : "Value" ; }
- The selector is normally the HTML element you want to style.
- Example:

```
<head>
<style>
p { color:red; text-align:center;  }
</style>
</head>
<body>
<p>Hello World!</p>
<p>This paragraph is styled with CSS.</p>
</body>
```

```
body{ background-color: gray;}
p { color: blue; }
h3{ color: white; }
```



Three Ways to Insert CSS

- **Embedded Style Sheets**

- Style defined between <STYLE>..</STYLE> tags. <STYLE> tags appear either in <HEAD> section or between </HEAD> and <BODY> tags.

- **Linked Style Sheets**

- Separate files (extn .CSS) that are linked to a page with the <LINK> tag. Are referenced with a URL. Placing a single <LINK> tag within the <HEAD> tags links the page that needs these styles.

```
<head>
  <link rel="stylesheet" type="text/css" href="mystyle.css">
</head>
```

- **Inline Style Sheets**

- Only applies to the tag contents that contain it. Used to control a single tag element. Tag inherits style from its parent. Eg.
 - <p style="color:sienna;margin-left:20px">This is a paragraph.</p>
 - <p style="background: blue; color: white;">A new background and font color with inline CSS</p>

Demo: Link Style Sheet

```
<html>
<head>
<link rel=stylesheet href="linked_ex.css"
      type="text/css">
</head>
<body>
<h2>This is Level 2 Heading, with style</h2>
<h1>This is Level 1 Heading, with style</h1>
<h3>This is Level 3 Heading, with style</h3>
<h4>This is Level 4 Heading, without style</h4>
</body>
</html>
```

```
body { background: black;
       color:green
     }
h1 { background: orange;
      font-family: Arial, Impact;
      color: blue;
      font-size:30pt;
      text-align: center
    }
h2, h3 { background: gold;
      font-family: Arial, Impact;
      color:red }
```

This is Level 2 Heading, with style

This is Level 1 Heading, with style

This is Level 3 Heading, with style

This is Level 4 Heading, without style

linked_ex.css

Inline Style Sheet

- All style attribute are specified in the tag it self. It gives desired effect on that tag only. Doesn't affect any other HTML tag.

```
<body style="background: white; color:green">
<h2 style="background: gold; font-family: Arial, Impact; color:red">
This is Level 2 Heading, with style</h2>
<h1 style="background: orange; font-family: Arial, Impact; color: blue;font-size:30pt; text-align: center">This is Level 1 Heading, with style</h1>
<h3 style="background: gold; font-family: Arial, Impact;color:red">
This is Level 3 Heading, with style</h3>
<h4>This is Level 4 Heading, without style</h4>
<h1>This is again Level 1 heading with default styles</h1>
</body>
```

This is Level 2 Heading, with style

This is Level 1 Heading, with style

This is Level 3 Heading, with style

This is Level 4 Heading, without style

Multiple Style Sheets

- If some properties have been set for the same selector in different style sheets, the values will be inherited from the more specific style sheet.
 - For example, an external style sheet has these properties for the h3 selector, & an internal style sheet has these:

```
H3 { color:red;  
     text-align:left;  
     font-size:8pt;  
 }
```

```
H3 { text-align:right;  
      font-size:20pt;  
 }
```

- If the page with the internal style sheet also links to the external style sheet the properties for h3 will be:
 - color:red;
 - text-align:right;
 - font-size:20pt;

The color is inherited from the external style sheet and the text-alignment and the font-size is replaced by the internal style sheet.

Grouping Selectors

- In style sheets there are often elements with the same style.
 - H1 { color:green; }
 - h2 { color:green; }
 - P { color:green; }
- To minimize the code, you can group selectors by separating each selector with a comma. Example
 - h1,h2,p { color:green; }

Types of selectors

- HTML selectors <tag>
 - Used to define styles associated to HTML tags. – already seen!!!!
- Class selectors (.)
 - Used to define styles that can be used without redefining plain HTML tags.
- ID selectors (#)
 - Used to define styles relating to objects with a unique id

This text would be blue

The text would be in red
This is level 2 heading

Data 1

Data 2

This is Level 4 Heading, without style

This is again Level 1 heading with default styles

```
<head>
<style>
#para1 {
  text-align:center;
  color:red;
}
</style>
</head>
<body>
  <p id="para1">Hello World!</p>
  <p>This paragraph is not affected by the style.</p>
</body>
```

```
<head>
<style>
H1.myClass {color: blue}
.myOtherClass { color: red; text-align:center}
</style>
<body style="background: white; color:green">
  <H1 class="myClass">This text would be blue</H1>
  <p class="myOtherClass">The text would be in red</P>
  <H3 class="myOtherClass">This is level 2 heading</H3>
  <table class=myotherClass border width=100%>
    <td>Data 1</td><td>Data 2</td>
  </table>
  <h3>This is Level 4 Heading, without style</h3>
  <h1>This is again Level 1 heading with default styles</h1>
```

Hello World!
This paragraph is not affected by the style.

class Selector

- The class selector is used to specify a style for a group of elements.
 - The class selector uses the HTML class attribute, and is defined with a ". "
 - Can also affect only specific HTML elements. Ex. all p elements with class="center" will be center-aligned (see below)

```
<head>
  <style>
    .center { text-align:center; }
  </style>
</head>
<body>
  <h1 class="center">Center-aligned heading</h1>
  <p class="center">Center-aligned paragraph.</p>
</body>
```

Center-aligned heading

Center-aligned paragraph.

```
<style>
  p.center { text-align:center; }
</style>
</head>
<body>
  <h1 class="center">This heading will not be affected</h1>
  <p class="center">This paragraph will be center-aligned.</p>
```

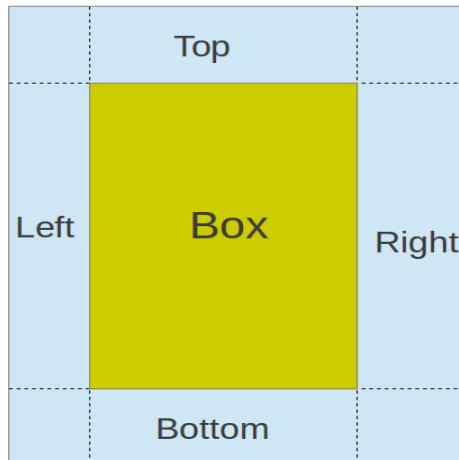
This heading will not be affected

This paragraph will be center-aligned.

CSS Box Model

- All HTML elements can be considered as boxes.
- The CSS box model is essentially a box that wraps around HTML elements, and it consists of:
 - **Margin** - Clears an area around the border. The margin does not have a background color, it is completely transparent
 - **Border** - A border that goes around the padding and content. The border is affected by the background color of the box
 - **Padding** - Clears an area around the content. The padding is affected by the background color of the box
 - **Content** - The content of the box, where text and images appear

```
<h1>Header 1</h1>  
<p> paragraph 1</p>
```



CSS Border

- The border property is a shorthand for the following individual border properties: border-width, border-style (required), border-color

```
p { border: 5px solid red; }
```

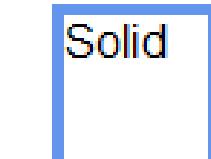
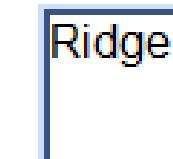
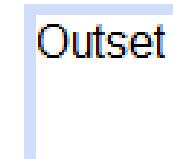
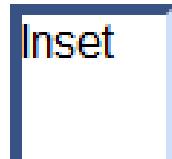
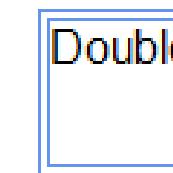
This is some text in a paragraph.

```
<style type="text/css">  
.box {  
    width: 100px;  
    height: 100px;  
    border-color: Blue;  
    border-width: 2px;  
    border-style: solid;  
}  
</style>
```

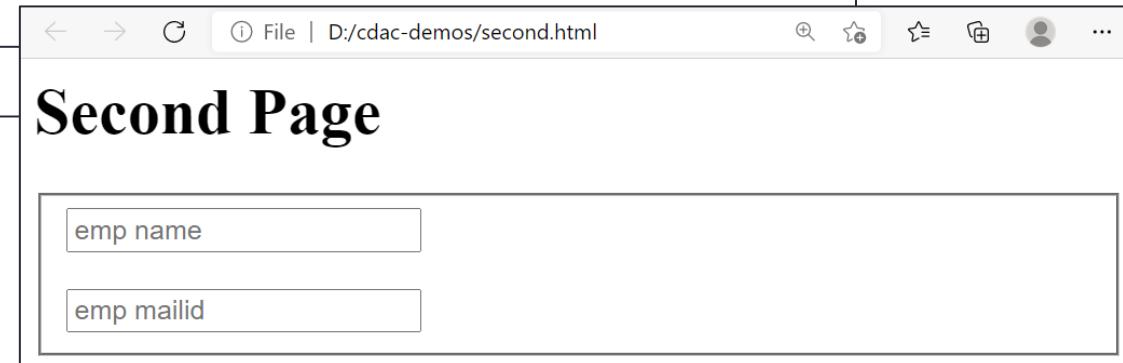
```
<div class="box">  
    Hello, world!  
</div>
```

```
<div class="box" style="border-style: dashed;">Dashed</div>  
<div class="box" style="border-style: dotted;">Dotted</div>  
<div class="box" style="border-style: double;">Double</div>  
<div class="box" style="border-style: groove;">Groove</div>  
<div class="box" style="border-style: inset;">Inset</div>  
<div class="box" style="border-style: outset;">Outset</div>  
<div class="box" style="border-style: ridge;">Ridge</div>  
<div class="box" style="border-style: solid;">Solid</div>
```

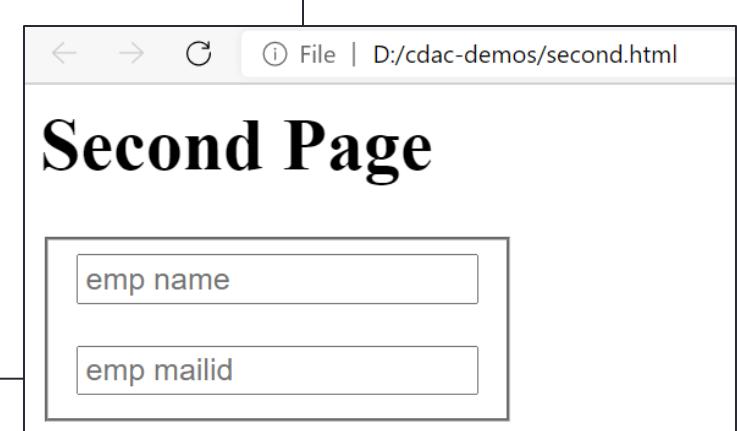
Hello, world!



```
<form>
  <fieldset>
    <input type="text" placeholder="emp name"><br><br>
    <input type="text" placeholder="emp mailid"><br>
  </fieldset>
</form>
```



```
<html>
<head>
  <style>
    fieldset{width:100px}
  </style>
</head>
<body>
  <h1> Second Page</h1>
  <form>
    <fieldset>
      <input type="text" placeholder="emp name">
      <input type="text" placeholder="emp mailid">
    </fieldset>
  </form>
</body>
</html>
```



Collapsing borders on table

```
table, th, td {  
    border: 1px solid black;  
    border-collapse: collapse;  
}
```

```
<table border="1">  
    <tr><th>First Name</th><th>Last Name</th></tr>  
    <tr><td>Anita</td><td>Patil</td></tr>  
    <tr><td>Kartik</td><td>Rao</td></tr>  
    <tr><td>Veena</td><td>Deshmukh</td></tr>  
</table>
```

| First Name | Last Name |
|------------|-----------|
| Anita | Patil |
| Kartik | Rao |
| Veena | Deshmukh |

```
<head>  
    <style>  
        table, th, td{border:1px solid black; border-collapse:collapse;}  
    </style>  
</head>  
<body>  
    <table border="1">  
        <tr><th>First Name</th><th>Last Name</th></tr>  
        <tr><td>Anita</td><td>Patil</td></tr>  
        <tr><td>Kartik</td><td>Rao</td></tr>  
        <tr><td>Veena</td><td>Deshmukh</td></tr>  
</table>
```

| First Name | Last Name |
|------------|-----------|
| Anita | Patil |
| Kartik | Rao |
| Veena | Deshmukh |

CSS3 border

- CSS 3 defines “border radius”, giving developers the possibility to make rounded corners on their elements.

```
<style>
#rcorners1 {
    border-radius: 25px;
    background: #8AC007;
    padding: 20px;
    width: 100px;
    height: 100px;
}
#rcorners2 {
    border-radius: 25px;
    border: 2px solid #8AC007;
    padding: 20px;
    width: 100px;
    height: 100px;
}
```

```
#rcorners3 {
    border-radius: 25px;
    background: url(paper.gif);
    background-position: left top;
    background-repeat: repeat;
    padding: 20px;
    width: 100px;
    height: 100px;
}
</style>
<p id="rcorners1">Rounded corners!</p>
<p id="rcorners2">Rounded corners!</p>
<p id="rcorners3">Rounded corners!</p>
```

Rounded corners!

Rounded corners!

Rounded corners!

CSS Styling

CSS Background

- CSS background properties are used to define the background effects of an element.
- CSS properties used for background effects:

| | |
|--------------------------------------------|--------------------------------------------------|
| <u>background-color</u> | Sets the background color of an element |
| <u>background-image</u> | Sets the background image for an element |
| <u>background-position</u> | Sets the starting position of a background image |
| <u>background-repeat</u> | Sets how a background image will be repeated |

- Example:

- `div {background-color:#b0c4de;}`
- `body {background-image:url('paper.gif');}`
- `body {background-image:url('gradient2.png');background-repeat:repeat-x;}`
- `body {background-image:url('img_tree.png'); background-repeat:no-repeat; background-position:right top; }`

With CSS, a color is specified by:

- *a HEX value - like "#ff0000"*
- *an RGB value - like "rgb(255,0,0)"*
- *a color name - like "red"*

- The `background-repeat` property sets if/how a background image will be repeated.
 - By default, (repeat) : a [background-image](#) is repeated both vertically and horizontally.
 - no-repeat : The background-image is not repeated. The image will only be shown once

Demo : CSS Background

```
body {  
    background-image: url("img_tree.gif"), url("img_flwr.gif");  
    background-color: #cccccc;  
}
```

```
body {  
    background: #00ff00 url("smiley.gif") no-repeat fixed center;  
}
```

```
<html>  
<head><style>  
body {  
background-image:url('img_tree.png');  
background-repeat:no-repeat;  
background-position:right top;  
margin-right:200px;  
}  
</style> </head>  
<body>  
<h1>Hello World!</h1>  
<p>Background no-repeat, set position example.</p>  
<p>Now the background image is only shown once, and positioned away from the text.</p>  
<p>In this example we have also added a margin on the right side, so the background image will never  
disturb the text.</p>  
</body></html>
```

Hello World!

Background no-repeat, set position example.

Now the background image is only shown once, and positioned away from the text.

In this example we have also added a margin on the right side, so the background image will never disturb the text.



CSS Text

- CSS Text Properties

| | |
|------------------------|---------------------------------------------------------------|
| <u>color</u> | Sets the color of text |
| <u>direction</u> | Specifies the text direction/writing direction |
| <u>letter-spacing</u> | Increases or decreases the space between characters in a text |
| <u>text-align</u> | Specifies the horizontal alignment of text |
| <u>text-decoration</u> | Specifies the decoration added to text |
| <u>text-indent</u> | Specifies the indentation of the first line in a text-block |
| <u>text-shadow</u> | Specifies the shadow effect added to text |
| <u>text-transform</u> | Controls the capitalization of text |
| <u>white-space</u> | Specifies how white-space inside an element is handled |
| <u>word-spacing</u> | Increases or decreases the space between words in a text |

```
<style>
h1 {text-decoration:overline;}
h2 {text-decoration:line-through;}
h3 {text-decoration:underline;}
</style>
<body>
<h1>This is heading 1</h1>
<h2>This is heading 2</h2>
<h3>This is heading 3</h3>
```

This is heading 1

This is heading 2

This is heading 3

Demo :

```
<style>
p.uppercase {text-transform:uppercase;}
p.lowercase {text-transform:lowercase;}
p.capitalize {text-transform:capitalize;}
</style>
<body>
<p class="uppercase">This is some text.</p>
<p class="lowercase">This is some text.</p>
<p class="capitalize">This is some text.</p>
</body>
```

THIS IS SOME TEXT.
this is some text.
This Is Some Text.

```
<head> <style>
h1 {text-align:center;color:#00ff00;}
p.date {text-align:right;}
p.main {text-align:justify;}
p.ex {color:rgb(0,0,255);}
p.indent {text-indent:50px;}
</style> </head>
<body>
<h1>Hello World!</h1>
<p class="date"> Sep 2013</p>
<p class="main indent">The CSS property text-align corresponds to the attribute align used in old versions of HTML. Text can either be aligned to the left, to the right or centred. In addition to this, the value justify will stretch each line so that both the right and left margins are straight. You know this layout from for example newspapers and magazines. </p>
<p class="ex">The property text-decoration makes it is possible to add different "decorations" or "effects" to text. </p>
</body>
```

Hello World!

Sep 2013

The CSS property text-align corresponds to the attribute align used in old versions of HTML. Text can either be aligned to the left, to the right or centred. In addition to this, the value justify will stretch each line so that both the right and left margins are straight. You know this layout from for example newspapers and magazines.

The property text-decoration makes it is possible to add different "decorations" or "effects" to text.

CSS : Styling Fonts

- CSS font properties define the font family, boldness, size, and the style of a text.

| | |
|-------------------------------------|--------------------------------------------------------------|
| <u>font-family</u> | Specifies the font family for text |
| <u>font-size</u> | Specifies the font size of text |
| <u>font-style</u> | Specifies the font style for text |
| <u>font-variant</u> | Specifies if a text should be displayed in a small-caps font |
| <u>font-weight</u> | Specifies the weight of a font |

- Example:

- p.normal {font-weight:normal;}
- p{font-family:"Times New Roman", Times;}
- p.italic {font-style:italic;}
- h1 {font-size:40px;}
- p.small { font-variant:small-caps; }

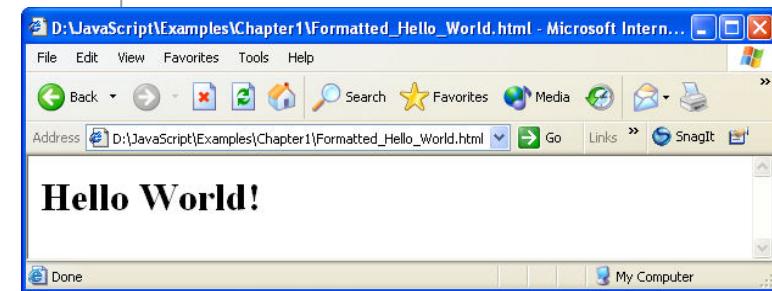
JAVASCRIPT

Overview

- JavaScript is Netscape's cross-platform, object-based scripting language
 - JavaScript code is embedded into HTML pages
 - It is a lightweight programming language
 - Client-side JavaScript extends the core language by supplying objects to control a browser and its Document Object Model
- Why use Javascript?
 - Provides HTML designers a programming tool :
 - Puts dynamic text into an HTML page
 - Reacts to events
 - Reads and writes to HTML elements :
 - Can be used to perform Client side validation

```
<html>
<head> </head>
<body>
<script type="text/javascript">
    document.write("<H1>Hello World!</H1>");
    alert("some message");
    console.log("some message");
</script>
</body></html>
```

```
<SCRIPT>
    JavaScript statements ...
</SCRIPT>
```



Embedding JavaScript in HTML

- Where to Write JavaScript?
 - Head Section
 - Body Section
 - External File

```
<html>
<head></head>
<body >
<script language="javascript">
    document.write("Hello World!")
</script>
</body>
</html>
```

```
<html>
<head>
<script type="text/javascript">
    function message() {
        alert("Hello World")
    }
</script>
</head>
<body onload="message()">
</body>
</html>
```

```
<head>
<script src="common.js">
    
</script>
</head>
<body>
    <script>
        document.write("display value of a variable"+msg)
    </script>
</body>
```

//common.js file contents

```
var msg
msg="

# in external file

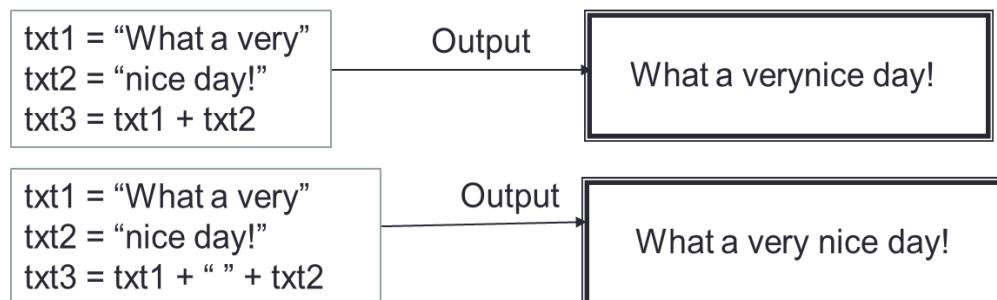
"
```

Data Types in JavaScript

- JavaScript is a free-form language. Need not declare all variables, classes, and methods
- Variables in JavaScript can be of type:
 - Number (4.156, 39)
 - String ("This is JavaScript")
 - Boolean (true or false)
 - Null (null) → usually used to indicate the absence of a value
- Defining variables. `var variableName = value`
- JavaScript variables are said to be un-typed or loosely typed
 - letters of the alphabet, digits 0-9 and the underscore (_) character and is case-sensitive.
 - Cannot include spaces or any other punctuation characters.
 - First character of name must be either a letter or the underscore character.
 - No official limit on the length of a variable name, but must fit within a line.

Javascript operators:

- Arithmetic Operators (+ , - , * , / , %)
- Assignment Operators(=,+=,-=,*=/,%=)
- Comparison Operators (==,!!=,<,<=,>,>=)
- Boolean Operators(&&,||,!)
- Bitwise Operators(&,|,!^,<<,>>,>>>)
- String Operators(=,+,+=)



Typeof Operator

| | |
|-------------------------------|-------------|
| x | typeof x |
| undefined | "undefined" |
| null | "object" |
| true or false | "boolean" |
| any number or NaN | "number" |
| any string | "string" |
| any function | "function" |
| any nonfunction native object | "object" |

| | | |
|--------|-------------------|-------------|
| typeof | undefinedvariable | "undefined" |
| typeof | 33 | "number" |
| typeof | "abcdef" | "string" |
| typeof | true | "boolean" |
| typeof | null | "object" |

| | |
|------------------------------|------------------------|
| typeof "John" | // Returns "string" |
| typeof 3.14 | // Returns "number" |
| typeof NaN | // Returns "number" |
| typeof false | // Returns "boolean" |
| typeof [1,2,3,4] | // Returns "object" |
| typeof {name:'John', age:34} | // Returns "object" |
| typeof new Date() | // Returns "object" |
| typeof function () {} | // Returns "function" |
| typeof myCar | // Returns "undefined" |
| typeof null | // Returns "object" |

```
<SCRIPT LANGUAGE="JavaScript">
```

```
var num1=20
var str1="abc"
var bool1=true
var num2=null
var var1;
document.write("type of str1 : "+typeof(str1)+"<BR>")
document.write("type of num1 : "+typeof(num1)+"<BR>")
document.write("type of bool1 : "+typeof(bool1)+"<BR>")
document.write("type of num2 : "+typeof(num2)+"<BR>")
document.write("type of var1 : "+typeof(var1)+"<BR>")
</SCRIPT>
```

```
type of str1 : string
type of num1 : number
type of bool1 : boolean
type of num2 : object
type of var1 : undefined
```

Control Structures and Loops

- JavaScript supports the usual control structures:

- the conditionals:

- if,
 - if...else
 - If ... else if ... else
 - Switch

```
if(condition) {  
    statement 1  
} else {  
    statement 2  
}
```

```
if(a>10) {  
    document.write("Greater than 10")  
} else {  
    document.write("Less than 10")  
}
```

```
document.write( (a>10) ? "Greater than 10" : "Less than 10" );
```

- iterations:

- for
 - while

```
<script>  
var n=20;  
switch(n){  
    case 10: document.write("Ten");  
              break;  
    case 20: document.write("Twenty");  
              break;  
    case 30: document.write("Thirty");  
              break;  
    default: document.write("Invalid!");  
}  
</script>
```

```
switch (variable) {  
    case outcome1 :{  
        //stmts for outcome 1  
        break; }  
    case outcome2 :{  
        //stmts outcome 2  
        break; }  
    default: {  
        //none of the outcomes is chosen  
    }  
}
```

```
for( [initial expression];[condition;][increment expression] ) {  
    statements  
}
```

```
for(var i=0;i<10;i++)  
{  
    document.write("Hello");  
}
```

```
while(condition) {  
    statements  
}
```

```
while(i<10) {  
    document.write("Hello");  
    i++;  
}
```

JavaScript Functions

```
function myFunction (arg1, arg2, arg3) {  
    statements  
    return  
}
```

```
function area(w1, w2, h) {  
    var area=(w1+w2)*h/2;  
    alert(area+" sq ft");  
}  
area(2,3,7); //calling the function
```

Calling the function :
myFunction("abc", "xyz", 4)
myFunction()

```
function diameter(radius){  
    return radius * 2;  
}
```

```
var d=diameter(5); //calling the function
```

- **Function expressions** - functions are assigned to variables

```
var myFunction = function() {  
    statements  
}
```

```
var area = function (radius) {  
    return Math.PI * radius * radius;  
};  
alert(area(5)); // => 78.5
```

- **Global and Local Variables**

```
<script language="Javascript">  
    var companyName="TechnoFlo"  
    function f(){  
        var empName="Henry"  
        document.write("Welcome to "+companyName+ ", "+empName)  
    }  
</script>
```

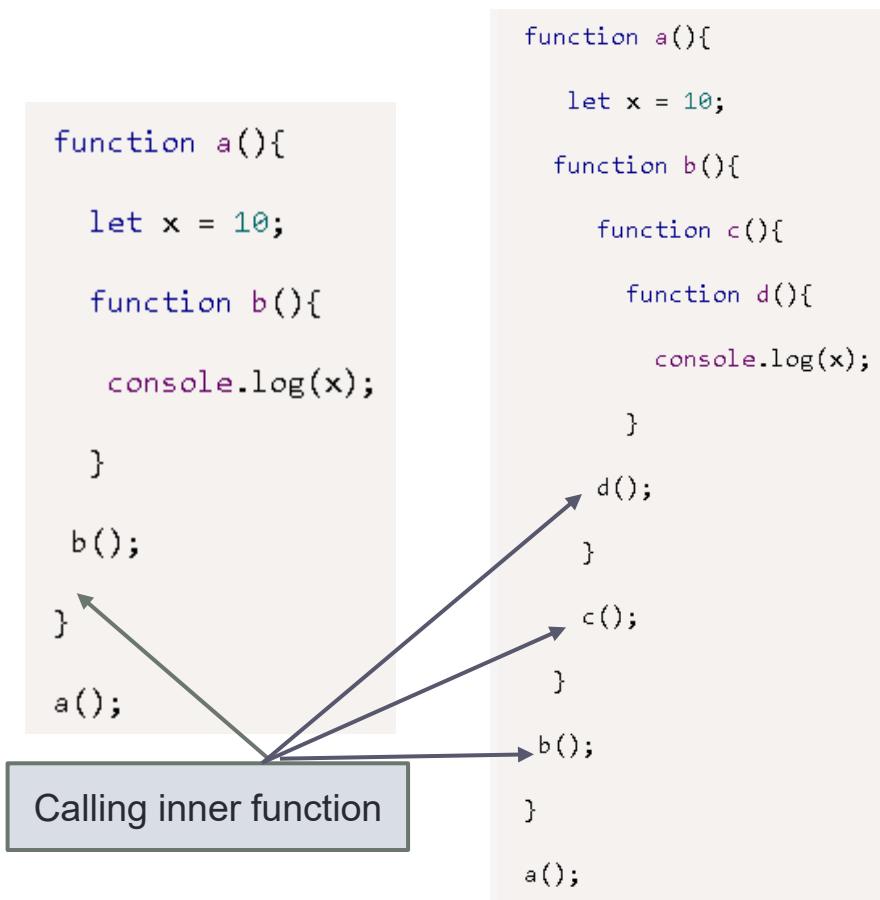
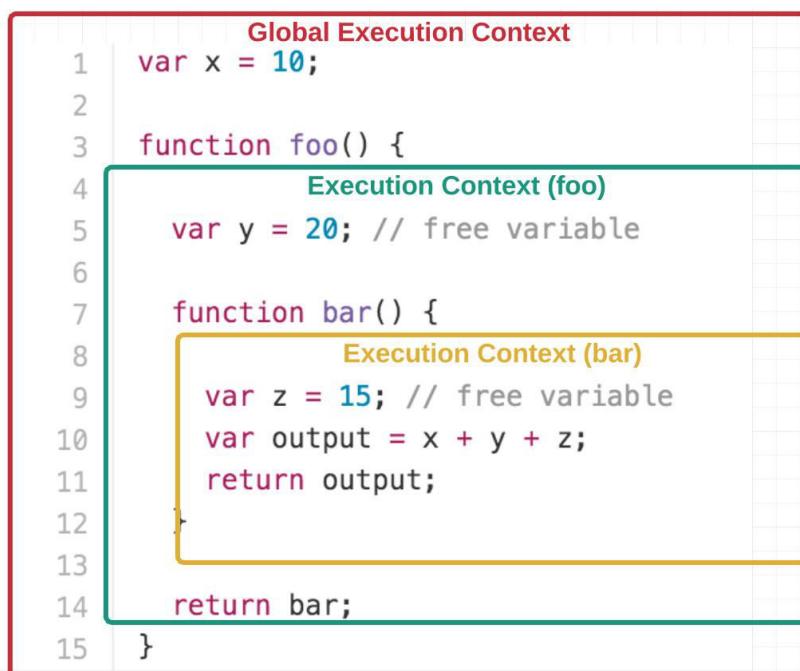
Global Variable

Local Variable

- Variables that exist only inside a function are called Local variables - they can't be changed by main code or other functions
- Within the body of a function, a local variable takes precedence over a global variable with the same name.
- Variables that exist throughout the script are called Global variables - Their values can be changed anytime in the code and even by other functions

Closures

- The closure is an inner function which always has access to the variables and parameters of its outer function, even when the outer function has returned.
 - The closure has three scope chains: it has access to its own scope (variables defined between its curly brackets), it has access to the outer function's variables, and it has access to the global variables.



Closures : Example-1

```
1. function sayHello() {  
2.     var say = function() {  
3.         console.log(hello);  
4.     }  
5.     // Local variable that goes into closure  
6.     var hello = 'Hello, world!';  
7.     return say;  
8. }  
9.  
10. var sayHelloClosure = sayHello();  
11. sayHelloClosure(); // 'Hello, world!'
```

- Focus on lines 10 and 11.
- At line number 10 we are done with the execution of function `sayHello()` and the entire body of function `say()` is returned and stored in `var sayHelloClosure`, due to the line 7 `return say`.
- *The return statement does not execute the inner function - function is executed only when followed by (), but rather the return statement returns the reference to the function as a function in JavaScript is also an object.*

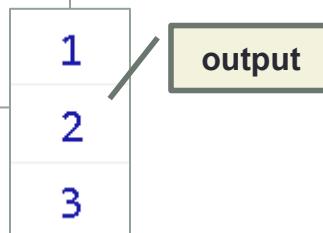
- We can access the variable `hello` which is defined in function `sayHello()` through function `say()`
- This is closure in action - that is inner function (`say()`) can have access to the outer function variables as well as all the global variables.
- closure is created when a child function keep the environment of the parent scope even after the parent function has already executed

Closures : Example-2

```
1. <html>
2.   <body>
3.     <script>
4.       function outer() {
5.         var i = 1;
6.         function inner(){
7.           return i++;
8.         }
9.         return inner;
10.      }

11.      var getInner = outer();
12.
13.      console.log(getInner());
14.      console.log(getInner());
15.      console.log(getInner());
16.    </script>
17.  </body>
18. </html>
```

- Focus on lines 11 to 15.
- At line number 11 we are done with the execution of **function outer()** and the entire body of **function inner()** is returned and stored in **var get_func_inner**, due to the line **9 return inner**.
- *The return statement does not execute the inner function - function is executed only when followed by (), but rather the return statement returns the reference to the function as a function in JavaScript is also an object.*



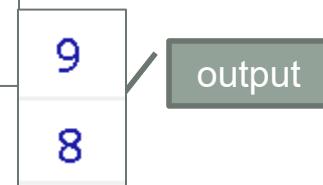
Closures : Example-3

```
<script>
    console.log("-----example-3-----")
    function outer(outer_arg) {
        function inner(inner_arg) {
            return outer_arg + inner_arg;
        }
        return inner;
    }

    var getInner = outer(5);

    console.log(getInner(4));
    console.log(getInner(3));
</script>
```

we used a parameter function rather than a default one. Note even when we are done with the execution of **outer(5)** we can access the **outer_arg** variable from the inner function. And on execution of inner function produce the summation of **outer_arg** and **inner_arg** as desired.



Predefined Functions

- **isFinite**: evaluates an argument to determine if it is a finite number.
- If needed, the parameter is first converted to a number.

```
isFinite (number) //where number is the number to evaluate
```

```
var a = isFinite(123) + "<br>"; //true  
var b = isFinite(-1.23) + "<br>"; //true  
var d = isFinite(0) + "<br>"; //true  
var e = isFinite("123") + "<br>"; //true  
var f = isFinite("Hello") + "<br>"; //false
```

- **isNaN** : Evaluates an argument to determine if it is “NaN” (not a number)

```
isNaN(0) //false  
isNaN('123') //false  
isNaN('Hello') //true  
isNaN('2005/12/12') //true
```

- **Parseint and parseFloat**

- Returns a numeric value for string argument.
- **parseInt (str)**
- **parseFloat (str)**

```
parseInt("3 blind mice") // => 3  
parseFloat(" 3.14 meters") // => 3.14
```

ES6 - Next gen Javascript

- **const** : from JS 1.5 onwards.- to define constants

- Eg :

```
const myBirthday = '18.04.1982';
myBirthday = '01.01.2001';    // error, can't reassign the constant!
```

```
const LANGUAGES = ['Js', 'Ruby', 'Python', 'Go'];
LANGUAGES = "Javascript";    // shows error.
LANGUAGES.push('Java');      // Works fine.
console.log(LANGUAGES);     // ['Js', 'Ruby', 'Python', 'Go', 'Java']
```

- **let** : to define block-scoped variables; can be used in four ways:

- as a variable declaration like var; in a for or for/in loop, as a substitute for var;
 - as a block statement, to define new variables and explicitly delimit their scope
 - to define variables that are scoped to a single expression.
 - Eg : let message = 'Hello!';
 - Eg : let user = 'John', age = 25, message = 'Hello';

```
if (true) {
  let a = 40;
  console.log(a); //40
}
console.log(a); // undefined
```

```
let a = 50;  let b = 100;
if (true) {
  let a = 60;
  var c = 10;
  console.log(a/c); // 6
  console.log(b/c); // 10
}
console.log(c); // 10
console.log(a); // 50
```

ES6 - Next gen Javascript

- Arrow functions : allows you to create functions in a cleaner way compared to regular functions
 - is a compact alternative to a traditional [function expression](#), but is limited and can't be used in all situations.

```
<script>
//non lambda
function greet(name) {
    console.log(name);
}
greet("shrilata");

//lambda-eg1
const greet1 = name => console.log(name);
greet1("sandeep");

//lambda-eg2
const add = (a,b) => a + b;

console.log(add(10,20));

//lambda-eg3
const strOp = str => {
    console.log(str.length);
    console.log(str.toUpperCase());
    console.log(str.charAt(0));
};

strOp("Hello");
```

Predefined Core Objects

String Objects

- Creating a string object:

- `var myString = new String("characters")`
- `var myString = "fred"`

- Properties of a string object:

- `length`: returns the number of characters in a string.

- `"Lincoln".length // result = 7`
- `"Four score".length // result = 10`
- `"One\n\two".length // result = 7`
- `"".length // result = 0`

- String Object methods:

- `charAt(index)` : returns the character at a specified position.

- Eg : `var str = "Hello world!";`
 - `str.charAt(0); //returns H`
 - `str.charAt(str.length-1)); //returns !`

- `concat()` : joins two or more strings
- `stringObject.concat(stringX,stringX,...,stringX)`
- Eg: `var str1="Hello ";`
`var str2="world!";`
`document.write(str1.concat(str2));`

String functions

- `indexOf ()` : returns the position of the first occurrence of a specified string value in a string.
 - index values start their count with 0.
 - If no match occurs within the main string, the returned value is -1.
 - `string.indexOf(searchString [, startIndex])`

```
Eg : var str="Hello world, welcome";
str.indexOf("Hello"); //returns 0
str.indexOf("wor")); //returns 6
str.indexOf("e",5); //returns 14
```

- `toLowerCase() / toUpperCase()`

```
Eg: var str="Hello World!";
str.toLowerCase() //returns hello world
str.toUpperCase() //returns HELLO WORLD
```

- `slice(startIndex [, endIndex])`
 - Extracts a part of a string and returns the extracted part in a new string

```
Eg : var str="Hello World";
      str.slice(6) //returns World
      str.slice(0,1) //returns H
```

String functions

- `split("delimiterCharacter"[, limitInteger])` - Splits a string into array of strings
 - `string.split("delimiterCharacter"[, limitInteger])`

```
var str = "zero one two three four";
var arr = str.split(" ");
for(i = 0; i < str.length; i++){ document.write("<br>" + arr[i]); }
```

```
var myString = "Anderson,Smith,Johnson,Washington"
var myArray = myString.split(",")
var itemCount = myArray.length // result: 4
```

Output :

zero
one
two
three
four

- Complete Example:

| | |
|-------------------------------------|------------------------------------------------|
| <code>var s = "hello, world"</code> | // Start with some text. |
| <code>s.charAt(0)</code> | // => "h": the first character. |
| <code>s.charAt(s.length-1)</code> | // => "d": the last character. |
| <code>s.substring(1,4)</code> | // => "ell": the 2nd, 3rd and 4th characters. |
| <code>s.slice(1,4)</code> | // => "ell": same thing |
| <code>s.slice(-3)</code> | // => "rld": last 3 characters |
| <code>s.indexOf("l")</code> | // => 2: position of first letter l. |
| <code>s.lastIndexOf("l")</code> | // => 10: position of last letter l. |
| <code>s.indexOf("l", 3)</code> | // => 3: position of first "l" at or after 3 |
| <code>s.split(", ")</code> | // => ["hello", "world"] split into substrings |
| <code>s.replace("h", "H")</code> | // => "Hello, world": replaces all instances |
| <code>s.toUpperCase()</code> | // => "HELLO, WORLD" |

String property : Prototype

- A prototype is a property or method that becomes a part of every new object created after the prototype items have been added.
 - Sometimes you want to add new properties/methods to all existing objects of a given type.
 - For strings, as an example, you may want to define a new method for converting a string into a new type of HTML font tag not already defined by JavaScript's string object.
 - A function definition (makeItHot()) accumulates string data to be returned to the object when the function is invoked as the object's method.
 - The this keyword extracts the object making the call, which you convert to a string for concatenation with the rest of the strings to be returned.
- prototype
 - Allows you to add properties and methods to an object.
 - Syntax : object.prototype.name=value

```
function makeItHot() {  
    return "<FONT COLOR='red'>" + this.toString() + "</FONT>"  
}  
  
String.prototype.hot = makeItHot  
document.write("<H1>This site is on " + "FIRE".hot() + "!!</H1>")
```

Math Properties

| Property | Value | Description |
|--------------|-----------------------|--------------------|
| Math.E | 2.718281828459045091 | Euler's constant |
| Math.LN | 0.6931471805599452862 | Natural log of 2 |
| Math.LN10 | 2.302585092994045901 | Natural log of 10 |
| Math.LOG2E | 1.442695040888963387 | Log base-2 of E |
| Math.LOG10E | 0.4342944819032518167 | Log base-10 of E |
| Math.PI | 3.141592653589793116 | PI |
| Math.SQRT1_2 | 0.7071067811865475727 | Square root of 0.5 |
| Math.SQRT2 | 1.414213562373095145 | Square root of 2 |

```

<script>
    document.write(Math.PI + "<br>");
    document.write(Math.SQRT2 + "<br>");
    document.write(Math.SQRT1_2 + "<br>");
    document.write(Math.E + "<br>");
</script>
```

3.141592653589793
 1.4142135623730951
 0.7071067811865476
 2.718281828459045

| Method syntax | Returns | Math Objects (Methods) |
|------------------------|-------------------------------------------------------------------------------------|------------------------|
| Math.abs(val) | Absolute value of <i>val</i> | |
| Math.acos(val) | Arc cosine (in radians) of <i>val</i> | |
| Math.asin(val) | Arc sine (in radians) of <i>val</i> | |
| Math.atan(val) | Arc tangent (in radians) of <i>val</i> | |
| Math.atan2(val1, val2) | Angle of polar coordinates <i>x</i> and <i>y</i> | |
| Math.ceil(val) | Returns the value of <i>x</i> rounded up to its nearest integer | |
| Math.cos(val) | Cosine of <i>val</i> | |
| Math.exp(val) | Euler's constant to the power of <i>val</i> | |
| Math.floor(val) | Next integer less than or equal to <i>val</i> | |
| Math.max(val1, val2) | The greater of <i>val1</i> or <i>val2</i> | |
| Math.min(val1, val2) | The lesser of <i>val1</i> or <i>val2</i> | |
| Math.pow(val1, val2) | <i>Val1</i> to the <i>val2</i> power | |
| Math.random() | Random number between 0 and 1 | |
| Math.round(val) | returns the nearest integer: <i>N</i> +1 when <i>val</i> >= n.5; otherwise <i>N</i> | |
| Math.sin(val) | Sine (in radians) of <i>val</i> | |
| Math.sqrt(val) | Square root of <i>val</i> | |
| Math.tan(val) | Tangent (in radians) of <i>val</i> | |
| Math.log(val) | Natural logarithm (base e) of <i>val</i> | |

Date

- Date object allows the handling of date and time information.
 - All dates are in milliseconds from January 1, 1970, 00:00:00.
 - Dates before 1970 are invalid dates.
- There are different ways to define a new instance of the date object:

```
var d = new Date()      //Current date  
var d = new Date(milliseconds)  
var d = new Date(dateString)  
var d = new Date(year, month, day, hours, minutes, seconds, milliseconds)
```

```
<script>  
  var d=new Date();  
  document.write(d);  
</script>
```

Tue Sep 24 2013 12:48:39 GMT+0530 (India Standard Time)

```
var d = new Date(86400000);  
var d = new Date(99,5,24,11,33,30,0);
```

Date Object - Methods

| | |
|----------------------------|------------------------------------------------|
| • getDate() | Date of the month (1 - 31) |
| • getDay() | Day of the week (0 - 6, 0-Sunday) |
| • getMonth() | The month (0 - 11, 0 - Jan.) |
| • getFullYear() | The year (4 digits) |
| • getHours() | Hour of the day (0 - 23) |
| • getMinutes() | Minutes (0 - 59) |
| • getSeconds() | Seconds (0 - 59) |
| • getTime() | Milliseconds since 1/1/1970 |
| • getTimezoneOffset() | Offset between local time and GMT |
| • setDate(dayValue) | 1-31 |
| • setHours(hoursValue) | 0-23 |
| • setMinutes(minutesValue) | 0-59 |
| • setMonth(monthValue) | 0-11 |
| • setSeconds(secondsValue) | 0-59 |
| • setTime(timeValue) | >=0 |
| • setYear(yearValue) | >=1970 |
| • valueOf() | returns number of millisecond since 1 jan 1970 |
| • Date.now() | same as valueOf() |

Array

- An array is data structure for storing and manipulating ordered collections of data.
- An array can be created in several ways.
 - Eg1: Regular

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

- Eg 2: Condensed: var cars=new Array("Spark","Volvo","BMW");
 - Eg 3: Literal: var cars=["Spark","Volvo","BMW"];
 - Eg 4: var matrix = [[1,2,3], [4,5,6], [7,8,9]];
 - Eg 5 : var sparseArray = [1,,,5];
- Iterating thru array

```
var arr = [44,55,34,21,89];
for(var i=0; i < arr.length; i++)
  console.log(arr[i]);

for(var i in arr)
  console.log(i,arr[i]);

for(var i of arr)
  console.log(i)
```

| |
|------|
| 44 |
| 55 |
| 34 |
| 21 |
| 89 |
| 0 44 |
| 1 55 |
| 2 34 |
| 3 21 |
| 4 89 |
| 44 |
| 55 |
| 34 |
| 21 |
| 89 |

Array Object Methods

- arrayObject.reverse()
- arrayObject.slice(startIndex, [endIndex])
- arrayObject.join(separatorString) : array contents will be joined and placed into arrayText by using the comma separator“
- arrayObject.push(): add one or more values to the end of an array

```
arrayObject.slice(startIndex [, endIndex])      //Returns: Array
var solarSys = new Array ("Mercury","Venus","Earth","Mars","Jupiter","Saturn")
var nearby = solarSys.slice(1,4)
// result: new array of "Venus", "Earth", "Mars"
```

```
arrayObject.concat(array2)
var a1 = new Array(1,2,3)
var a2 = new Array("a","b","c")
var a3 = a1.concat(a2)
// result: array with values 1,2,3,"a","b","c"
```

| |
|---------------------------|
| Andrew Monica Catie Jenna |
| joinedarr type : string |

```
var names = ["Andrew","Monica","Catie","Jenna"];
var joined_arr = names.join("|")
console.log(joined_arr)
console.log("joinedarr type :",typeof(joined_arr))
```

```
a = [] // Start with an empty array
a.push("zero") // Add a value at the end. a = ["zero"]
a.push("one", "two") // Add two more values. a = ["zero", "one", "two"]
```

Number

- The **Number** object is a wrapper object allowing you to work with numerical values.
 - A Number object is created using the `Number()` constructor or as a literal
 - JavaScript creates Number objects when a variable is set to a number value, for example `var num = 255.336`; **It is seldom necessary to create Number objects explicitly.**

| Property | Description |
|---------------------------------------------|-----------------------------------------------------------------|
| <code>constructor</code> | Returns the function that created JavaScript's Number prototype |
| <code>MAX_VALUE</code> | Returns the largest number possible in JavaScript |
| <code>MIN_VALUE</code> | Returns the smallest number possible in JavaScript |
| <code>NEGATIVE_INFINITY</code> | Represents negative infinity (returned on overflow) |
| <code>NaN</code> | Represents a "Not-a-Number" value |
| <code>POSITIVE_INFINITY</code> | Represents infinity (returned on overflow) |
| <code>prototype</code> | Allows you to add properties and methods to an object |
| <u><code>Number.MAX_SAFE_INTEGER</code></u> | The maximum safe integer in JavaScript ($2^{53} - 1$). |

Number Methods and properties

```
console.log(Number.MIN_VALUE)
console.log(Number.MAX_VALUE)
console.log(Number.NEGATIVE_INFINITY)
console.log(Number.POSITIVE_INFINITY)
console.log(Number.prototype)
console.log(Number.MIN_SAFE_INTEGER)
console.log(Number.MAX_SAFE_INTEGER)
```

5e-324
1.7976931348623157e+308
-Infinity
Infinity
► Number {*0*, constructor: *f*,
-9007199254740991
9007199254740991

| Method | Description |
|-------------------------|-------------------------------------------------------------------|
| isFinite() | Checks whether a value is a finite number |
| isInteger() | Checks whether a value is an integer (new in ES6) |
| isNaN() | Checks whether a value is Number |
| isSafeInteger() | Checks whether value can be safely represented in JavaScript. |
| toExponential(x) | Converts a number into an exponential notation |
| toFixed(x) | Formats a number with x numbers of digits after the decimal point |
| toPrecision(x) | Formats a number to x length |
| valueOf() | Returns the primitive value of a number |
| toString() | Converts a number to a string |
| parseFloat()/parseInt() | Same as global counterparts |

```
let num = new Number('18.907');
console.log(num.toExponential()); //Converts a number into an Exponential notation.
console.log(num.toFixed());
console.log(num.toFixed(2)); //Rounds up a number to x digits after the decimal.
console.log(num.toPrecision());
console.log(num.toPrecision(3)); //Rounds up a number to a length of x digits.
console.log(typeof num.toString()); //toString() Returns a String value of a number object.
console.log(num.valueOf()); //Returns the primitive value of the Number object
```

| |
|-----------|
| 1.8907e+1 |
| 19 |
| 18.91 |
| 18.907 |
| 18.9 |
| string |
| 18.907 |

```
let numObj = 5.123456
```

```
console.log(numObj.toPrecision()) // logs '5.123456'
console.log(numObj.toPrecision(5)) // logs '5.1235'
console.log(numObj.toPrecision(2)) // logs '5.1'
console.log(numObj.toPrecision(1)) // logs '5'
```

```
let numObj = 53.123456
```

```
console.log(numObj.toPrecision()) // logs '5.123456'
console.log(numObj.toPrecision(5)) // logs '53.123'
console.log(numObj.toPrecision(2)) // logs '53'
console.log(numObj.toPrecision(1)) // logs '5e+1'
```

Boolean Object

- The Boolean object represents two values, either "true" or "false".
- The following syntax creates a **boolean object**. `var val = new Boolean(value);`
 - If value parameter is omitted or is 0, -0, null, false, NaN, undefined, or the empty string (""), the object has an initial value of false.
 - Everything else is true; including any object, an empty array ([]), or the string "false", create an object with an initial value of true.
 - Ie, Everything With a "Value" is True and Everything Without a "Value" is False
- Boolean Methods:
 - `toString()` : Returns a string of either "true" or "false" depending upon the value of the object.
 - `valueOf()` : Returns the primitive value of the Boolean object.

| |
|----------------------------|
| ▶ Boolean { <i>false</i> } |
| ▶ Boolean { <i>true</i> } |
| ▶ Boolean { <i>true</i> } |

```
<script language="javascript">
| var bool1 = new Boolean("");
| console.log(bool1);

var bool2 = new Boolean(0);
console.log(bool2);

var bool3 = new Boolean(undefined);
console.log(bool3);

var bool4 = new Boolean(null);
console.log(bool4);

var bool5 = new Boolean(NaN);
console.log(bool5);

var bool6 = new Boolean("some text");
console.log(bool6);

var bool7 = new Boolean(1);
console.log(bool7);

</script>
```

User defined Objects

- **Objects can be created in several ways:**

1. Using the Object() constructor:
2. Using Object.create() method:
3. Using the bracket's syntactic sugar. eg : var b = {};
4. Using a function constructor
5. Using class (ES6)

- **Using Object Initializers**

- Syntax : objName = {property1:value1, property2:value2, ... }
- Eg

```
var empty = {} ; // An object with no properties
var point = { x:0, y:0 } ;
var person = { "name ":"amit", "age":23} ;
var emp = {"name":"Amit",
           "age": 37.5,
           "married": true,
           "address":{ "city":"Pune" , "state":"Mah" },
           "hobbies":["swimming","reading","music"]
}
```

Examples : Using Object Initializers

// Example 1

```
var myFirstObject = {};
myFirstObject.firstName = "Andrew";
myFirstObject.lastName = "Grant";
console.log(myFirstObject.firstName);
```

// Example 3

```
var myThirdObject = new Object();
myThirdObject.firstName = "Andrew";
myThirdObject.lastName = "Grant";
console.log(myThirdObject.firstName);
```

```
var myFirstObject = {};
myFirstObject.firstName = "Andrew";
console.log(myFirstObject.firstName);
myFirstObject.firstName = "Monica";
console.log(myFirstObject.firstName);
myFirstObject["firstName"] = "Catie";
console.log(myFirstObject["firstName"]);
```

// Example 2

```
var mySecondObject = {
  firstName: "Andrew",
  lastName: "Grant"
};
console.log(mySecondObject.firstName);
```

//Adding Methods to Objects

```
var person= {
  name: "Andrew",
  age: 21,
  info: function () {
    console.log("Name" + this.name );
    console.log("Age" + this.age );
  }
};
person.info();
for (var prop in person) {
  console.log(person[prop]);
}
```

Creating New Objects

- Using function Constructors

```
function person(name, age) {  
    this.name = name  
    this.age = age  
}  
p1 = new person( "Ken" , 33 )  
console.log(p1.age, p1.name)
```

```
function car(make, year, owner) {  
    this.make = make  
    this.year = year  
    this.owner = owner  
}  
c1 = new car( "Tiago", 2022, p1 )  
console.log(c1.make, c1.year, c1.owner.name)
```

- Accessing properties

```
var name = p1.name  
c1.make = "Baleno"  
var age = c1.owner.age  
  
console.log(name, age, c1.make) //Ken 33 Baleno
```

Creating New Objects

- You **cannot** add a new property to an existing object constructor
 - Eg : Car.price = null → gives error
- The **prototype** property allows you to add new properties to object constructors:

```
car.prototype.model = "Tiago"
car.prototype.price = 730000
console.log(c1.make, c1.model, c1.price)
//Tata Tiago 730000
```

Only modify your own prototypes. Never modify the prototypes of standard JavaScript objects.

```
c2 = new car("Maruti", 2020, p1)
c2.model = "Suzuki"
c2.price = 500000
console.log(c2.make, c2.model, c2.price)
//Maruti Suzuki 500000
```

- Defining methods

```
function Person(name, age){
    this.name = name
    this.age = age
    this.display = function(){
        console.log(this.name, this.age)
    }
}
p1 = new Person("Soha", 30)
p1.display() //Soha 30
```

Newer Javascript – Javascript classes

```
<script>
class Person {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }
  getName() {
    return this.firstname + ' ' + this.lastname;
  }
}

var me = new Person('Shrilata', 'T');
console.log(me.getName());
</script>
```

constructor method is always defined with the name "constructor"

```
class Emp extends Person {
  getJob() {
    return 'Programmer';
  }
}

var me = new Emp('Anil', 'Patil');
console.log(me.getName());
console.log(me.getJob());
```

Classes can have methods, which defined as functions, albeit without needing to use the function keyword.

Inheritance

```
class Car {  
    constructor(brand) {  
        this.carname = brand;  
    }  
    present() {  
        return 'I have a ' + this.carname;  
    }  
}  
  
class Model extends Car {  
    constructor(brand, mod) {  
        super(brand);  
        this.model = mod;  
    }  
    show() {  
        return this.present() + ', it is a ' + this.model;  
    }  
}  
  
let myCar = new Model("Ford", "Mustang");  
console.log(myCar.show()); //I have a Ford, it is a Mustang
```

Inheritance ensures code reuse.

Polymorphism

```
class Animal {  
    speak() {  
        console.log("Animal makes a sound");  
    }  
}  
  
class Dog extends Animal {  
    speak() {  
        console.log("Dog barks");  
    }  
}  
  
class Cat extends Animal {  
    speak() {  
        console.log("Cat meows");  
    }  
}
```

Polymorphism allows objects to behave differently using the same interface

Animal makes a sound
Dog barks
Cat meows

```
const animal = new Animal()  
animal.speak()  
  
const dog = new Dog()  
dog.speak()  
  
const cat = new Cat()  
cat.speak();
```

Encapsulation

- Encapsulation wraps up data and information under a single unit.
 - In OOP, it means binding together the data and the functions that manipulate them together in a class

```
class Person{
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    getDetails() {
        return `${this.name} is ${this.age} years old.`;
    }
}
const p1 = new Person("Soha", 30);
console.log(p1.getDetails()); //Soha is 30 years old.
```

Different ways to achieve encapsulation in JavaScript?

- JavaScript is a loosely typed language and does not have traditional access modifiers like private, protected, and public which are common in other object-oriented programming languages.
- However, we can achieve encapsulation in JavaScript through the following methods:
 - Closures
 - Constructor functions
 - Class syntax

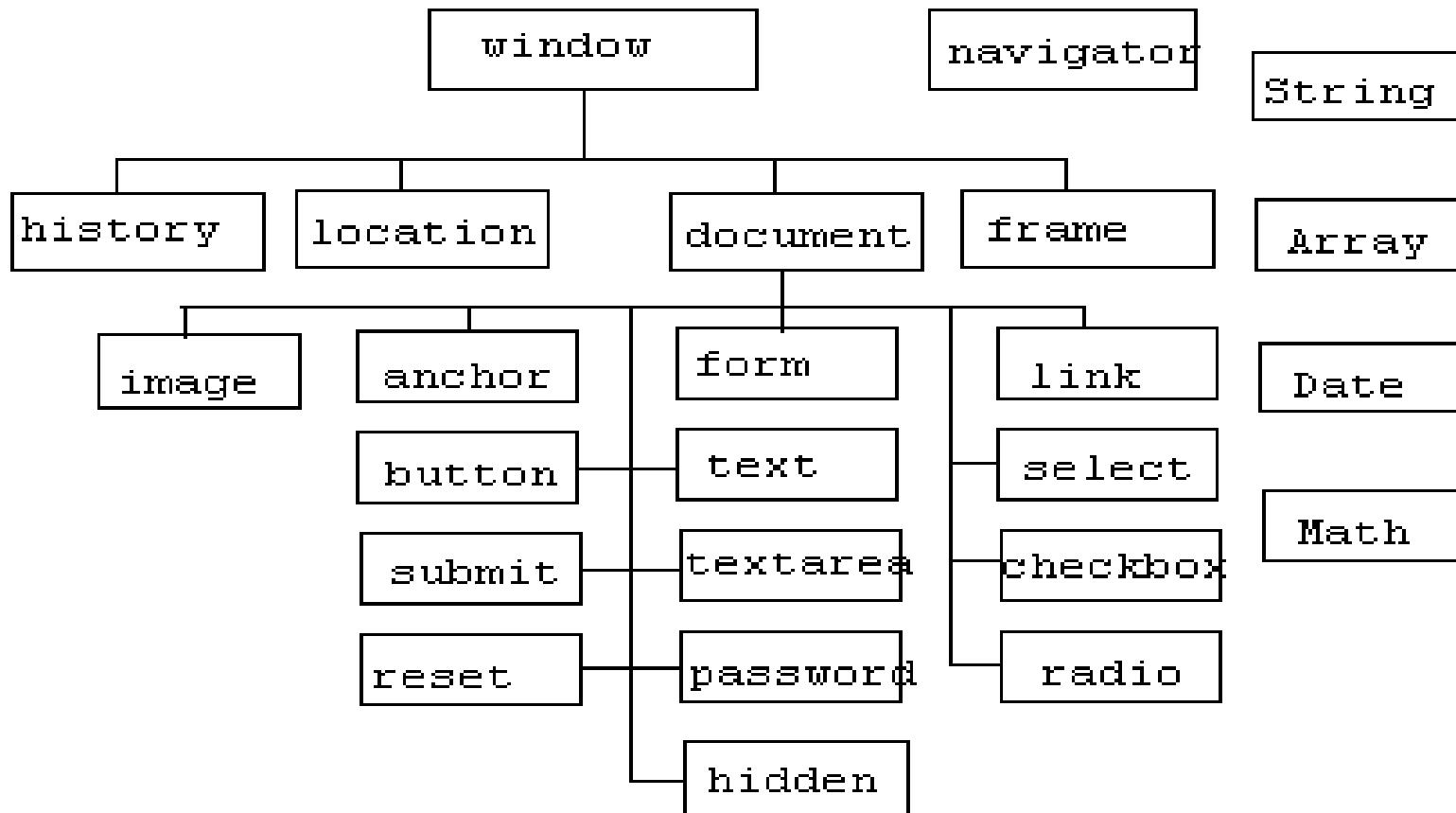
```
function Person(name) {  
    let _name = name;  
    this.getName = function () {  
        return _name;  
    };  
    this.setName = function (newName) {  
        _name = newName;  
    };  
}  
const person = new Person('John Doe');  
console.log(person.getName()); // outputs 'John Doe'  
person.setName('Jane Doe');  
console.log(person.getName()); // outputs 'Jane Doe'  
console.log(person._name) //undefined
```

```
class Student{  
  
    #sid;  
    #sname;  
  
    constructor(sid, sname){  
        this.#sid = sid;  
        this.#sname = sname;  
    }  
  
    displayStudDetails(){  
        console.log(this.#sid, this.#sname)  
    }  
  
}  
  
var s1 = new Student(1001, "Kartik")  
var s2 = new Student(1002, "Neha")  
var s3 = new Student(1003, "Ravi")  
  
s1.displayStudDetails()  
s2.displayStudDetails()  
s3.displayStudDetails()  
  
s1.sid = 999  
s1.sname = "aaa"  
s1.displayStudDetails()
```

The class syntax supports the creation of private properties and methods with the `#` symbol, although this is a proposed feature that may not be widely supported yet.

```
1001 'Kartik'  
1002 'Neha'  
1003 'Ravi'  
1001 'Kartik'
```

JavaScript Document Object Model



Navigator Object Hierarchy

Window Object Methods

- `alert(message)`
 - `window.alert("Display Message")`
- `confirm(message)`
 - `window.confirm("Exit Application ?")`
- `prompt(message,[defaultReply])`
 - `var input=window.prompt("Enter value of X")`

setInterval and setTimeout methods

```
<body>
<input type="text" id="clock" size="35" />
<script language=javascript>
var int=self.setInterval("clock()",50)
function clock() {
    var ctime=new Date()
    document.getElementById("clock").value=ctime
}
</script>
<button onclick="int=window.clearInterval(int)">Stop interval</button>
</body>
```

```
<head> <script type="text/javascript">
function timedMsg()  {
    var t=setTimeout("alert('5 seconds!')",5000)
}
</script> </head>
<body> <p>Click on the button. An alert box will be displayed after 5 seconds.</p>
<form>
<input type="button" value="Display timed alertbox!" onClick="timedMsg()">
</form>
</body>
```

Document Object

- When an HTML document is loaded into a web browser, it becomes a document object; root node of the HTML document and owns all other nodes

| | |
|--------------------------------------------------------|------------------------------------------------------------------------|
| <u>document.anchors</u> | Returns a collection of all the anchors in the document |
| <u>document.baseURI</u> | Returns the absolute base URI of a document |
| <u>document.cookie</u> | Returns all name/value pairs of cookies in the document |
| <u>document.forms</u> | Returns a collection of all the forms in the document |
| <u>document.getElementById()</u> | Returns the element that has the ID attribute with the specified value |
| <u>document.getElementsByName()</u> | Accesses all elements with a specified name |
| <u>document.getElementsByTagName()</u> | Returns a NodeList containing all elements with the specified tagname |
| <u>document.images</u> | Returns a collection of all the images in the document |
| <u>document.lastModified</u> | Returns the date and time the document was last modified |
| <u>document.links</u> | Returns a collection of all the links in the document |
| <u>document.referrer</u> | Returns the URL of document that loaded current document |
| <u>document.title</u> | Sets or returns the title of the document |
| <u>document.URL</u> | Returns the full URL of the document |
| <u>document.write()</u> | Writes HTML expressions or JavaScript code to a document |
| <u>document.writeln()</u> | Same as write(), but adds a newline character after each statement |

Examples:

```
<html>
<body>
<p id="intro">Hello World!</p>
<p>This example demonstrates the <b>getElementById</b> method!</p>
<script>
  x=document.getElementById("intro");
document.write("<p>The text from the intro paragraph: " + x.innerHTML + "</p>");
</script>
</body>
</html>
```

Hello World!

This example demonstrates the **getElementById** method!

The text from the intro paragraph: Hello World!

```
<body>
<p id="p1">Hello World!</p>
<p id="p2">Hello World!</p>

<script>
document.getElementById("p2").style.color = "blue";
document.getElementById("p2").style.fontFamily = "Arial";
document.getElementById("p2").style.fontSize = "larger";
</script>
```

Hello World!

Hello World!

The paragraph above was changed by a script.

```
<p>The paragraph above was changed by a script.</p>
</body>
```

Examples

```
<script>
    function f1(){
        var str = "";
        var plist = document.getElementsByName("c1");
        for(i=0;i<plist.length;i++){
            if(plist[i].checked)
                str = plist[i].value + " &nbsp;" + str ;
        }
        document.getElementById("s1").innerHTML = "Skills : " + str;
    }
</script>
</head>
<body>
    <H1> Welcome to Javascript </H1>
    Skills:
    <input type="checkbox" name="c1" value="java">Java
    <input type="checkbox" name="c1" value="JS">JavaScript
    <input type="checkbox" name="c1" value="JSP">JSP

    <input type="button" value="click" onclick="f1()"><br>
    <span id="s1"></span>
</body>
```

Welcome to Javascript

Skills: Java JavaScript JSP
Skills : JSP java

```

<html>
  <head>
    <SCRIPT>
      function changeColor(){
        var para = document.getElementById("p1");
        para.style.color="blue";
        para.style.backgroundColor = "lightgray";
        para.style.font = "italic bold 30px arial,serif";
      }
      function revertColor(){
        var para = document.getElementById("p1");
        para.style.color="black";
        para.style.backgroundColor = "white";
        para.style.font = "12px arial,serif";
      }
    </SCRIPT>
  </head>
  <body>
    <div>
      <p id="p1" onmouseover="changeColor()"
          onmouseout="revertColor()"
          onclick="f1()" >
        Hover with mouse to see color change
      </p>
    </div>
  </body></html>

```

Mouse events



**Hover with mouse
to see color
change**

DOM manipulation using javascript

```
<!DOCTYPE html>
<html>
<body>
<p>This is para-1</p>
<p>This is para-2</p>
<p>This is para-3</p>
<button onclick="myFunction()">Change!!</button>
<script>
function myFunction() {
    var nodelist = document.getElementsByTagName("p");
    var i;
    for (i = 0; i < nodelist.length; i++) {
        nodelist[i].style.backgroundColor = "yellow";
    }
}
</script>
</body>
</html>
```

This is para-1

This is para-2

This is para-3

Change!!

This is para-1

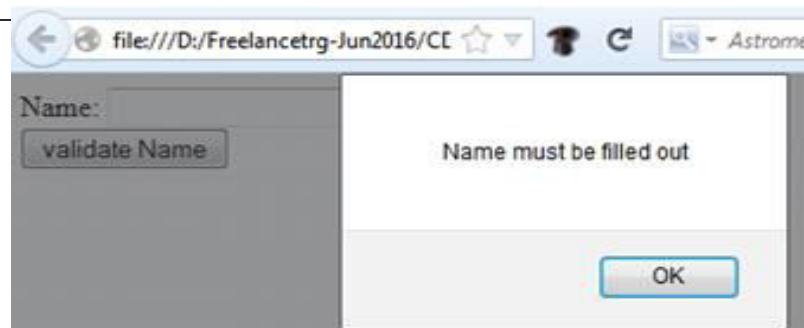
This is para-2

This is para-3

Change!!

Form validation

```
<html>
<head>
<script>
function validate(){
    var x=document.getElementById("fname").value;
    if (x == null || x == "") {
        alert("Name must be filled out");
        return false;
    }
}
</script>
</head>
<body>
<form id="form1" onsubmit="return validate()">
Name: <input type="text" name="fname" id="fname" />
<input type="submit" value="validate Name" />
</form>
</body></html>
```



Input a number between 1 and 10:

Submit

Input not valid

Input a number between 1 and 10:

Submit

Input OK

```
<body>
<input type="text" name="numb" />
<button type="button" onclick="myFunction()>Submit</button>
<p id="demo"></p>
<script>
function myFunction() {
    var x, text;
    x = document.getElementById("numb").value;
    if (isNaN(x) || x < 1 || x > 10) {
        text = "Input not valid";
    } else {
        text = "Input OK";
    }
    document.getElementById("demo").innerHTML = text;
}
</script>
</body>
```

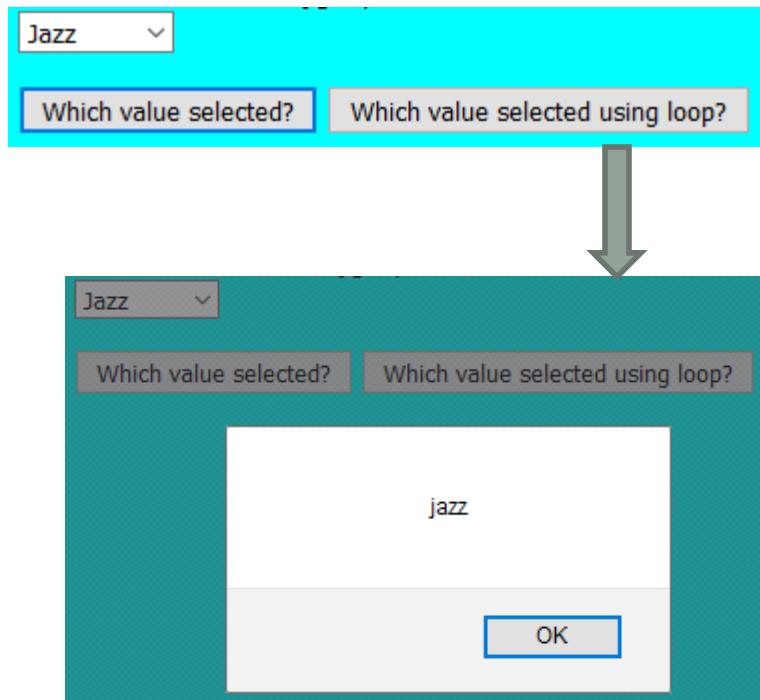
```

<SCRIPT>
function valSelected1(){
    var sel = document.getElementById("musicTypes");
    alert(sel.value);      // prints value, not text
    var opt = sel.options[sel.selectedIndex];
    alert(opt.text);      //option.text prints text
}

function valSelected3(){
    var sel = document.getElementById("musicTypes");  var opt;
    for ( var i = 0, len =; i < sel.options.length; i++ ) {
        opt = sel.options[i];
        if ( opt.selected == true ) { break; }
    }
    alert(opt.value);
}
</SCRIPT>
<FORM NAME="selectForm">
    <SELECT name="musicTypes" id="musicTypes">
        <OPTION VALUE="rnb" SELECTED> R&B </OPTION>
        <OPTION VALUE="jazz"> Jazz </OPTION>
        <OPTION VALUE="blues"> Blues </OPTION>
    </SELECT>
    <INPUT TYPE="button" VALUE="value selected?" onClick="valSelected1()">
    <INPUT TYPE="button" VALUE="value selected using loop?" onClick="valSelected3()">
</FORM>
</BODY>

```

Example



Example

```
<SCRIPT>
function valSelected(){
    var radio = document.getElementsByName("coffee");
    for(var i = 0; i < radio.length; i++){
        if(radio[i].checked) console.log("coffee selected : " + radio[i].value);
    }
    var checklist = document.getElementsByClassName("c1");
    for(i=0; i<checklist.length; i++){
        if (checklist[i].checked == true) console.log("Music selected : " + checklist[i].value);
    }
}
</SCRIPT>

<FORM NAME="selectForm">
<B>Which Music types do you like?</B>
<input type="checkbox" class="c1" id="c1" value="blues">Blues</input>
<input type="checkbox" class="c1" id="c2" value="classical">Classical</input>
<input type="checkbox" class="c1" id="c3" value="opera">Opera</input>

<b>Choose Coffee to go with your music!</b><br>
<INPUT TYPE="radio" name="coffee" id="coffee" VALUE="cappuchino">Cappuchino</input>
<INPUT TYPE="radio" name="coffee" id="coffee" VALUE="latte">Latte</input>
<INPUT TYPE="radio" name="coffee" id="coffee" VALUE="Mocha">Mocha</input>
<INPUT TYPE="button" VALUE="Which option selected?" onClick="valSelected()">
</FORM>
```

Which Music types do you like?

Blues Classical Opera

Choose Coffee to go with your music!

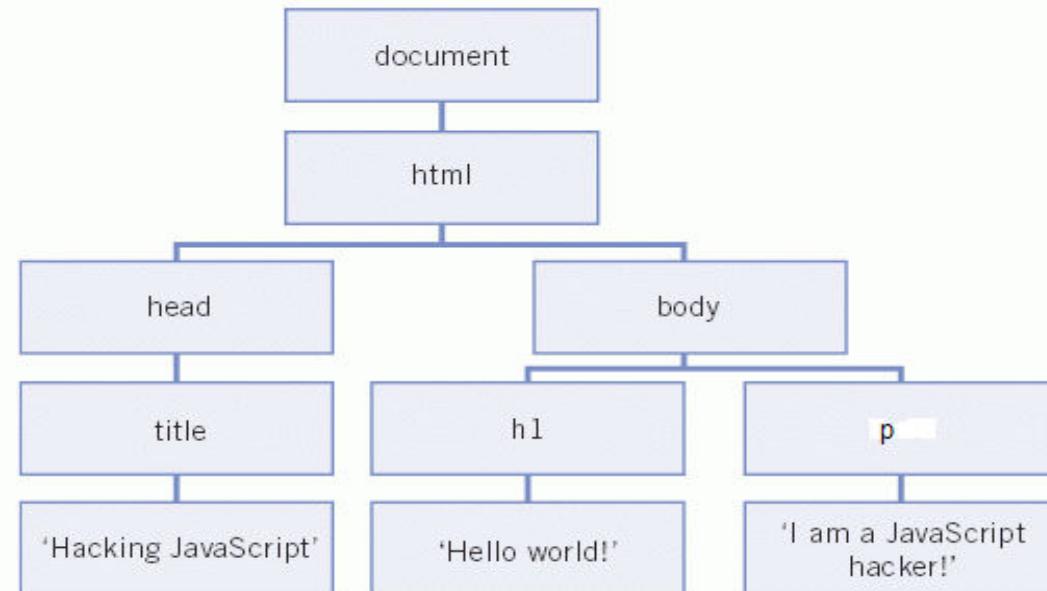
Cappuchino Latte Mocha

coffee selected : cappuchino
Music selected : blues
Music selected : classical

Documents As Trees of Nodes

- The Document object, its Element objects, and Text objects are all Node objects. Node defines the following important properties:
 - `parentNode` : node that is the parent of this one, or null for nodes like the Document object that have no parent.
 - `childNodes` : read-only NodeList that is a representation of a Node's child nodes.
 - `firstChild, lastChild`: first & last child nodes of a node; null if node has no children.
 - `nextSibling, previousSibling` : The next and previous sibling node of a node.
 - `nodeValue` : textual content of a Text or Comment node.
 - `nodeName` : tag name of an Element, converted to uppercase.
 - `nodeType` : kind of node this is.

| Node type | value |
|-----------|-------|
| Document | 9 |
| Element | 1 |
| Text | 3 |
| Comments | 8 |



DOM manipulation using javascript

- The **querySelector()** method returns the first element that matches one or more CSS selectors. If no match is found, it returns null.
 - Its equivalent to getElementById()
 - However, querySelector() and querySelectorAll() are newer methods; with these we are free to target elements based on any CSS selector, thus we have more flexibility.
 - Syntax : `var ele = document.querySelector(selector);`
 - ele – First matching element or null (if no element matches the selectors)
 - selector – one or more CSS selectors, such as "#foold", ".fooClass" etc
 - Eg : `let e = document.querySelector("p"); //selects the first paragraph in html doc`
- **querySelectorAll()** returns a NodeList representing a list of the document's elements that match the specified group of selectors.
 - Syntax : `elementList = parentNode.querySelectorAll(selectors);`
 - Eg : To obtain a NodeList of all of the <p> elements in the document:
`const matches = document.querySelectorAll("p");`
 - Eg To return a list of all <div> elements within the document with a class of either note or alert:
`const matches = document.querySelectorAll("div.note, div.alert");`

DOM manipulation using javascript : querySelector()

```
<p>paragraph one</p>
<div>div one</div>
<p>paragraph two</p>
<div>div two</div>
<input type="button" value="style"
       onclick="f1()">
<script>
    function f1() {
        var firstDiv = document.querySelector('div');
        firstDiv.style.color = 'red';
        firstDiv.style.backgroundColor = 'gold';
    }
</script>
```

The screenshot shows the browser's developer tools with the 'Elements' tab selected. The DOM tree on the right side of the tools shows the following structure:

- <html>
- > <head>...</head>
- > <body> == \$0
 - <p>paragraph one</p>
 - <div>div one</div>
 - <p>paragraph two</p>
 - <div>div two</div>

The element <div>div one</div> is highlighted with a yellow background. In the main content area, the text "div one" is displayed in red color and gold background.

The screenshot shows the browser's developer tools with the 'Elements' tab selected. The DOM tree on the right side of the tools shows the following structure:

- <html>
- > <head>...</head>
- > <body> == \$0
 - <p>paragraph one</p>
 - <div style="color: red; background-color: gold;">div one</div>
 - <p>paragraph two</p>
 - <div>div two</div>

The element <div style="color: red; background-color: gold;">div one</div> is highlighted with a yellow background. In the main content area, the text "div one" is displayed in red color and gold background.

DOM manipulation using javascript : querySelectorAll()

```
<h2>querySelectorAll() Method</h2>
<p>This is paragraph 1.</p>
<p>This is paragraph 2.</p>
<button onclick="f1()">Change</button>
<script>
    function f1() {
        let plist = document.querySelectorAll("p");
        let i;
        for (i = 0; i < plist.length; i++) {
            plist[i].style.backgroundColor = "lightblue";
            plist[i].style.color = "white";
        }
    }
</script>
```

querySelectorAll() Method

This is paragraph 1.

This is paragraph 2.

Change

querySelectorAll() Method

This is paragraph 1.

This is paragraph 2.

Change

Creating, Inserting, and Deleting Nodes

- **createElement()** : create new Element nodes
- **createTextNode()** : Creates text nodes
- **cloneNode()** : returns a new copy of the node
- **appendChild() or insertBefore()** : insert new node into the document
- **removeChild()**: removes a node from the document tree
 - this method isn't invoked on the node to be removed but on parent of that node
 - Eg : To remove the node n from the document : `n.parentNode.removeChild(n);`

```
<h2>Load external js into &lt;head&gt;</h2>
<button onclick="loadasync('myscript.js')>Change</button>
<script>
  function loadasync(url) {
    var head = document.getElementsByTagName("head")[0]; // Find <head>
    var s = document.createElement("script"); // Create a <script> element
    s.src = url; // Set its src attribute
    head.appendChild(s); // Insert the <script> into head
  }
}
```

Load external js into <head>

Change

```
<html lang="en">
  <head></head>
```

```
<head> == $0 ⌂
  <script src="myscript.js"></script>
</head>
```

DOM manipulation using javascript : insertBefore()

```
<div id="div1">
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>

<script>
  var newpara = document.createElement("p");
  var node = document.createTextNode("This is new.");
  newpara.appendChild(node);

  var parent = document.getElementById("div1");
  var oldpara = document.getElementById("p1");
  parent.insertBefore(newpara, oldpara);
</script>
```

This is new.

This is a paragraph.

This is another paragraph.

The screenshot shows the browser's developer tools with the 'Elements' tab selected. The DOM tree is displayed, starting with the <html> element, followed by <head> and <body>. Inside the <body>, there is a <div id="div1">. This div contains three <p> elements: one with the text "This is new.", one with "This is a paragraph.", and one with "This is another paragraph.". The <p> element containing "This is new." is highlighted with a red border, matching its appearance in the browser preview. The browser preview itself is visible on the left side of the tools, showing the same three paragraphs in sequence.

DOM manipulation using javascript : removeChild()

```
<body>
The title of the document is:
<div id="div1">
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>

<script>
var parent = document.getElementById("div1");
var child = document.getElementById("p1");
parent.removeChild(child);
</script>
```

this method isn't invoked on the node to be removed but on parent of that node

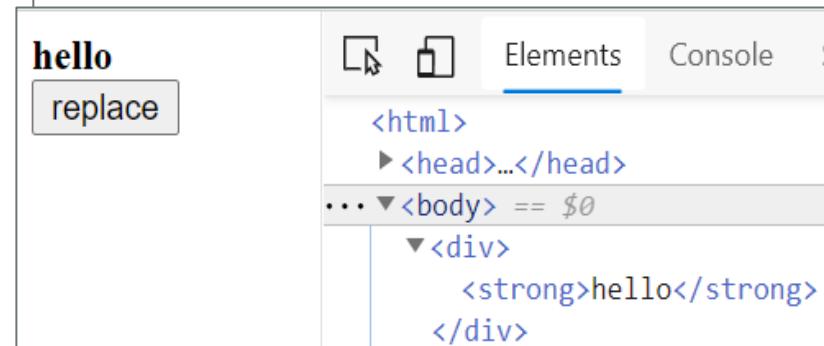
The title of the document is:

This is another paragraph.

DOM manipulation using javascript

- The **replaceChild()** method replaces a child element with another one belonging to the parent element that calls this method.
- In this example the child element `` belonging to the `<div>` parent element is replaced with a newly created `` tag.

```
<div>
  <strong>hello</strong>
</div>
<input type="button" value="replace"
       onclick="fReplace()">
<script>
function fReplace(){
  var em = document.createElement('em');
  var strong = document.querySelector('strong');
  var div = document.querySelector('div');
  em.textContent = 'hi';
  div.replaceChild(em, strong);
}
</script>
```



Regular Expressions

- A regular expression is an object that describes a pattern of characters.
 - Its matched against a text string, when you perform searches & replacements
 - Perform client-side data validations or any other extensive text entry parsing
 - RegExp objects may be created either with the **RegExp() constructor** or using a **special literal** syntax.
 - regular expression literals are specified as characters within a pair of slash (/)

var re = / / → simple pattern to match the space character

var re = / /g → matching a string on a global basis

var re = /web/i → a case-insensitive match

var re = /web/gi → expression is both case-insensitive and global

- Eg

```
str = "I love JavaScript!";
regexp = /love/;
alert( str.search(regexp) ); // 2
```

| | |
|---|--------------------------------------------------------------------------------------|
| i | Perform case-insensitive matching |
| g | Perform a global match (find all matches rather than stopping after the first match) |
| m | Perform multiline matching |

RegEx – Special Characters (Contd.)

- \d Numeral: Find any single digit 0 through 9
 - /\d\d\d/ matches “212” and “415” but not “B17”
- \D Non-numeral: Find any non-digit
 - /\D\D\D/ matches “ABC” but not “212” or “B17”
- \s Single White Space: Find any single space character
 - /over\sbite/ matches “over bite” but not “overbite” or “over bite”
- \S Single Non-White Space:
 - /over\Sbite/ matches “over-bite” but not “overbite” or “over bite”
- \w Letter, Numeral, or Underscore:
 - /A\w/ matches “A1” and “AA” but not “A+”
- \W Not letter, Numeral, or Underscore:
 - /A\W/ matches “A+” but not “A1” and “AA”

RegEx – Special Characters (Contd.)

- “.” Any Character Except Newline:
 - /.../ matches “ABC”, “1+3”, “A 3” or any 3 characters
- [...] Character Set: any character in the specified character set
 - /[AN]BC/ matches “ABC” and “NBC”
- [^...] Negated Character Set: any character not in the specified character set
 - /[^AN]BC/ matches “BBC” and “CBC” but not “ABC” or “NBC”

• Positional Metacharacters

- “^” - At the beginning of a string or line
 - /^Fred/ matches “Fred is OK” but not “I’m with Fred” or “Is Fred here?”
- “\$” - At the end of a string or line
 - /Fred\$/ matches “I’m with Fred” but not “Fred is OK” or “Is Fred here?”

RegEx – Counting Metacharacters

- “*” - Zero or More Times:
 - /Ja*vaScript/ matches “JvaScript”, “JavaScript” & “JaaavaScript” but not “JovaScript”
- “?” - Zero or One Time:
 - /Ja?vaScript/ matches “JvaScript” or “JavaScript” but not “JaaavaScript”
- “+” - One or More Times:
 - /Ja+vaScript/ matches “JavaScript” or “JaavaScript” but not “JvaScript”
- {n} - Exactly n Times:
 - /Ja{2}vaScript/ matches “JaavaScript” but not “JvaScript” or “JavaScript”
- {n,} - N or More Times:
 - /Ja{2,}vaScript/ matches “JaavaScript” or “JaaavaScript” but not “JavaScript”
- {n,m} - At Least n, At Most m Times:
 - /Ja{2,3}vaScript/ matches “JaavaScript” or “JaaavaScript” but not “JavaScript”

Regular Expression Object

- Create Regular Expression:

```
regExpObject = /pattern/ [g | i | gi]
```

```
regExpObject = new RegExp(pattern, flag)
```

- Eg: re = new RegExp("pushing", "g");
- Eg: var zipcode = new RegExp("\d{6}", "g");

- Methods that use regular expressions

| Method | Description |
|-----------|-------------------------------------------------------------------------------------------------------------|
| exec() | executes a search for a match in a string. It returns an array of information or null on a mismatch. |
| test() | tests for a match in a string. It returns true or false |
| match() | executes a search for a match in a string. It returns an array of information or null on a mismatch. |
| search() | tests for a match in a string; returns the index of the match, or -1 if search fails |
| replace() | executes a search for a match in a string, and replaces the matched substring with a replacement substring. |
| split() | uses a reg exp or a fixed string to break a string into an array of substrings. |

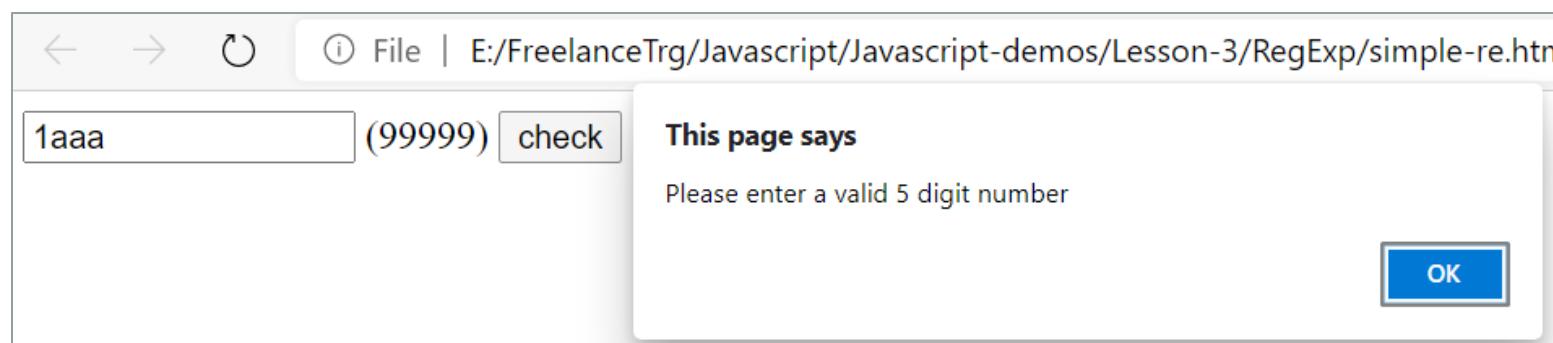
```
var pattern = /java/g;
var text = "JavaScript is more fun than java!";
var result = pattern.exec(text)
console.log(result) //["java", index: 28, input: "JavaScript is more fun than java!"]
```

When there's a "g" flag, then str.match returns an array of all matches. There are no additional properties in that array, and parentheses do not create any elements.
With no "g" flag, looks for the first match only.

```
let str = "HO-Ho-ho!";
let result = str.match( /ho/ig ); //global search
alert( result ); // HO, Ho, ho (all matches, case-insensitive)
```

```
s1 = "how are you all doing";
var re = new RegExp("o","g");
console.log(s1.replace(re,"z")); //hzw are yzu all dzing
```

```
<script >
function checkpostal(){
var re5digit=/^\d{5}$/;
//regular expression defining a 5 digit number
if (document.myform.myinput.value.search(re5digit)==-1)
  alert("Please enter a valid 5 digit number")
}
</script>
<form name="myform">
  <input type="text" name="myinput" size=15> (99999)
  <input type="button" onClick="checkpostal()" value="check">
</form>
```



Demo

```
<head>
<title>ssn Checker</title>
<script type="text/javascript">
var RE_SSN = /^[0-9]{3}[-]?[0-9]{2}[-]?[0-9]{4}$/;

function checkSsn(ssn) {
  if (RE_SSN.test(ssn)) {
    alert("VALID SSN");
  } else {
    alert("INVALID SSN");
  }
}

</script>
</head>
<body>
<form onsubmit="return false;">
  <input type="text" name="ssn" size="20"> (999-99-9999)
  <input type="button" value="Check"
    onclick="checkSsn(this.form.ssn.value);">
</form>
</body>
```

The screenshot shows a simple web application interface. On the left, the source code of the HTML file is displayed. On the right, a form is presented with a text input field containing the string '(999-99-9999)' and a button labeled 'Check'.

JSON

JSON (JavaScript Object Notation)

- JSON is a simple and easy to read and write data exchange format.
 - It is easy for humans to read and write and easy for machines to parse and generate.
 - It is based on a subset of the JavaScript, Standard ECMA-262
 - JSON is a text format that is completely language independent; can be used with most of the modern programming languages.
 - The filename extension is .json
 - JSON Internet Media type is application/json
 - It's popular and implemented in countless projects worldwide, for those don't like XML, JSON is a very good alternative solution.

- Values supported by JSON

- Strings : double-quoted Unicode, with backslash escaping
- Numbers:

 - double-precision floating-point format in JavaScript

- Booleans : true or false
- Objects: an unordered, comma-separated collection of key:value pairs enclosed in [curly braces](#), with the ':' character separating the key and the value; the keys must be strings and should be distinct from each other
- Arrays : an ordered, comma-separated sequence of values enclosed in square brackets; the values do not need to be of the same type
- Null : A value that isn't anything

```
var obj = {  
    "name": "Amit",  
    "age": 37.5,  
    "married": true,  
    "address": { "city": "Pune", "state": "Mah" },  
    "hobbies": ["swimming", "reading", "music"]  
}
```

Demo

```
<script language="javascript">
var JSONObject = { "name" : "Amit",
    "address" : "B-123 Bangalow",
    "age" : 23,
    "phone" : "011-4565763",
    "MobileNo" : 0981100092
};

var str =
"<h2><font color='blue'>Name </font>::" + JSONObject.name + "</h2>" +
"<h2><font color='blue'>Address </font>::" + JSONObject.address + "</h2>" +
"<h2><font color='blue'>Age </font>::" + JSONObject.age + "</h2>" +
"<h2><font color='blue'>Phone No </font>::" + JSONObject.phone + "</h2>" +
"<h2><font color='blue'>Mobile No </font>::" + JSONObject.MobileNo + "</h2>";

document.write(str);
</script>
```

Name ::Amit

Address ::B-123 Bungalow

Age ::23

Phone No ::011-4565763

Mobile No ::981100092

Demo

```
<script>
var students = {
    "Students": [
        { "Name": "Amit Goenka",
          "Major": "Physics"
        },
        { "Name": "Smita Pallod",
          "Major": "Chemistry"
        },
        { "Name": "Rajeev Sen",
          "Major": "Mathematics"
        }
    ]
}

var i=0
document.writeln("students.Students.length : " + students.Students.length);
for(i=1;i<students.Students.length+1;i++) {
    document.writeln("<b>Name : </b>" + students.Students[i].Name + " ");
    document.writeln("<b>Majoring in : </b>" + students.Students[i].Major);
    document.writeln("<br>");
}
</script>
```

```
students.Students.length : 3
Name : Amit Goenka Majoring in : Physics
Name : Smita Pallod Majoring in : Chemistry
Name : Rajeev Sen Majoring in : Mathematics
```

Serializing Objects

- serialization is the process of converting an object's state to a string from which it can later be restored.
 - Sometimes we receive a raw JSON string, and we need to convert it to an object. And when we want to send a JavaScript object across the network, we need to convert it to JSON (a string) before sending.
 - **JSON.parse()**: Accepts a JSON string as a parameter, and returns the corresponding JavaScript object.
 - **JSON.stringify()**: Accepts an object as a parameter, and returns the equivalent JSON string.

```
o = {x:1, y:{z:[false,null,""]}};           // Define a test object
s = JSON.stringify(o);                     // s is '{"x":1,"y":{"z":[false,null,""]}}'
p = JSON.parse(s);                        // p is a deep copy of o
console.log(p.x);                         //1
```

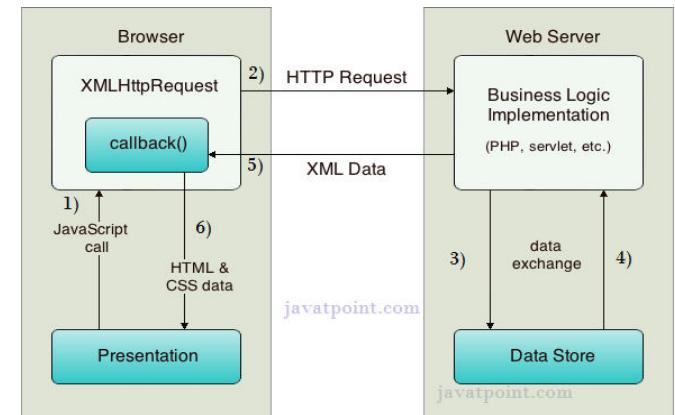
```
<body>
<script>
//Imagine we received this text from a web server:
  var str = '{ "name":"John", "age":30, "city":"NY" }';
//Use JSON.parse() to convert text into JavaScript object
  var obj = JSON.parse(str);
</script>
<p id="p1"></p>
<script>
document.getElementById("p1").innerHTML = obj.name;
</script>
```

typeof(obj) returns object

What is Ajax ?

- “Asynchronous JavaScript And XML”
 - AJAX is not a programming language, but a technique for making the user interfaces of web applications more responsive and interactive
 - It provide a simple and standard means for a web page to communicate with the server without a complete page refresh.
- Why Ajax?
 - Intuitive and natural user interaction
 - No clicking required. Call can be triggered on any event
 - Mouse movement is a sufficient event trigger
 - "Partial screen update" replaces the "click, wait, and refresh" user interaction model
 - Only user interface elements that contain new information are updated (fast response)
 - The rest of the user interface remains displayed as it is without interruption (no loss of operational context)

AJAX architecture



XMLHttpRequest

- JavaScript object - XMLHttpRequest object for asynchronously exchanging the XML data between the client and the server
- XMLHttpRequest Methods
 - open("method", "URL", syn/asyn) : Assigns destination URL, method, mode
 - send(content) : Sends request including string or DOM object data
 - abort() : Terminates current request
 - getAllResponseHeaders() : Returns headers (labels + values) as a string
 - getResponseHeader("header") : Returns value of a given header
 - setRequestHeader("label", "value") : Sets Request Headers before sending
- XMLHttpRequest Properties
 - onreadystatechange : Event handler that fires at each state change
 - readyState values – current status of request
 - Status : HTTP Status returned from server: 200 = OK
 - responseText : get the response data as a string
 - responseXML : get the response data as XML data

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

Creating an AJAX application

- Step 1: Get an instance of XHR object

```
xhr = new XMLHttpRequest();
```

- Step 2: Make the request

```
xhr.open('GET', 'http://www.example.org/some.file', true);
xhr.send(null);
```

```
xhr.open("POST", "AddNos.jsp");
xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
xhr.send("tno1=100&tno2=200");
```

- Step 3 : Attach callback function to xhr object

```
function loadDoc() {
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            document.getElementById("demo").innerHTML = this.responseText;
        }
    };
    xhttp.open("GET", "ajax_info.txt", true);
    xhttp.send();
}
```

Ajax Demo

```
<script>
var xhr;
function getData(){
    xhr = new XMLHttpRequest();

    if(xhr){
        xhr.open("GET", "Sample.txt", true);
        xhr.send();

        xhr.onreadystatechange = function(){
            if(xhr.readyState == 4 && xhr.status == 200){
                document.getElementById("lblresult").innerHTML=xhr.responseText;
            }
        } //end of callback function
    }
}

</script> <body>
<input type="button" onclick="getData()" value="Getresult"/>
<div id="lblresult"></div>
</body>
```

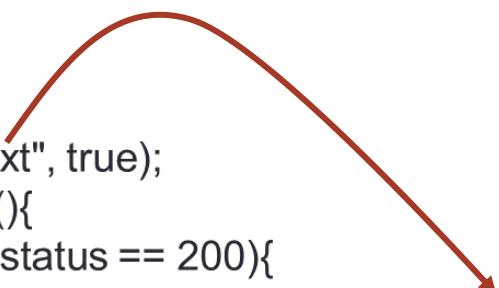
//sample.txt
hi how r u this is the response data from file

Getresult

hi how r u this is the response data from file

AJAX Demo with JSON

```
<script>
var xmlhttp;
function getData(){
    getHTTPRequestObject();
    if(xmlhttp){
        xmlhttp.open("GET", "EmpJSONData.txt", true);
        xmlhttp.onreadystatechange = function(){
            if(xmlhttp.readyState == 4 && xmlhttp.status == 200){
                var obj = JSON.parse(xmlhttp.responseText);
                var displaytext = "";
                displaytext += "Emp name : " + obj.name + "<br>" +
                    "Designation : " + obj.desig + "<br>" +
                    "Age : " + obj.age + "<br>" +
                    "Salary : " + obj.sal;
                document.getElementById("lblres").innerHTML = displaytext;
            }
        }
    }
}</script><body>
<h3 id="lblres">Result</h3>
<input type="button" id="btgetJSONdata" onclick="getData()" value="GetData">
</body>
```



```
{ "name":"Kapil Verma",
  "desig":"ASE",
  "age":23,
  "sal":22000
}
```

Emp name : Kapil Verma
Designation : ASE
Age : 23
Salary : 22000

GetJsonData

```

<html>
<head>
<script>

function sendRequest(){
    //step 1: create XHR object
    var xhr=new XMLHttpRequest();
    //step 2: Generate Request
    var url="https://jsonplaceholder.typicode.com/users";
    xhr.open("GET",url,true);
    //step 2: Define onreadystatechange function
    xhr.onreadystatechange=function(){
        if(xhr.readyState==4){
            var data=xhr.responseText;
            var jsdata=JSON.parse(data);
            // alert(jsdata.length);
            var str=<select>;
            for(var i=0;i<jsdata.length;i++){
                str+=<option value='"+jsdata[i].id+"'>"+jsdata[i].id+"---"+jsdata[i].name+"---"+jsda
            } //end of for
            str+=</select>;
            document.getElementById("mydiv").innerHTML=str;
        }
    } // end of onreadystatechange function
    xhr.send();
}

</script>
</head>

```

```

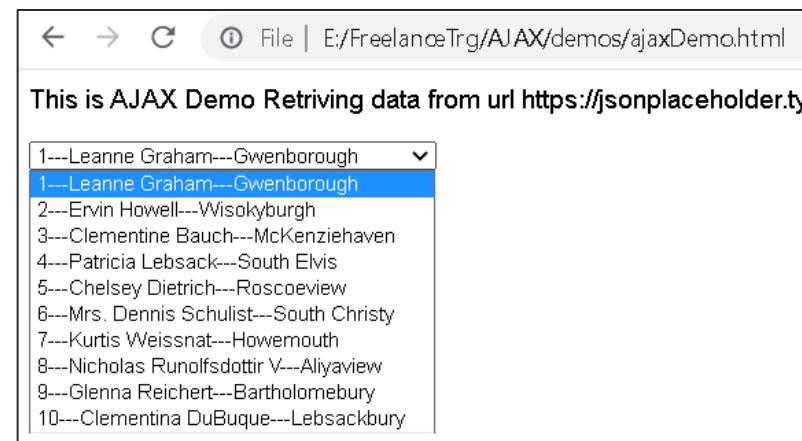
<body>


This is AJAX Demo Retriving data from url  

https://jsonplaceholder.typicode.com/users


<div id="mydiv"></div>
<button type="button" onclick="sendRequest()">GetData</button>
</body>
</html>

```



JQUERY

The screenshot shows the official jQuery website at jquery.com. The page features a dark header with a red 'Donate' button. Below the header is a blue navigation bar with links for 'Download', 'API Documentation', 'Blog', 'Plugins', and 'Browser Support'. The main content area highlights 'Lightweight Footprint', 'CSS3 Compliant', and 'Cross-Browser' support. A prominent orange 'Download jQuery v3.5.1' button is centered, with a note below it stating: 'The 1.x and 2.x branches no longer receive patches.' There's also a link to 'View Source on GitHub →'.

Software

The screenshot shows the 'Downloading jQuery' page at jquery.com/download/. The page title is 'Downloading jQuery'. It explains that compressed and uncompressed files are available, noting that the uncompressed file is best for development and the compressed file saves bandwidth and improves performance. It mentions that source maps are included in compressed files but not in the compressed file itself. Below this text is a note about upgrading to version 3.5.1. At the bottom, there are two download links: 'Download the compressed, production jQuery 3.5.1' and 'Download the uncompressed, development jQuery 3.5.1'.

• Choosing a Text Editor

- Text editors that support jQuery include Brackets, Sublime Text, Kwrite, Gedit, Notepad++, PSPad, or TextMate.

jQuery Introduction

- jQuery is a lightweight, cross browser and feature-rich JavaScript library which is used to manipulate DOM
 - Originally created by John Resig in early 2006.
 - The jQuery project is currently run and maintained by a distributed group of developers as an open-source project.
- Why jQuery
 - JavaScript is great for a lot of things especially manipulating the DOM but it's pretty complex stuff. jQuery abstracts away a lot of the complexity involved in dealing with the DOM, and makes creating effects super easy.
 - It can locate elements with a specific class
 - It can apply styles to multiple elements
 - It solves the cross browser issues
 - It supports method chaining
 - It makes the client side development very easy

Including jQuery in HTML Document

- jQuery library can be included in a document by linking to a local copy or to one of the versions available from public servers.
- Eg : include a local copy of the jQuery library

```
<html>
<head>
<title>Test jQuery</title>
<script src="../scripts/jquery-3.5.1.min.js"></script>
</head>
<body>
    <!-- body of HTML -->
</body>
</html>
```

- Eg : include the library from a publicly available repository
 - There are several well-known public repositories for jQuery; these repositories are also known as Content Delivery Networks (CDNs).

```
<script src="https://code.jquery.com/jquery-3.1.1.min.js"></script>
<script src="http://ajax.aspnetcdn.com/ajax/jquery/jquery-3.1.1.min.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.1.1/jquery.min.js"></script>
```

Using jQuery

```
<html>
<head>
<title>Test jQuery</title>
<script type="text/javascript" src="jquery-3.5.1.js"></script>
<script type="text/javascript">
$(document).ready(function() {
    alert('Hi');
});
</script>
</head>
<body>
    Welcome to jQuery
</body>
</html>
```

```
//shortcut...
$(function() {
    // jQuery code
});
```

Introduction to selectors

- jQuery uses same CSS selectors used to style html page to manipulate elements
 - CSS selectors select elements to add style to those elements where as jQuery selectors select elements to add behavior to those elements.
 - Selectors allow page elements (Single or Multiple) to be selected.
- Selector Syntax
 - `$(selectorExpression)`
 - `jQuery(selectorExpression)`
 - `$(selector).action()`

```
<html>
<head>
<title>Test jQuery</title>
<script src="../scripts/jquery-3.5.1.min.js"></script>
<script>
$(document).ready(function() {
    $("h2").css("color","red");
    jQuery("h1").html("New Header-1");
});
</script>
</head>
<body>
    <h1>jQuery Enabled</h1>
    <p>Para-1</p>
    <p>Para-2</p>
    <h2>Header2</h2>
</body>
</html>
```

New Header-1

Para-1

Para-2

Header2

Selectors

- **Selecting by Tag Name:**
- Selecting single tag takes the following syntax
 - `$('p')` – selects all `<p>` elements
 - `$('a')` – selects all `<a>` elements
- To reference multiple tags, use the (,) to separate the elements
 - `$('p, a, span')` - selects all paragraphs, anchors and span elements

• **Selecting Descendants**

- `$('ancestor descendant')` - selects all the descendants of the ancestor
 - `$('table tr')` - Selects all `tr` elements that are the descendants of the `table` element
- Descendants can be children, grand children etc of the designated ancestor element.

Demo

```
<style type="text/css">
.redDiv{background-color:red; color:white;}
</style>
<script src=..\Scripts\jquery-3.5.1.js></script>
<script>
$(document).ready(function() {
    var paragraphs = $('p');
    alert(paragraphs.length); //4
    paragraphs.css('background-color','blue');
    paragraphs.each(function(){
        alert($(this).html());
    });
    var collections = $('div,p');
    alert(collections.length); //6
    var bodydivs = $('body div');
    alert(bodydivs.length); //2
});
</script>
</head>
<body>
<div class="redDiv" >


Para-1



Para-2

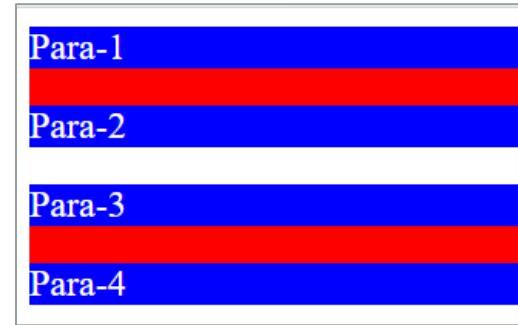

</div>
<div class="redDiv">


Para-3



Para-4


</div>
</body>
```



Selecting by Element ID

- It is used to locate the DOM element very fast.
- Use the # character to select elements by ID
 - `$("#first")` — selects the element with `id="first"`
 - `$('#myID')` – selects the element with `id=" myID "`

```
<script src=".\\Scripts\\jquery-3.5.1.js"></script>
<script>
$(document).ready(function()
{
    alert($('#testDiv').html());
    $('#testDiv').html("Changed Text in Div!!");
    $('#p1').hide();
});
</script>
</head>
<body>
<div id="testDiv">Test Div</div>
<p id="p1"> Para-1</p>
<p>Para-2</p>
</body>
```

Test Div

Para-1

Para-2

Changed Text in Div!!

Para-2

Selecting Elements by Class Name

- Use the (.) character to select elements by class name
 - `$(".intro")` — selects all elements with class="intro"
- To reference multiple tags, use the (,) character to separate class name.
 - `('.blueDiv, .redDiv')` - selects all elements containing class blueDiv and redDiv
- Tag names can be combined with elements name as well.
 - `('div.myclass')` – selects only those `<div>` tags with class="myclass"

```
<script>
$(document).ready(function(){
    $(".bold").css("font-weight", "bold");
});
</script>
</head>
<body>
<ul>
    <li class="bold">Test 1</li>
    <li>Test 2</li>
    <li class="bold">Test 3</li>
</ul>
</body>
```

- Test 1
- Test 2
- Test 3

```
<style type="text/css">
.blueDiv{background-color:lightblue; color:darkblue;}
.redDiv{background-color:orange; color:red;}
</style>
<script src=..\Scripts\jquery-3.5.1.js></script>
<script>
$(document).ready(function(){
    var collection = $('.blueDiv');
    collection.css('border','5px solid blue');
    collection.css('padding','10px');
});
</script>
</head>
<body>
<div class="blueDiv">
<p>para-1</p>
</div>
<div class="redDiv">
<p>para-2</p>
</div>
<div class="blueDiv">
<p>para-3</p>
</div>
</body>
```

para-1

para-2

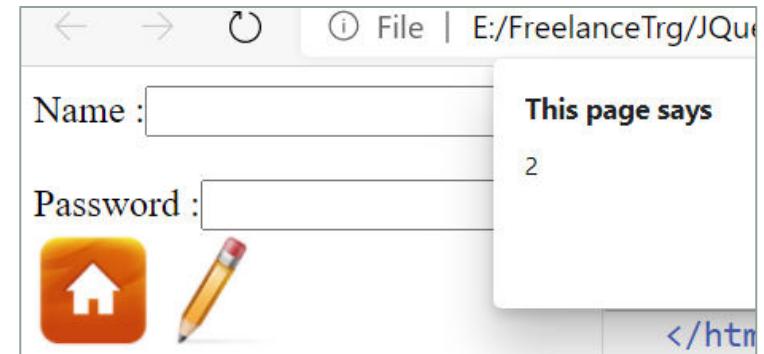
para-3

Selecting by attribute values

- Use brackets [attribute] to select on attribute name and/or attribute value
 - `$(‘a[title]’)` - selects all anchor elements that have a title attribute
 - `$(‘a[title=“trainer”]’)` – selects all `<a>` elements that have a “trainer” title attribute value

```
<script src=".\\Scripts\\jquery-3.5.1.js"></script>
<script>
$(document).ready(function()
{
    var list = $('div[title="first"],img[height]' );
    alert(list.length); //2
});
</script>
</head>
<body>
<div title="first">
Name :<input type="text"/>
</div><br>
<div title="second" >
Password :<input type="password"/>
</div>


</body>
```



Selecting by input elements

- To select input elements of type : <input>:
 - `$('input[type="text"]').css("background", "yellow");`
- To select all input elements
 - `$(':input')` - Selects all form elements (input, select, textarea, button).
 - `$(':input[type="radio"]')` – selects all radio buttons
 - `$(":text")` - All input elements with type="text"
 - `$(":password")` - All input elements with type="password"
 - `$(":radio")` - All input elements with type="radio"
 - `$(":checkbox")` - All input elements with type="checkbox"
 - `$(":submit")` - All input elements with type="submit"
 - `$(":reset")` - All input elements with type="reset"
 - `$(":button")` - All input elements with type="button"
 - `$(":file")` - All input elements with type="file"

```
<script src=".\\Scripts\\jquery-3.5.1.js"></script>
<script>
$(document).ready(function()
{
    var inputs = $(':input');
    alert($(inputs[2]).val()); //Chennai
    $(":text").css({ background: "yellow", border: "3px red solid" });
});
</script>
</head>
<body>
Name : <input id="txtName" type="text" value="Karthik"><br>
Age : <input id="txtAge" type="text" /><br>
City :
<select id="city">
<option value="Bangalore">Bangalore</option>
<option value="Chennai" selected="selected">Chennai</option>
<option value="Mumbai">Mumbai</option>
</select><br>
</body>
```

| | |
|--------|-----------|
| Name : | Karthik |
| Age : | |
| City : | Chennai ▾ |

Filters

- The **index-related selectors** (`:eq()`, `:lt()`, `:gt()`, `:even`, `:odd`) filter the set of elements that have matched the expressions that precede them.
 - They narrow the set down based on the order of the elements within this matched set.
 - Eg, if elements are first selected with a class selector (`.myclass`) and four elements are returned, these elements are given indices 0 through 3
- **`eq()`** - Select the element at index n within the matched set.
 - Eg : `$("p:eq(1)"`) - Select the second `<p>` element
 - Eg : `$("element:eq(0)"`) is same as `$('element:first-child')`
- **`$('element:odd')` and `$('element:even')`** selects odd and even positions respectively. **0 based indexing**
 - Odd returns (1,3,5...) and Even returns (0,2,4...)
- **`:gt()` and `lt()`** - Select all elements at an index > or < index within the matched set
 - Eg : `$("tr:gt(3)"`) : Select all `<tr>` elements after the 4 first
 - Eg : `$("tr:lt(4)"`) : Select the 4 first `<tr>` elements

```

<table border="1">
  <tr><td>TD #0</td><td>TD #1</td><td>TD #2</td></tr>
  <tr><td>TD #3</td><td>TD #4</td><td>TD #5</td></tr>
  <tr><td>TD #6</td><td>TD #7</td><td>TD #8</td></tr>
</table>
<script>
$( "td:eq( 2 )" ).css( "color", "red" );
//$( "tr:first" ).css( "font-style", "italic" ); //is same as below line
$( "tr:eq(0)" ).css( "font-style", "italic" );
$( "td:gt(4)" ).css( "backgroundColor", "yellow" );
</script>

```

| | | |
|-------|-------|-------|
| TD #0 | TD #1 | TD #2 |
| TD #3 | TD #4 | TD #5 |
| TD #6 | TD #7 | TD #8 |

```

<script type="text/javascript">
$(document).ready(function() {
  $('tr:odd').css('background-color','tomato');
  $('tr:even').css('background-color','bisque');
});
</script>
<table border=1 cellspacing=5 cellpadding=5>
  <th>column 1<th>column 2<th>column 3
  <tr><td>data 1</td><td>data 2</td><td>data 3
  <tr><td>data 4</td><td>data 5</td><td>data 6
  <tr><td>data 7</td><td>data 8</td><td>data 9
  <tr><td>data 10</td><td>data 11</td><td>data 12
</table>

```

| column 1 | column 2 | column 3 |
|----------|----------|----------|
| data 1 | data 2 | data 3 |
| data 4 | data 5 | data 6 |
| data 7 | data 8 | data 9 |
| data 10 | data 11 | data 12 |

Basic Filters continued

- **:first , :last** - Selects the first/last matched element.
 - :first is equivalent to :eq(0) and :lt(1). This matches only a single element, whereas, [:first-child](#) can match more than one; one for each parent.
- **:header** - Selects all elements that are headers, like h1, h2, etc

```
<script>
$(document).ready(function() {
    $(":header").css({ background: "#ccc", color: "blue" });
    $('tr:first-child').css('background-color','tomato');
});
</script>
</head>
<body>
<h1>Table 1</h1>
<table>
    <tr><td>Row 1</td></tr>
    <tr><td>Row 2</td></tr>
    <tr><td>Row 3</td></tr>
</table>
<br/><br/>
<h2>Table 2</h2>
<table border=1>
    <th>column 1<th>column 2<th>column 3
    <tr><td>data 1</td><td>data 2</td><td>data 3
```

Table 1

Row 1
Row 2
Row 3

Table 2

| column 1 | column 2 | column 3 |
|----------|----------|----------|
| data 1 | data 2 | data 3 |
| data 4 | data 5 | data 6 |
| data 7 | data 8 | data 9 |

Child Filter

- `$('element:first-child')` and `$('element:last-child')` selects the first child & last child of its parent.
 - `$('span:first-child')` returns the span which is a first child for all the groups
- `:nth-child()` - Selects all elements that are the nth-child of their parent. **1-based indexing**
 - Eg : `$(".p:nth-child(3)")` - Select each `<p>` element that is the third child of its parent

```
<style>
span {color: blue; }
</style>
</head>
<body>
  <ul>
    <li>John</li>
    <li>Karl</li>
    <li>Brandon</li>
  </ul> <hr>
  <ul><li>Sam</li></ul> <hr>
  <ul>
    <li>Glen</li>
    <li>Tane</li>
  </ul>
<script>
$( "ul li:nth-child(2)" ).append( "<span> - 2nd!</span>" );
</script>
```

- John
- Karl - 2nd!
- Brandon

- Sam

- Glen
- Tane - 2nd!

```
<style>
    span {color: #008;}
    span.sogreen { color: green; font-weight: bolder;}
    span.solast {text-decoration: line-through;}
</style>
<script src=..\Scripts\jquery-3.5.1.min.js></script>
</head>
<body>
<div> <span>John,</span><span>Karl,</span><span>Brandon</span></div>
<div> <span>Glen,</span> <span>Tane,</span><span>Ralph</span> </div>
<script>
$( "div span:first-child" ) .css( "text-decoration", "underline" )
    .hover(function() {
        $( this ).addClass( "sogreen" );
    }, function() {
        $( this ).removeClass( "sogreen" );
    });
$( "div span:last-child" ).css({ color:"red", fontSize:"80%" })
.hover(function() {
    $( this ).addClass( "solast" );
}, function() {
    $( this ).removeClass( "solast" );
});
$("div span:nth-child(2)").css("background-color","yellow");
</script>
```

John, Karl, Brandon
Glen, Tane, Ralph

John, Karl, Brandon
Glen, Tane, Ralph

John, Karl, Brandon
Glen, Tane, Ralph

Content Filters

- **:contains()** will select elements that match the contents.
 - `$('div:contains("hello")')` - selects div's which contains the text hello (match is case sensitive)
- **:empty** - Select all elements that have no children (incl text nodes)
- **:has** : Selects elements which contain at least one element that matches the specified selector.
 - Eg : `$("p:has(span)")` - Select all `<p>` elements that have a `` element inside of them
 - Eg : `$("div:has(p,span,li)").css("border","solid red")`; - Select all `<div>` elements that have at least one of the given elements inside
- **:parent** - Select all elements that have at least one child node (either an element or text).
 - Eg : `$("td:parent")` - Select all `<td>` elements with children, including text

```

<style>
.test { border: 3px inset red; width:250px }
</style>
</head>
<body>
<table border="1">


```

| | |
|------------|------------|
| TD #0 | Was empty! |
| TD #2 | Was empty! |
| Was empty! | TD#5 |

John Resig
 George Martin
Malcom John Sinclair
 J.Ohn

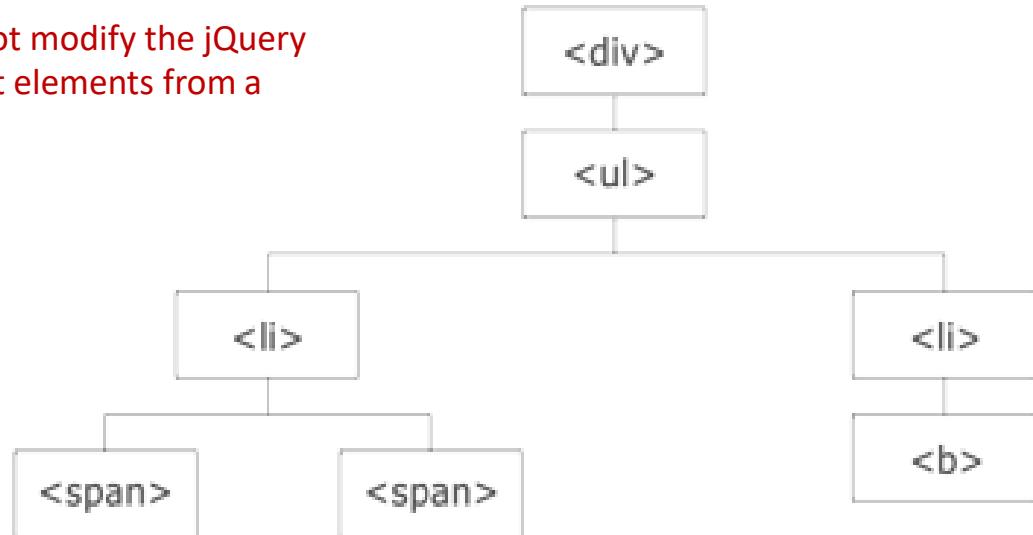
Hello in a paragraph

Hello again! (with no paragraph)

jQuery Traversing

- jQuery traversing, which means "move through", are used to "find" (or select) HTML elements based on their relation to other elements.
 - Start with one selection and move through that selection until you reach the elements you desire.

Most of the DOM Traversal Methods do not modify the jQuery DOM object and they are used to filter out elements from a document based on given conditions

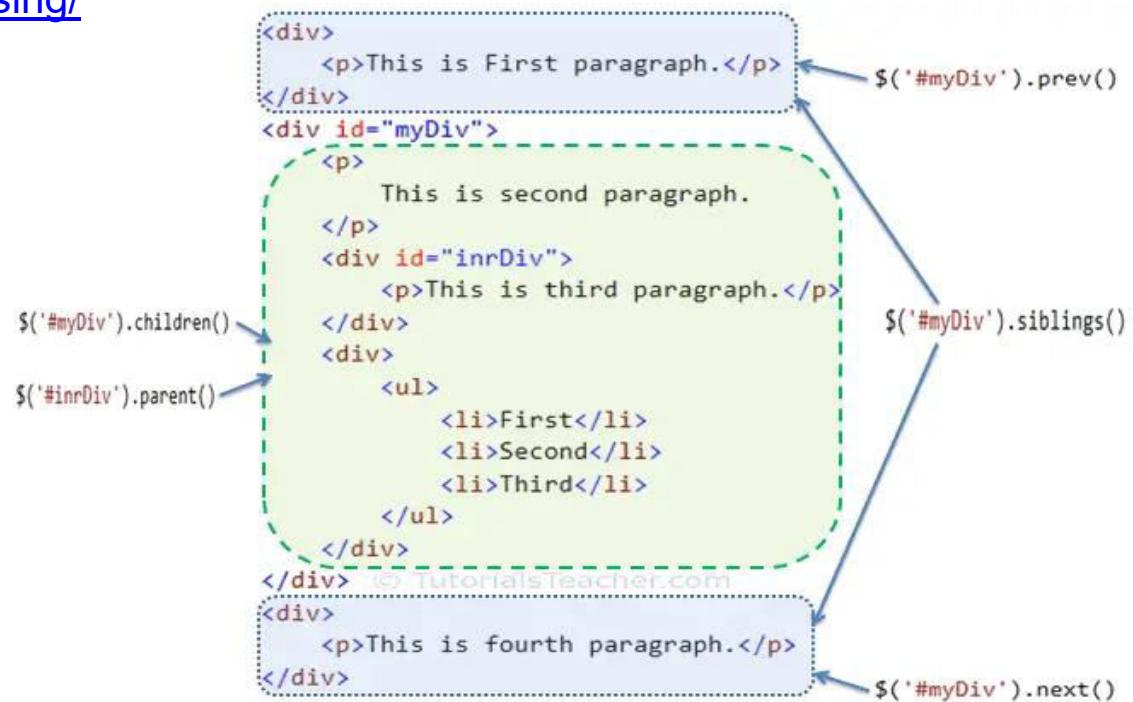


- The <div> element is the **parent** of , and an **ancestor** of everything inside of it
- The element is the **parent** of both elements, and a **child** of <div>
- The left element is the **parent** of , **child** of and a **descendant** of <div>
- The element is a **child** of the left and a **descendant** of and <div>
- And so on....

| Methods | Description |
|------------|--------------------------------------------------------------------------------------------|
| children() | Get all the child elements of the specified element(s) |
| each() | Iterate over specified elements and execute specified call back function for each element. |
| find() | Get all the specified child elements of each specified element(s). |
| first() | Get the first occurrence of the specified element. |
| next() | Get the immediately following sibling of the specified element. |
| parent() | Get the parent of the specified element(s). |
| prev() | Get the immediately preceding sibling of the specified element. |
| siblings() | Get the siblings of each specified element(s) |

jQuery methods for traversing DOM elements

<https://api.jquery.com/category/traversing/>



jQuery Traversing -> filtering

- **.eq()** - Reduce the set of matched elements to the one at the specified index.
 - Eg : \$("p").eq(1).css("background-color", "yellow") - Select the second <p> element (index number 1)
- **.filter()** - Reduce the set of matched elements to those that match the selector or pass the function's test
 - Eg : \$("p").filter(".intro") - Return all <p> elements with class "intro"
- **.first() / last()**
 - Eg : \$("div p").first() - Select first <p> element inside first <div> element
- **.has()** - Reduce the set of matched elements to those that have a descendant that matches the selector or DOM element.
 - Eg : \$("p").has("span") - Return all <p> elements that have element inside

jQuery Traversing -> Tree Traversal

- `.children()` - Returns all direct children of the selected element
 - Eg : `$("ul").children().css({"color":"red","border":"2px solid red"})` - Return elements that are direct children of ``

```
<ul class="ulclass">
  <li>Mocha</li>
  <li>Expresso</li>
  <li>Cappuchino</li>
  <li>Latte</li>
</ul>
<script>
$(".ulclass").children().css({"color":"red","border":"2px solid red"})
</script>
```

- Mocha
- Expresso
- Cappuchino
- Latte

- `.find()` - Returns descendant elements of the selected element
- `.next() / prev()` - Returns the next / previous sibling element of the selected element
- `nextAll()` - returns all next sibling elements of the selected element
- `parent()` - Returns the direct parent element of the selected element

```
<body>
  <div class="great-grand-parent">
    <div class="grand-parent">
      <ul class="parent">
        <li class="child-one">Child One</li>
        <li class="child-two">Child Two</li>
      </ul>
    </div>
  </div>
<script>
$( ".child-two" ).parent().css( "border", "2px solid red" );
</script>
```

- Child One
- Child Two

```
<body>
  <h3>Highlight all nested lists</h3>
  <ul>
    <li>Java
      <ul>
        <li>Servlets</li>
        <li>JSP</li>
        <li>Spring</li>
      </ul>
    </li>
    <li>Python</li>
    <li>DotNet
      <ul>
        <li>C#</li>
        <li>VB.net</li>
      </ul>
    </li>
    <li>PHP</li>
    <li>NodeJS</li>
  </ul>
<script>
  $("ul").find("li ul").css("background-color", "yellow");
</script>
```

- `children()` - returns all the direct children of the matched element.
- `find()` - returns all the descendant elements of the matched element.

Highlight all nested lists

- Java
 - Servlets
 - JSP
 - Spring
- Python
- DotNet
 - C#
 - VB.net
- PHP
- NodeJS

```
<style>
.blue {background: blue;}
.highlight{background-color: pink}
</style>
<script src="..\Scripts\jquery-3.5.1.min.js"></script>
</head>
<body>
<div>
<p>In inner para-1</p>
<p class="intro">In inner para-2</p>
</div>
<p id="outro">In outer para-1</p>
<p>In outer para-2</p>
<ul>
<li>list item 1</li>
<li>list item 2</li>
<li>list item 3</li>
<li>list item 4</li>
<li>list item 5</li>
</ul>
</div>
<script>
$( "li" ).eq(4).css( "background-color", "red" );
$( "body" ).find( "li" ).eq( 2 ).addClass( "blue" );
$("li").filter(":even").css("background-color","yellow");
$("p").filter(".intro,#outro").css("background-color","blue");
$( "div p" ).first().addClass( "highlight" );
</script>
```

In inner para-1

In inner para-2

In outer para-1

In outer para-2

- list item 1
- list item 2
- list item 3
- list item 4
- list item 5

Method Chaining

- Chaining is a good way to avoid selecting elements more than once. Eg:
 - `$("#div").fadeOut();`
 - `$("#div").css("color", "red");`
 - `$("#div").text("hello world");`
- Instead of doing that and running `$("#div")` three times, you could do this:
 - `$("#div").fadeOut().css("color", "red").text("hello world");`

ATTRIBUTES AND TEMPLATES

Working with Attributes

- Object attributes can be used using attr():
 - var val = \$('#customDiv').attr('title'); - Retrieves the title attribute value
- **.attr(attributeName,value)** : accesses an object's attributes and modifies the values.
 - \$('img').attr('title','Image title'); - changes the title attribute value to Image title.
- To modify multiple attributes, pass JSON object.

```
$("img").attr({  
    "title": "image title",  
    "style" : "border:5px dotted red"  
});
```



```

```

- You can also remove attributes entirely using **.removeAttr()**.

jQuery Templates

- Template is a form, model, sample or predefined shape used to provide consistent look and feel to information presented on a website
 - jQuery Templates are client-side based templates.
 - [JQuery templating let you render JSON to HTML elements.](#)
- The benefits of using jQuery Templates are:
 - Easily convert JSON object to HTML without need for parsing
 - Reusability
 - Rendered and cached on client-side
 - Templates are written in pure HTML with Template Tags and simple jQuery code for magic to happen
 - Maximize the separation of UI and DATA
 - jQuery Templates is an official plugin for the jQuery Library. We need to add reference to the jQuery Templates library.
 - `<script
src="http://ajax.microsoft.com/ajax/jquery.templates/beta1/jquery tmpl.min.js" />`
 -

```

<script>
$(document).ready(function() {
    var studList = [
        { Name: "Anil", Surname: "Patil", Grade: 'A' },
        { Name: "Soha", Surname: "Kumari", Grade: 'B' },
        { Name: "Arnav", Surname: "Rao", Grade: 'A' },
        { Name: "Vanita", Surname: "Kapoor", Grade: 'B' }
    ];
    $("#studTemplate").tmpl(studList).appendTo("#students");
});
</script>

<script id="studTemplate" type="text/html">
<tr><td>${Name}</td><td>${Surname}</td>
    {{if Grade=='A'}}
        (<td style='color:red'>First Class</td>)
    {{else}}
        (<td style='color:blue'>Second-class</td>)
    {{/if}}
</tr>
</script>

<body>
    <table border="1" id="students"></table>
</body>
</html>

```

| | | |
|--------|--------|--------------|
| Anil | Patil | First Class |
| Soha | Kumari | Second-class |
| Arnav | Rao | First Class |
| Vanita | Kapoor | Second-class |

Getting Content

- `text()` - Sets or returns the **text** content of selected elements
- `html()` - Sets or returns the content of selected elements (including **HTML** markup)
- `val()` - Sets or returns the value of form fields

```
<script>
$(document).ready(function() {
    $("#div1").html('<a href="example.html">Link</a><b>hello</b>');
    $("#div2").text('<a href="example.html">Link</a><b>hello</b>');
});
</script>
</head>
<body>
<div id="div1"></div>
<div id="div2"></div>
</body>
```

`$.html()` treats the string as
HTML, `$.text()` treats the content
as text

Linkhello

Linkhello

```
<div id="div1">
    <a href="example.html">Link</a>
    <b>hello</b>
</div>
<br>
<div id="div2"><a href="example.html">Link</a><b>hello</b>
</div>
```

Adding and Removing Nodes

- In traditional approach adding and removing nodes is tedious.
- To insert nodes four methods available:
- Appending adds children at the end of the matching elements
 - `.append()`
 - `.appendTo()`
- Prepending adds children at the beginning of the matching elements
 - `.prepend()`
 - `.prependTo()`
- To wrap the elements use `.wrap()`
 - Eg: `$(“p”).wrap(“<h1></h1>”)` //wraps “p” in `<h1>` tags
- To remove nodes from an element use `.remove()`

```
<script>
$(document).ready(function() {
    $("button").click(function(){
        $("h2").append("<p>Surprise text!</p>");
    });
});
</script>
</head>
<body>
    <h2>This is a header.</h2>
    <button>Click me</button>
</body>
```

This is a header.

Click me

This is a header.

Surprise text!

Click me

```
<body> == $0
<h2>This is a header.</h2>
<button>Click me</button>
</body>
```

```
<body> == $0
<h2>
    "This is a header."
    <p>Surprise text!</p>
</h2>
<button>Click me</button>
</body>
```

```
<script>
$(document).ready(function() {
    $("button").click(function(){
        $("h2").prepend("<p>Surprise text!</p>");
    });
});
</script>
```

Surprise text!

This is a header.

Click me

```
• <body>
  • <h2> == $0
    <p>Surprise text!</p>
    "This is a header."
    <button>Click me</button>
  </body>
```

```

<script>
$(document).ready(function() {
    $('button').click(function() {
        //append can be a element, but can be simple text too
        $('#hello').append(' world!');
    });
});
</script>
</head>
<body>
    <p id="hello">Hello</p>
    <button>Click me</button>
</body>

```



```

• ▼<body> == $0
  <p id="hello">Hello</p>
  <button>Click me</button>
</body>

```

```

▼<body>
.. ▼<p id="hello"> == $0
  "Hello"
  " world!"
</p>
<button>Click me</button>
</body>

```

```

<script>
$(document).ready(function() {
    $("#btn").click(function(){
        $("h2").remove();
    });
});
</script>
<body>
    <button id="btn">Remove</button><br>
    <h2>Press the button to remove this heading</h2>
</body>

```

Remove

Press the button to remove this heading

```

<script type="text/javascript">
$(document).ready(function(){
    var i=3;
    $("#btn").click(function(){
        $("ol").append("<li>New Item " + i++ + "</li>");
    });
});
</script>
</head>
<body>
    <button id="btn">Append</button>
    <ol>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
    </ol>
</body>

```

Append

1. Item 1
2. Item 2
3. Item 3

Append

1. Item 1
2. Item 2
3. Item 3
4. New Item 3
5. New Item 4

▼ <body>

<button id="btn">Append</button>

▼

- ▶ ...
- ▶ ...

▼ == \$0

::marker
"Item 3"

</body>

▼

- ▶ ...
- ▶ ...

▼

::marker
"Item 3"

- ▶ ...

▼ == \$0

::marker
"New Item 4"

```
<script>
$(document).ready(function() {
  $("#btn").click(function(){
    $("div").wrap $("<h3 style='text-align:center;color:blue'></h3>");
  });
})
</script>
<body>
  <div>Hello</div>
  <button id="btn">Wrap</button><br>
</body>
```



```
▼ <body> == $0
  <div>Hello</div>
  <button id="btn">Wrap</button>
  <br>
</body>
```



```
▼ <body>
  . ▼ <h3 style="text-align:center;color:blue"> == $0
    |   <div>Hello</div>
    |   </h3>
    |   <button id="btn">Wrap</button>
    |   <br>
    |   </body>
```

```
<script>
$(document).ready(function() {
    $("#btn").click(function(){
        $("h2").empty();
        $(".d1").empty();
    });
});
</script>
<body>
    <button id="btn">Remove</button><br>
    <h2>Remove contents of this heading</h2>
    <div class="d1">
        <p>Para-1</p>
        <p>Para-2</p>
    </div>
</body>
```

Remove

Remove contents of this heading

Para-1

Para-2

This method removes not only child (and other descendant) elements, but also any text within the set of matched elements.

```
▼ <body>
    <button id="btn">Remove</button>
    <br>
    <h2>Remove contents of this heading</h2>
    • ▼ <div class="d1"> == $0
        <p>Para-1</p>
        <p>Para-2</p>
    </div>
</body>
```

Remove

```
▼ <body>
    <button id="btn">Remove</button>
    <br>
    <h2></h2>
    • <div class="d1"></div> == $0
    </body>
```

Other DOM manipulation methods

- Methods that allow us to insert new content outside an existing element:
 - .after() : Insert content after each element in the set of matched elements.
 - .before() : Insert content before each element in the set of matched elements.
 - .insertAfter() : Insert every element in the set of matched elements after the target.
 - .insertBefore() : Insert every element in the set of matched elements before the target.
 - .replaceAll() : Replace each target element with the set of matched elements.
 - .replaceWith() : Replace each element in the set of matched elements with the provided new content and return the set of elements that was removed.

```
<script>
$(document).ready(function() {
  $("button").click(function(){
    $("h2").after("<p>Surprise text!</p>");
  });
});
</script>
</head>
<body>
  <h2>This is a header.</h2>
  <button>Click me</button>
</body>
```

This is a header.

Click me

This is a header.

Surprise text!

Click me

▼ <body> == \$0

```
<h2>This is a header.</h2>
<button>Click me</button>
</body>
```

▼ <body> == \$0

```
<h2>This is a header.</h2>
<p>Surprise text!</p>
<button>Click me</button>
</body>
```

```

<script>
$(document).ready(function() {
    $("#btn1").click(function(){
        $("<h3>New content</h3>").replaceAll("p");
    });
    $("#btn2").click(function(){
        $("p.p2").replaceWith("<h3>New heading</h3>");
    });
});
</script>
</head>
<body>
    <p class="p1">Para-1</p>
    <p class="p2">Para-2</p>
    <button id="btn1">Replace All</button>
    <button id="btn2">Replace With</button>
</body>

```

Para-1

Para-2

Replace All

Replace With

New content

New content

Replace All

Replace With

<body> == \$0

<h3>New content</h3>

<h3>New content</h3>

<button id="btn1">Replace All</button>

<button id="btn2">Replace With</button>

Para-1

New heading

Replace All

Replace With

<body> == \$0

<p class="p1">Para-1</p>

<h3>New heading</h3>

<button id="btn1">Replace All</button>

<button id="btn2">Replace With</button>

</body>

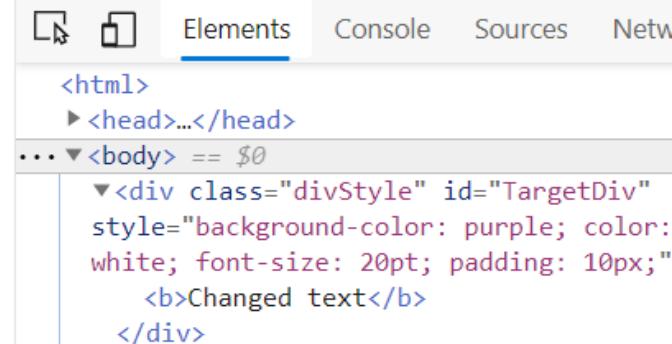
Modifying Styles

- `.css()` function is used to modify an object's style
 - `$(‘div’).css(‘color’,‘red’);`
- Multiple styles can be modified by passing a JSON Object

```
<script>
$(document).ready(function() {
    $('#TargetDiv').css({
        'background-color': 'purple',
        'color': 'white',
        'font-size': '20pt',
        'padding': '10px'
    })
    .html('<b>Changed text</b>');
});
</script>
</head>
<body>
<div id="TargetDiv">Target Div</div>
</body>
```

```
$(‘div’).css({
    “color”:“red”,
    “font-weight”:“bold”
});
```

Changed text



jQuery DOM utility functions

- The jQuery library defines a number of utility functions that are namespaced by `jQuery/$` and that don't operate on a jQuery object.
- Some of them are:
 - `$.each()` : is used to iterate over arrays and objects

```
$.each([ 33,44,55 ], function( index, value ) {
    console.log( index + ": " + value );
});
$.each(['Dia', 'Ria','Nia'], function( index, value ) {
    console.log( index + ": " + value );
});
var person = {
    "name": "shrilata",
    "email": "shri@gmail.com",
};
$.each( person, function( key, value ) {
    console.log( key + ": " + value );  });


```

| |
|-----------------------|
| 0: 33 |
| 1: 44 |
| 2: 55 |
| 0: Dia |
| 1: Ria |
| 2: Nia |
| name: shrilata |
| email: shri@gmail.com |

- `$.isArray(obj)` : determine whether the argument is an array.
- `$.inArray()` : is used to returns a value's index in an array, or -1 if the value is not in the array.

```
console.log($.isArray([1,2,3])) //true
```

```
var arr = [22,33,44,55];
console.log($.inArray(44,arr)) //2
```

jQuery DOM utility functions

- **\$.extend(target, object1 [, objectN])** : Merge the contents of two or more objects together into the first object.

```
var emp = {  
    fname: 'Anita',  
    lname: 'Patil',  
    skills: ['Java', 'Spring']  
};  
var project = {  
    details: { name: 'Claims', client: 'SCE' },  
    duration: 100  
};  
  
// Merge object2 into object1  
$.extend( emp, project );  
console.log(emp)
```

► details: {name: 'Claims', client: 'SCE'}
duration: 100
fname: "Anita"
lname: "Patil"
► skills: (2) ['Java', 'Spring']

- **\$.merge(first, second)** : merges the contents of two arrays together into the first array.

```
var first = [ "a", "b", "c" ];  
var second = [ "d", "e", "f" ];  
$.merge(first, second);  
console.log(first) //['a', 'b', 'c', 'd', 'e', 'f']
```

Working with Classes

- The four methods for working with css class attributes are
- **.addClass()** : adds one or more classes to the class attribute of each element.
 - `$('p').addClass('classOne');`
 - `$('p').addClass('classOne classTwo');`
- **.hasClass()** : returns true if the selected element has a matching class
 - `if($('p').hasClass('classOne')) { //perform operation}`
- **removeClass()** remove one or more classes
 - `$('p').removeClass('classOne classTwo');`
- To remove all class attributes for the matching selector
 - `$('p').removeClass();`
- **.toggleClass()** : alternates adding or removing a class based on the current presence or absence of the class.
 - `$('#targetDiv').toggleClass('highlight');`

```
<style>
.important{
    font-weight:bold;
    font-size:xx-large;
    color:red;
}
.blue {color:blue;}
</style>
<script>
$(document).ready(function() {
    $("button").click(function(){
        $("h1,h2,p").addClass("blue");
        $("div").addClass("important");
    });
});
</script>
</head>
<body>
<h1>Heading 1</h1>
<h2>Heading 2</h2>
<p>This is a paragraph.</p>
<p>This is another paragraph.</p>
<div>This is some important text!</div>
<button>Add classes to elements</button>
```

Heading 1

Heading 2

This is a paragraph.

This is another paragraph.

This is some important text!

Add classes to elements

Heading 1

Heading 2

This is a paragraph.

This is another paragraph.

This is some

Add classes to elements

Working with Classes (Contd)

```
<style type="text/css">
.highlight{ background-color:yellow; }
</style>
<script>
$(document).ready(function() {
    $('input[type="text"]').addClass('highlight');
    //$('#txtLastName').removeClass('highlight');
});
function FocusBlur(element) {
    $(element).toggleClass('highlight');
}
</script>
</head>
<body>
<table>
<tr>
<td>FirstName : </td>
<td><input id="txtFirstName" onFocus="FocusBlur(this)"
           onBlur="FocusBlur(this)" /></td>
</tr>
<tr>
<td>LastName : </td>
<td><input id="txtLastName" onFocus="FocusBlur(this)"
           onBlur="FocusBlur(this)" /></td>
</tr>
</table>
```

FirstName :

LastName :

jQuery Event Model Benefits

- Events notify a program that a user performed some type of action
- jQuery Events
 - click()
 - blur()
 - focus()
 - dblclick()
 - mousedown()
 - mouseup()
 - mouseover()
 - keydown()
 - keypress()
 - hover() : hover() method takes two functions and is a combination of the mouseenter() and mouseleave() methods.

```
$("img").mouseover(function () {  
    $(this).css("opacity", "0.3");  
});  
$("img").mouseout(function () {  
    $(this).css("opacity", "1.0");  
});
```



```
$("#p1").hover(function(){  
    alert("You entered p1!");  
},  
function(){  
    alert("Bye! You now leave p1!");  
});
```

Handling Click Events

- `.click(handler([eventObject]))` is used to listen for a click event or trigger a click event on an element
 - `$('#submitButton').click(function() { alert('Clicked') });`

```
<script>
$(document).ready(function(){
    $('#SubmitButton').click(function(){
        var fName = $('#txtFirstName').val();
        var lName = $('#txtLastName').val();
        $('#tgtDiv').html("<b>" +fName+ " " +lName+ "</b>");
    });
});
</script>
</head>
<body>
FirstName : <input id="txtFirstName" type="text" /><br>
LastName : <input id="txtLastName" type="text" /><br>
<input id="SubmitButton" type="submit" value="Submit"/><br>
<div id="tgtDiv" />
</body>
```

FirstName : Anil

LastName : Patil

Submit

Anil Patil

```
<style>
p {
    color: red;
    margin: 5px;
    cursor: pointer;
}
p:hover {background: yellow;}
</style>
</head>
<body>


First Paragraph



Second Paragraph



Yet one more Paragraph


<script>
$( "p" ).click(function() {
    $( this ).slideUp();
});
</script>
</body>
```

First Paragraph
Second Paragraph
Yet one more Paragraph

```
<style type="text/css">
.highlight{background-color:yellow;};
</style>
<script src="..\Scripts\jquery-3.5.1.js"></script>
<script>
    $(document).ready(function() {
        $('#targetDiv').mouseenter(function() {
            toggle(this);
        });
        $('#targetDiv').mouseleave(function() {
            toggle(this);
        });
        $('#targetDiv').mouseup(function(e) {
            $(this).text('X : '+e.pageX+ ' Y :'+e.pageY );
        });
        function toggle(element){
            $(element).toggleClass('highlight');
        }
    });
</script>
</head>
<body>
<div id="targetDiv">Welcome to events</div>
</body>
```

Welcome to events

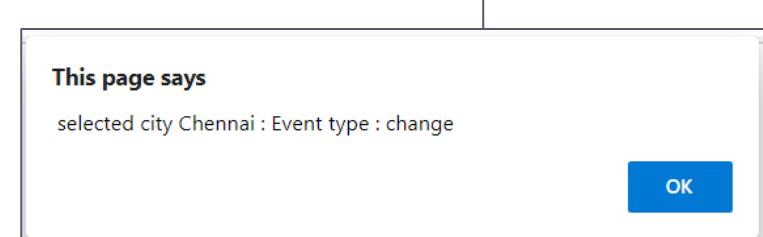
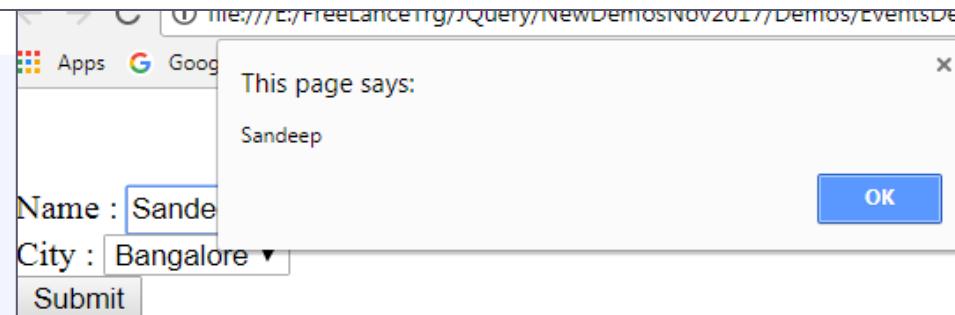
X : 123 Y :17

```

<script>
$(document).ready(function() {
    $('.testClass').change(function() {
        alert($(this).val());
    });
    $('#City').change(disp);

    function disp(e){
        alert(" selected city " + $(this).val() + " : Event type : " + e.type)
    }
});
</script>
</head>
<body><BR><BR>
Name : <input id="txtName" class="testClass" autofocus/>
City :
<select id="City" >
    <option value="Bangalore">Bangalore</option>
    <option value="Chennai" >Chennai</option>
    <option value="Mumbai">Mumbai</option>
</select><br>
<input type="submit" value="Submit"/></td>
</body>

```



Using on() and off()

- The on() method attaches one or more event handlers for the selected elements.

```
$( "#dataTable tbody tr" ).on( "click", function() {
    console.log( $( this ).text() );
});
```

```
<script>
$(document).ready(function() {
    $("#div1").on("click",function(){
        $(this).css("background-color","pink");
    });
    $("h2").on("mouseover mouseout",function(){
        $(this).toggleClass("intro");
    });
    $("#div2").on({
        mouseover: function(){
            $(this).css("background-color", "lightgray");
        },
        mouseout: function(){
            $(this).css("background-color", "lightblue");
        },
        click: function(){
            $(this).hide();
        }
    });
});
</script>
```

Header-2

Text within h4 element

Text within para

This is Div-2

Header-2

Text within h4 element

Text within para

```
<body>
<h2>Header-2</h2>
<div id="div1">
    <h4>Text within h4 element</h4>
    <p>Text within para</p>
</div>
<div id="div2"> This is Div-2 </div>
</body>
```

ANIMATIONS

Showing and Hiding Elements

- To set a duration and a callback function
 - `show(duration, callback)`
 - duration is the amount of time taken (in milliseconds), and callback is a callback function jQuery will call when the transition is complete.
- The corresponding version of `hide()`
 - `hide(duration, callback)`
- To toggle an element from visible to invisible or the other way around with a specific speed and a callback function, use this form of `toggle()`
 - `toggle(duration, callback)`

```

<style type="text/css">
.blueDiv{
    background-color:blue;
    color:white;font-size:30pt;
    display:none
}
</style>
<script src="..\Scripts\jquery-3.5.1.js"></script>
<script>
$(document).ready(function(){
    $('#btnShow').click(function(){
        //fast(200),normal(400) and slow(600) can also be used
        $('.blueDiv').show(5000,function(){
            $('#results').text('Show Animation Completed');
        });
    });
    $('#btnHide').click(function(){
        $('.blueDiv').hide(5000,function(){
            $('#results').text('Hide Animation Completed');
        });
    });
    $('#btnSH').click(function(){
        $('.blueDiv').toggle(5000,function(){
            $('#results').text('Animation Completed');
        });
    });
});
</script>

```



```

<body>
<input id="btnShow" type="button" value="Show"/>
<input id="btnHide" type="button" value="Hide"/>
<input id="btnSH" type="button" value="Show/Hide"/>
<div class="blueDiv">
<p>Welcome to animations!!</p>
</div>
<div id="results"></div>
</body>

```

jQuery Sliding Effects

- The jQuery slide methods slide elements up and down.
- jQuery has the following slide methods:
 - `$(selector).slideDown(speed,callback)` : *Display the matched elements with a sliding motion*
 - `$(selector).slideUp(speed,callback)` : *Hide the matched elements with a sliding motion.*
 - `$(selector).slideToggle(speed,callback)`
- The speed parameter can take the following values: "slow", "fast", "normal", or milliseconds.
- The callback parameter is the name of a function to be executed after the function completes.

```
$("#flip").click(function(){  
    $("#panel").slideDown();  
});
```

jQuery Fading Effects

- The jQuery fade methods gradually change the opacity for selected elements.
- jQuery has the following fade methods:
 - `$(selector).fadeIn(speed,callback)`
 - `$(selector).fadeOut(speed,callback)`
 - `$(selector).fadeTo(speed,opacity,callback)`
- The speed parameter can take the following values: "slow", "fast", "normal", or milliseconds.
 - The opacity parameter in the `fadeTo()` method allows fading to a given opacity.
 - The callback parameter is the name of a function to be executed after the function completes.

```
$('.blueDiv').fadeTo(1000,0.1,function(){
    $('#results').text('FadeTo Animation Completed');
});
```

Fade out Fade In Fade To

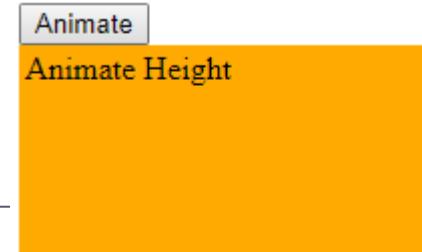
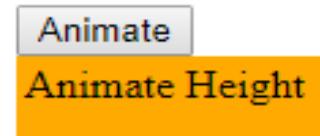
Welcome to jQuery

FadeTo Animation Completed

Creating Custom Animation

- Custom animation can be created in jQuery with the `animate()` function
 - `animate(params, duration, callback)`
 - `params` contains the properties of the object you're animating, such as CSS properties, `duration` is the optional time in milliseconds that the animation should take and `callback` is an optional callback function.

```
<style type="text/css">
#content { background-color:#ffaa00; width:300px; height:30px; padding:3px; }
</style>
<script type="text/javascript">
$(document).ready(function() {
    $("#animate").click(function() {
        $("#content").animate({"height": "100px", "width": "350px"}, "slow");
    });
});
</script>
</head>
<body>
<input type="button" id="animate" value="Animate"/>
<div id="content">Animate Height</div> </body>
```



Jquery plugins

- A jQuery plugin is a method that we use to extend jquery's prototype object.
 - It is a piece of code written in a JavaScript file that enables all jQuery objects to inherit any methods that need to be added.
- Plugin is used to work with a collection of elements.
 - In fact, we can consider each method that comes with jQuery core as a plugin, eg .css(), .hide(), .fadeout(), .addClass() etc
- There are a large number of ready to use plugins available which you can download from repository link at <https://plugins.jquery.com/tag/jquery/>
 - This is official jQuery plug-in repository, where jQuery plug-in developers submit their plug-ins.
 - We can look for the best plug-ins from this repository.
- We can create our own custom plugins; jQuery comes with all the necessary hooks to build plugins easily
 - A plug-in is piece of code written in a standard JavaScript file.
 - These files provide useful jQuery methods which can be used along with jQuery library methods.

The screenshot shows the main navigation bar at the top with links for jQuery, Plugins, UI, Meetups, Forum, Blog, About, and Donate. Below the navigation is the jQuery logo and a search bar with options for Popular Plugins, Latest Releases, Browse Categories, and All Plugins.

The main content area has a dark header with "PLUGINS" and a search bar labeled "Search Plugins". The main content area displays a list of plugins under the heading "PLUGINS: AJAX". It includes a search filter for Project release API compatibility, a "Categories" link, and a detailed description of the "AJAX-ZOOM :: IMAGE ZOOM & PAN GALLERY 2" plugin.

SEARCH

Search for:

USER LOGIN

[Login/Register](#)

BROWSE PLUGINS

[All Plugins](#) [Latest Releases](#)

[Home](#) > [Downloads](#)

PLUGINS: AJAX

Filter by Project release API compatibility: [Login](#) or [register](#) to modify the filter.

[Categories](#)

AJAX-ZOOM :: IMAGE ZOOM & PAN GALLERY 2

AJAX-ZOOM is a powerful [image zoom & pan gallery](#) plugin based on jQuery and PHP. With over 250 options it is very flexible and can be integrated into any website.

AJAX-ZOOM uses image tiles to quickly generate a portion of an image. It also has the ability to watermark zoomed portions on the fly thus adding protection to the source image. The program supports Gdlib and ImageMagick and can be used as gallery script or stand alone image zoom interface. If you want to present high resolution images on the web, AJAX-ZOOM is a perfect tool for you.

jQuery Ajax features

- jQuery provides several functions that can be used to send and receive data

| Method | Description |
|---------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>\$.ajax()</code> | Performs an async AJAX request |
| <code>\$.ajaxPrefilter()</code> | Handle custom Ajax options or modify existing options before each request is sent and before they are processed by <code>\$.ajax()</code> |
| <code>\$.ajaxSetup()</code> | Sets the default values for future AJAX requests |
| <code>\$.ajaxTransport()</code> | Creates an object that handles the actual transmission of Ajax data |
| <code>\$.get()</code> | Loads data from a server using an AJAX HTTP GET request |
| <code>\$.getJSON()</code> | Loads JSON-encoded data from a server using a HTTP GET request |
| <code>\$.parseJSON()</code> | Deprecated in version 3.0, use JSON.parse() instead. Takes a well-formed JSON string and returns the resulting JavaScript value |
| <code>\$.getScript()</code> | Loads (and executes) a JavaScript from a server using an AJAX HTTP GET request |
| <code>\$.param()</code> | Creates a serialized representation of an array or object (can be used as URL query string for AJAX requests) |
| <code>\$.post()</code> | Loads data from a server using an AJAX HTTP POST request |

Using get()

- `$.get(url [, data] [, success] [, dataType])` : retrieve data from a server using GET
 - datatype can be html, xml, json

```
<!-- AjaxGet.html -->
<html>
<head>
<title>Hello there</title>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.4/jquery.min.js">
<script>
$(document).ready(function() {
    $('#btn1').click(function(){
        $.get('/sample.txt',function(data){
            $('#targetDiv').html(data);
        });
    });
});
</script>
</head>
<body>
<h3>Welcome to Node!
<input id="btn1" type="button" value="get!" />
<div id="targetDiv"/>
</body>
</html>
```

Welcome to Node!

```
sample.txt
1 This is sample txt
2 Sending some response to
3 our AJAX application
```

```
//First.js
var express = require('express');
var app = express()

app.get("/sample.txt",function(req, resp){
    resp.sendFile("sample.txt", { root : __dirname})
})

app.get("/",function(req, resp){
    resp.sendFile("AjaxGet.html", { root : __dirname})
})
app.listen(3000, function(){
    console.log("server started at http://localhost:3000")
})
```

MYEXPRESSAPP

- node_modules
- AjaxGet.html
- First.js
- package-lock.json
- package.json
- sample.txt

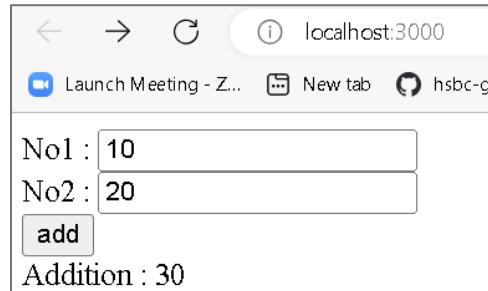
Welcome to Node!

This is sample txt Sending some response to our AJAX application

```

<!-- AjaxGet.html -->
<html>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.
</script>
$(document).ready(function() {
    $('#btn1').click(function(){
        $.get('/add',
            { no1 : $('#n1').val(), no2: $('#n2').val() },
            function(data){
                $('#targetDiv').html(data);
            });
    });
});
</script>
</head>
<body>
    No1 : <input id="n1"><br>
    No2 : <input id="n2"><br>
    <input type="button" id="btn1" value="add">
<div id="targetDiv"/>
</body>
</html>

```



`$.get(url [, data] [, success] [, dataType])`

```

//adder.js
var express = require('express');
var app = express()

app.get("/",function(req, resp){
    resp.sendFile("AjaxGet1.html", { root : __dirname})
})

app.get('/add', function (req, res) {
    let n1 = parseInt(req.query.no1);
    let n2 = parseInt(req.query.no2);
    res.send("Addition : " + (n1 + n2));
});

app.listen(3000, function(){
    console.log("server started at http://localhost:3000")
})

```

Using post()

- `$.post(url [, data] [, success] [, dataType])`: Send data to the server using a HTTP POST request.

```
<!-- AjaxPost.html -->
<html>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/jque
<script>
$(document).ready(function() {
    $('#btn1').click(function(){
        $.post('/register',
            { username : $('#uname').val(), email: $('#email').val() },
            function(data){
                $('#targetDiv').html(data + "Successfully registered");
            });
    });
});
</script>
</head>
<body>
    Name : <input id="uname"><br>
    Email : <input id="email"><br>
    <input type="submit" id="btn1" value="Register">
    <div id="targetDiv"/>
</body>
</html>
```

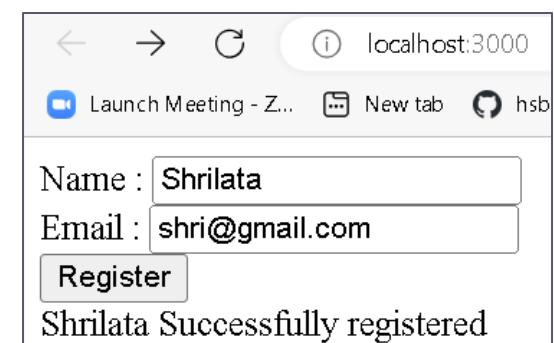
```
//Register.js
var express = require('express');
var app = express()

var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: false }));

app.get("/",function(req, resp){
    resp.sendFile("AjaxPost.html", { root : __dirname})
})

app.post('/register', function (req, res) {
    let username = req.body.username ;
    let email = req.body.email ;
    //registering user....
    res.send(username);
});

app.listen(3000, function(){
    console.log("server started at http://localhost:3000")
})
```



Getting JSON data

- `$.getJSON(url [, data] [, success])` can retrieve data from a server

```
<!-- AjaxJson.html -->
<html>
<head>
<title>Hello there</title>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.4/">
<script>
$(document).ready(function() {
    $('#btn1').click(function(){
        $.getJSON('/empData',function(data){
            $('#outputDiv').html('<p> Name: ' + data.empname + '</p>');
            $('#outputDiv').append('<p>Emp ID : ' + data.empid+ '</p>');
            $('#outputDiv').append('<p> Department: ' + data.dept+ '</p>');
        });
    });
});
</script>
</head>
<body>
<input id="btn1" type="button" value="Get Data" />
<div id="outputDiv"/>
</body>
</html>
```

The screenshot shows a browser window at localhost:3000. A button labeled "Get Data" has been clicked. The page displays three pieces of information: "Name: Shrilata", "Emp ID : 12345", and "Department: Training". These correspond to the data retrieved from the JSON file.

```
emp.json > ...
1  {
2      "empname": "Shrilata",
3      "empid" : 12345,
4      "dept": "Training"
5 }
```

localhost:3000

Get Data

Name: Shrilata

Emp ID : 12345

Department: Training

```
//FetchJson.js
var express = require('express');
var app = express()

app.get("/empData",function(req, resp){
    resp.sendFile("emp.json", { root : __dirname})
})

app.get("/",function(req, resp){
    resp.sendFile("AjaxJson.html", { root : __dirname})
})
app.listen(3000, function(){
    console.log("server started at http://localhost:3000")
})
```

Using ajax() function

- ajax() function is configured by assigning values to JSON properties

```
<!-- Ajax.html -->
<html>
<head>
<title>Hello there</title>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
<script>
$(document).ready(function() {
    $('#btn1').click(function(){
        $.ajax({
            url: "sampleData",
            success: function(result){
                $("#targetDiv").html(result);
            }
        });
    });
});
</script>
</head>
<body>
<h3>Welcome to Node!
<input id="btn1" type="button" value="Get Data!"/>
<div id="targetDiv"/>
</body>
</html>
```



```
sample.txt
1 This is sample txt
2 Sending some response to
3 our AJAX application
```

```
//MainAjax.js
var express = require('express');
var app = express()

app.get("/sampleData",function(req, resp){
    resp.sendFile("sample.txt", { root : __dirname})
})

app.get("/",function(req, resp){
    resp.sendFile("Ajax.html", { root : __dirname})
})
app.listen(3000, function(){
    console.log("server started at http://localhost:3000")
})
```

```

<!-- Ajax1.html -->
<html>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.4/"></script>
<script>
$(document).ready(function() {
    $('#btn1').click(function(){
        $.ajax({
            type : "GET",
            url: "personData",
            datatype : "json", // type of data expected back from the server
            success : function(data, status, xhr) { //data contains entire response
                var str = "<table border='1'>";
                str += "<tr><th>Person name</th><th>Age</th></tr>";
                for(p of data){
                    str += "<tr><td>" + p.pname + "</td><td>" + p.age + "</td></tr>";
                }
                str += "</table>";
                $("#targetDiv").html(str);
            },
            error : function(xhr, status, error) {
                $("#targetDiv").html(xhr.statusText);
            }
        });
    });
}

<body>
<input id="btn1" type="button" value="Person Data" />
<div id="targetDiv"/>
</body>
</html>

```

person.json > ...

```

1 [ {"pname": "Anita", "age": 55},
2   {"pname": "Kavita", "age": 34},
3   {"pname": "Sunita", "age": 21}
4 ]
5 ]

```

localhost:3000

| Person Data | |
|-------------|-----|
| Person name | Age |
| Anita | 55 |
| Kavita | 34 |
| Sunita | 21 |

```

//MainAjax1.js
var express = require('express');
var app = express()

app.get("/personData",function(req, resp){
    resp.sendFile("person.json", { root : __dirname})
})

app.get("/",function(req, resp){
    resp.sendFile("Ajax1.html", { root : __dirname})
})
app.listen(3000, function(){
    console.log("server started at http://localhost:3000")
})

```

```
jQuery.ajax({
```

```
    type: "GET|POST|DELETE|PUT",
```

Request Type

```
    url: url,
```

Page URL

```
    data: '{param:"value"}',
```

Data to be sent in the body of the request.

```
    dataType:"text|html|json|jsonp|script|xml",
```

Specifies the type of data expected in the response and how that data should be processed by jQuery.

```
    complete: function(){.....},
```

```
    success: success_callback,
```

```
    error: error_callback
```

```
});
```

function to be called when the request finishes (after success and error callbacks are executed)

Callback function representing success

Loading HTML content from server

- `$(selector).load(url,data,callback)` allows HTML content to be loaded from a server and added into DOM object.
 - `$("#targetDiv").load('GetContents.html');`
- A selector can be added after the URL to filter the content that is returned from the calling load().
 - `$("#targetDiv").load('GetContents.html #Main');`
- Data can be passed to the server using `load(url,data)`
 - `$('#targetDiv').load('Add.aspx',{firstNumber:5,secondNumber:10})`
- `load ()` can be passed a callback function

```
$('#targetDiv').load('Notfound.html', function (res,status,xhr) {  
    if (status == "error") {  
        alert(xhr.statusText);  
    }  
});
```

Using get(), getJSON() & post()

- `$.get(url,data,callback,datatype)` can retrieve data from a server.
 - datatype can be html, xml, json

```
$.get('GetContents.html',function(data){  
    $('#targetDiv').html(data);  
},'html');
```

- `$.getJSON(url,data,callback)` can retrieve data from a server.

```
$.getJSON('GetContents.aspx,{id:5},function(data){  
    $('#targetDiv').html(data);  
});
```

- `$.post(url,data,callback,datatype)` can post data to a server and retrieve results.

Using ajax() function

- ajax() function is configured by assigning values to JSON properties

```
$.ajax({  
    url: "employee.asmx/GetEmployees",  
    data : null,  
    contentType: "application/json; charset=utf-8",  
    datatype: 'json',  
    success: function(data,status,xhr){  
        //Perform success operation  
    },  
    error: function(xhr,status,error) {  
        //show error details  
    }  
});
```

```
$("button").click(function() {  
    $.ajax({  
        url: "demo_test.txt",  
        success: function(result){  
            $("#div1").html(result);  
        }  
    });  
});
```

BOOTSTRAP 5

Pre-reqs:

- HTML
- CSS
- JavaScript

Introduction

- Bootstrap is an opensource frontend framework developed by Twitter.
 - It is the most popular ***HTML, CSS, and JavaScript*** framework for developing responsive, mobile first web sites.
 - Bootstrap is a free and open source collection of tools for creating websites and web applications.
 - Bootstrap contains a set of CSS- and HTML-based templates for styling forms, elements, buttons, navigation, typography, and a range of other UI components.
 - It also comes with optional JavaScript plugins to add interactivity to components.
- Bootstrap is promoted as being **One framework, every device**.
 - This is because websites built with Bootstrap will automatically scale between devices — whether the device is a mobile phone, tablet, laptop, desktop computer, screen reader, etc.
- Responsive web design is about creating web sites which automatically adjust themselves to look good on all devices, from small phones to large desktops.
 - Developers can then create a single design that works on any kind of device: mobiles, tablets, smart TVs, and PCs

Where to Get Bootstrap 5?

- There are three ways to start using Bootstrap 5 on your own web site.
 - Download Bootstrap 5 from getbootstrap.com : <https://getbootstrap.com/docs/5.0/getting-started/download/>
 - Using CDN (Content Delivery Network).
 - The <https://getbootstrap.com/docs/5.0/getting-started/introduction/> page gives CDN links for CSS and js files
 - Install bootstrap via package managers like NPM or YARN

```
<link  
  href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css"  
  rel="stylesheet"  
  integrity="sha384-EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTWspd3yD65VohhpuuCOmLASjC"  
  crossorigin="anonymous">  
  
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/js/bootstrap.bundle.min.js" integrity="sha384-MrcW6ZMFYlzcLA8NL+NtUVF0sA7MsXsP1UyJoMp4YLEuNSfAP+JcXn/tWtlaxVXM"  
  crossorigin="anonymous">  
</script>
```

The attributes `integrity` and `crossorigin` have been added to CDN links to implement Subresource Integrity (SRI). It is a security feature that enables you to mitigate the risk of attacks originating from compromised CDNs, by ensuring that the files your website fetches (from a CDN or anywhere) have been delivered without unexpected or malicious modifications. It works by allowing you to provide a cryptographic hash that a fetched file must match.

Create First Web Page With Bootstrap 5

- Bootstrap 5 requires a containing element to wrap site contents.
 - There are two container classes to choose from:
 - The `.container` class provides a responsive fixed width container
 - The `.container-fluid` class provides a full width container, spanning the entire width of the viewport

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Bootstrap Example</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css" rel="stylesheet">
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/js/bootstrap.min.js">
</head>
<body>
  <div class="container-fluid">
    <h1>My First Bootstrap Page</h1>
    <p>This part is inside a container class.</p>
  </div>
</body>
</html>
```

file | E:/FreelanceTrg/Bootstrap/BootStrap4/Demo/Intro/first.html

My First Bootstrap Page

This is some text.

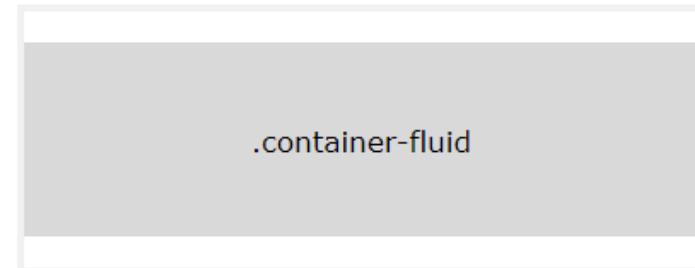
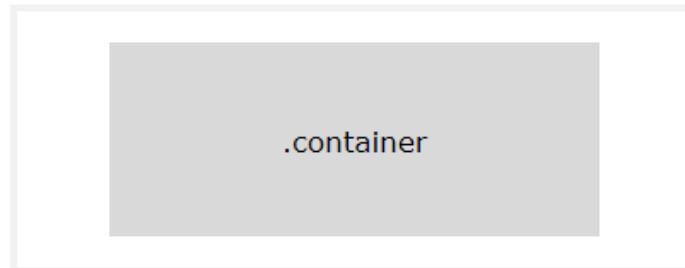
To ensure proper rendering and touch zooming, add this `<meta>` tag

- The `width=device-width` part sets the width of the page to follow the screen-width of the device (which will vary depending on the device).
- The `initial-scale=1` part sets the initial zoom level when the page is first loaded by the browser.

Bootstrap Container

- Bootstrap container is basically used in order to create a centered area that lies within the page and generally deals with the margin of the content and the behaviors that are responsible for the layout.
 - It contains the grid system (row elements, which in turn are the container of columns).
- There are two container classes in Bootstrap:
 - `.container`: provides a fixed width container with responsiveness. It will not take the complete width of its viewport.
 - `.container-fluid`: provides a full width container of the viewport and its width will change (expand or shrink) on different screen sizes.

```
<body>
  <div class="container">
    <h1>Container</h1>
  </div>
</body>
```



Bootstrap Grid System

- Bootstrap **grid system** allows to properly house the website's content.
 - Grid system divides the screen into columns—up to 12 in each row. (rows are infinite) ; that can be used to create various types of layouts.
 - The column widths vary according to the size of screen they're displayed in.
 - Hence, Bootstrap's grid system is **responsive**, as the columns resize themselves dynamically when the size of browser window changes.
 - If you do not want to use all 12 columns individually, you can group the columns together to create wider columns

| | | | | | | | | | | | | | | | |
|---------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--|--|--|--|
| span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | | | | |
| span 4 | | | | span 4 | | | | span 4 | | | | | | | |
| span 4 | | | | span 8 | | | | | | | | | | | |
| span 6 | | | | | | span 6 | | | | | | | | | |
| span 12 | | | | | | | | | | | | | | | |

If the sum of the cols in your row doesn't get to 12, then they don't fill the whole row. And if it goes beyond 12 then it will move to the next line (it will only display the sum of the first elements <=12 on the first line).

Building a Basic Grid

- Recommended: place all the rows and columns inside a container to ensure proper alignment and padding.
 - Container creates a fixed width container in the browser window, while container-fluid creates a full-width fluid container.
 - Hence, it is a good practice to wrap all the contents within a container.
 - To create a container in our HTML page: `<div class="container">...</div>`
 - Next, create a row inside a container, then start creating the columns.
 - Bootstrap has a class `row` for creating rows:

```
<div class="container">
  <div class="row">
    //add desired number of cols here
  </div>
</div>
```

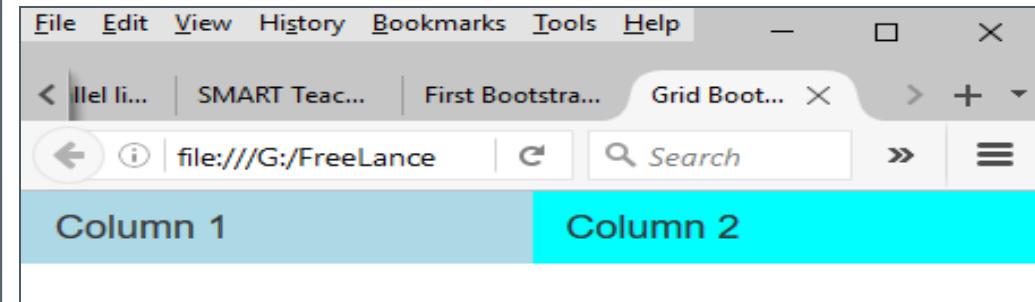
Grid Classes

- The Bootstrap 5 grid system has 6 classes:
 - .col-xs (extra small devices - screen width less than 576px)
 - .col-sm- (small devices - screen width equal to or greater than 576px)
 - .col-md- (medium devices - screen width equal to or greater than 768px)
 - .col-lg- (large devices - screen width equal to or greater than 992px)
 - .col-xl- (xlarge devices - screen width equal to or greater than 1200px)
 - .col-xxl(Extra extra large- screen width equal to or greater than 1400px)

```
<style>
.mycol1{ background: lightblue;}
.mycol2{ background: cyan;}
</style>
</head>

<body>
<div class="container-fluid">
<div class="row">
<div class="col mycol1">
<h4>Column 1</h4>
</div>
<div class="col mycol2">
<h4>Column 2</h4>
</div>
</div>
</div>
</body>
```

Example



Building a Basic Grid

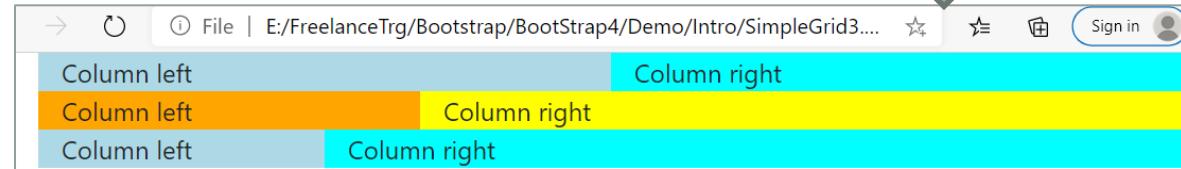
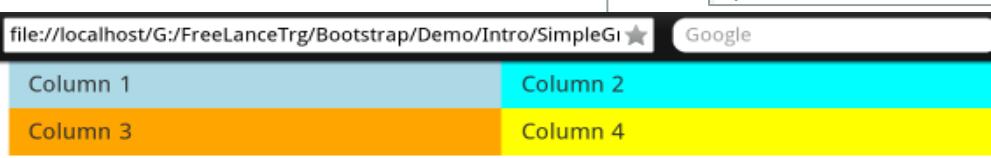
```
<style>
    .col1{ background: lightblue;}
    .col2{ background: cyan;}
    .col3{ background: orange;}
    .col4{ background: yellow;}
</style>

</head>
<body>
    <div class="container">
        <div class="row">
            <div class="col-12 col-sm-6 col1">
                <h4>Column 1</h4>
            </div>
            <div class="col-12 col-sm-6 col2">
                <h4>Column 2</h4>
            </div>
            <div class="col-12 col-sm-6 col3">
                <h4>Column 3</h4>
            </div>
            <div class="col-12 col-sm-6 col4">
                <h4>Column 4</h4>
            </div>
        </div>
    </div>
</body>
```

```
<style>
    .col1{ background: lightblue;}
    .col2{ background: cyan;}
    .col3{ background: orange;}
    .col4{ background: yellow;}
</style>
</head>
<body>
    <div class="container">
        <!--Row with two equal columns-->
        <div class="row">
            <div class="col-md-6 col1">Column left</div>
            <div class="col-md-6 col2">Column right</div>
        </div>

        <!--Row with two columns divided in 1:2 ratio-->
        <div class="row">
            <div class="col-md-4 col3">Column left</div>
            <div class="col-md-8 col4">Column right</div>
        </div>

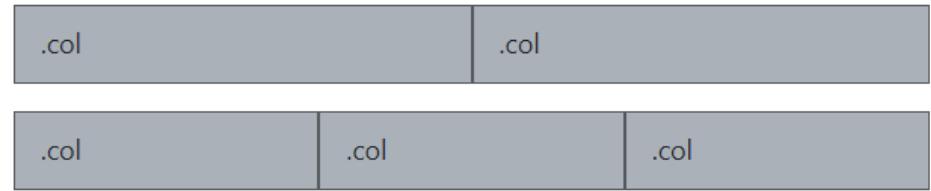
        <!--Row with two columns divided in 1:3 ratio-->
        <div class="row">
            <div class="col-md-3 col1">Column left</div>
            <div class="col-md-9 col2">Column right</div>
        </div>
    </div>
</body>
```



Bootstrap Auto-layout Columns

- You can create equal width columns for all devices (xs, sm, md, lg, and xl) through simply using the class .col, without specifying any column number.

```
<body>
  <div class="container">
    <!--Row with two equal columns-->
    <div class="row">
      <div class="col">col-1</div>
      <div class="col">col-2</div>
    </div>
  </div>
    <!--Row with three equal columns-->
    <div class="row">
      <div class="col">col-3</div>
      <div class="col">col-4</div>
      <div class="col">col-5</div>
    </div>
  </div>
</body>
```



MISC COMPONENTS

Bootstrap Typography

- Typography ? The style and appearance of printed matter.
 - Typography refers to the various styles present in Bootstrap style sheets which define how various text elements will appear on the web page.
 - It shows how certain text elements are rendered when we use Bootstrap without including the style classes.
 - Typography **includes colors, fonts, headings, text alignment, background color, etc**
- HTML uses default font and style to create headings, paragraphs, lists and other inline elements.
 - Bootstrap overrides default and provides consistent styling across browsers for common typographic elements.
 - Eg, Bootstrap provides its own style for all six standard heading levels
 - Bootstrap 5 uses a default font-size of 16px, and its line-height is 1.5.

My First Bootstrap Page

This is some text.

My First Bootstrap Page

This is some text.

Typography

- Bootstrap 5 styles HTML headings (`<h1>` to `<h6>`) with a bolder font-weight and an increased font-size

```
<h1>h1. Bootstrap heading</h1>
<h2>h2. Bootstrap heading</h2>
<h3>h3. Bootstrap heading</h3>
<h4>h4. Bootstrap heading</h4>
<h5>h5. Bootstrap heading</h5>
<h6>h6. Bootstrap heading</h6>
```

h1. Bootstrap heading
h2. Bootstrap heading
h3. Bootstrap heading
h4. Bootstrap heading
h5. Bootstrap heading
h6. Bootstrap heading

- You can make a content stand out by adding one of several classes.
- You can also transform the text to lowercase, uppercase or make them capitalize.

```
<p>This is how a normal paragraph looks like in Bootstrap.</p>
<p class="lead">This is how a paragraph stands out in Bootstrap.</p>
<p class="text-start">Left aligned text.</p>
<p class="text-center">Center aligned text.</p>
<p class="text-end">Right aligned text.</p>
<p class="text-lowercase">Text in lowercase</p>
<p class="text-uppercase">Text in uppercase</p>
<p class="text-capitalize">Text in capitalize</p>
```

This is how a normal paragraph looks like in Bootstrap.

This is how a paragraph stands out in Bootstrap.

Left aligned text.

Center aligned text.

text in lowercase

TEXT IN UPPERCASE

Text In Capitalize

Right aligned text.

Working with content : Text Coloring

- Colors are the powerful method of conveying important information in website design.
- Bootstrap has handful of emphasis utility classes that can be used for this purpose such as showing success message in green color, warning or error message in red color, etc.

```
<p class="text-muted">This text is muted.</p>
<p class="text-primary">This text is important.</p>
<p class="text-secondary">This text is secondary.</p>
<p class="text-success">This text indicates success.</p>
<p class="text-info">This text represents some information.</p>
<p class="text-warning">This text represents a warning.</p>
<p class="text-danger">This text represents danger.</p>
```

```
<p class="bg-primary">This text is important.</p>
<p class="bg-success">This text indicates success.</p>
<p class="bg-info">This text represents some information.</p>
<p class="bg-warning">This text represents a warning.</p>
<p class="bg-danger">This text represents danger.</p>
```

This text is muted.
This text is important.
This text is secondary.
This text indicates success.
This text represents some information.
This text represents a warning.
This text represents danger.

This text is important.
This text indicates success.
This text represents some information.
This text represents a warning.
This text represents danger.

Tables

- Bootstrap provides an efficient layout to build elegant tables
 - You can create tables with basic styling that has horizontal dividers and small cell padding, by just adding the Bootstrap's class `.table` to the `<table>` element.

```
<table class="table">
  <tr><th>Name</th><th>Age</th></tr>
  <tr><td>Kavita</td><td>23</td></tr>
  <tr><td>Anita</td><td>33</td></tr>
</table>
```

| Name | Age |
|--------|-----|
| Kavita | 23 |
| Anita | 33 |

- The `.table-striped` class adds zebra-stripes to a table
- The `.table-bordered` class adds borders on all sides of the table and cells
- The `.table-sm` class makes a table more compact by cutting cell padding in half
- The `.table-dark` class creates inverted version of this table, i.e. table with light text on dark backgrounds

```
<table class="table table-dark">
  <tr><th>Name</th><th>Age</th></tr>
  <tr><td>Kavita</td><td>23</td></tr>
  <tr><td>Anita</td><td>33</td></tr>
</table>
```

| Name | Age |
|--------|-----|
| Kavita | 23 |
| Anita | 33 |

Tables

```
<table class = "table table-striped table-bordered table-sm">
  <caption>Basic Table Layout</caption>
  <thead class="thead-light">
    <tr><th>Name</th><th>City</th></tr>
  </thead>
  <tbody>
    <tr><td>Soha</td><td>Bangalore</td></tr>
    <tr><td>Shrilata</td><td>Pune</td></tr>
    <tr><td>Sandeep</td><td>Mumbai</td></tr>
    <tr class="table-success">
      <td>Sheela</td>
      <td>Delhi</td>
    </tr>
  </tbody>
</table>
```

| Name | City |
|----------|-----------|
| Soha | Bangalore |
| Shrilata | Pune |
| Sandeep | Mumbai |
| Sheela | Delhi |

```
<table class="table table-bordered table-sm">
  <thead class="thead-dark">
    <tr>
      <th>Month</th><th>Savings</th>
    </tr>
  </thead>
  <tbody>
    <tr><td>January</td><td>$100</td></tr>
    <tr><td>February</td><td>$80</td></tr>
  </tbody>
  <tfoot>
    <tr><td>Sum</td><td>$180</td></tr>
  </tfoot>
</table>
```

| Month | Savings |
|----------|---------|
| January | \$100 |
| February | \$80 |
| Sum | \$180 |

Jumbotron

- A jumbotron indicates a big box for calling extra attention to some special content or information.
 - Jumbotrons are no longer supported in Bootstrap 5. However, you can use a `<div>` element and add special helper classes together with a color class to achieve the same effect
 - Tip: Inside a jumbotron you can put nearly any valid HTML, including other Bootstrap elements/classes.

```
<div class="mt-4 p-5 bg-info text-white rounded">
  <h1>Bootstrap Tutorial</h1>
  <p>Bootstrap is the most popular HTML, CSS, and JS framework for developing
     responsive, mobile-first projects on the web.</p>
</div>
```

Bootstrap Tutorial

Bootstrap is the most popular HTML, CSS, and JS framework for developing responsive, mobile-first projects on the web.

images

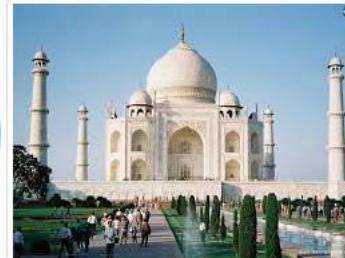
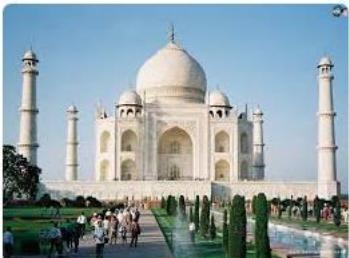
- To add images on the webpage use element `` , it has **three** classes to apply simple styles to images.
 - `.rounded`: This adds rounded corners to our image.
 - `.rounded-circle`: This turns our images into a circle.
 - `.img-thumbnail`: This adds borders to our image thus shaping it into a thumbnail.
- Alignment : These classes decide the floating attribute of images:
 - `.float-end`: This floats our image to the right.
 - `.float-start`: This floats it to the left.
 - Center an image by adding the utility classes `.mx-auto` (`margin:auto`) and `.d-block` (`display:block`) to the image

```




<p>... </p>
```

Images are essential part of a website because they convey a lot of information and make websites attractive. But adding images and making them responsive can be a tiring task using CSS. Bootstrap 4 comes with predefined images classes for making our responsive. The `img` tag is used to include an image in our website. We can style this tag with the Bootstrap 4 classes that are categorized in terms of the functions



BOOTSTRAP ICONS

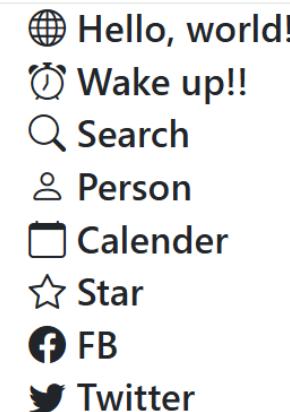
Bootstrap Icons

- Bootstrap 5 has released a new SVG icon library which was undertaken by the co-founder of Bootstrap Mark Otto.
<https://icons.getbootstrap.com/>
- Bootstrap Icons are SVGs, so you can include them into your HTML in a few ways depending on how your project is setup.
- The simplest way to include Bootstrap icons in a web page is using the CDN link.
 - This link points to a CSS file that has all the necessary classes to generate font icons.

```
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.5.0/font/bootstrap-icons.css">
```

- To use Bootstrap icons in your code you'll require an `<i>` tag with an individual icon class `.bi-*` applied on it. The general syntax for using Bootstrap icons is:
- `<i class="bi-class-name"></i>` Where class-name is the name of the icon class, e.g. search, person

```
<h1><i class="bi-globe"></i> Hello, world!</h1>
<h1><i class="bi-alarm"></i> Wake up!! </h1>
<h1><i class="bi-search"></i> Search</h1>
<h1><i class="bi-person"></i> Person</h1>
<h1><i class="bi-calendar"></i> Calender</h1>
<h1><i class="bi-star"></i> Star</h1>
<h1><i class="bi-facebook"></i> FB</h1>
<h1><i class="bi-twitter"></i> Twitter</h1>
```



Using Icons in Bootstrap 5

- Placing icons inside buttons:

```
<button type="submit" class="btn btn-primary">
  <span class="bi-search"></span> Search
</button>
<button type="submit" class="btn btn-warning">
  <span class="bi-save"></span> Save
</button>
```



- Similarly, you can place icons inside the navs, forms, tables, paragraphs or anywhere you want

Alerts

- Bootstrap comes with a very useful component for displaying alert messages in various sections of our website
 - You can use them for displaying a success message, a warning message etc
 - These messages can be annoying to visitors, hence they should have dismiss functionality added to give visitors the ability to hide them.
 - Alerts are created with the `.alert` class, followed by one of the contextual classes like `.alert-success`, `.alert-info`

```
<div class="alert alert-warning">  
    Amount has been transferred successfully.  
</div>  
<div class="alert alert-success alert-dismissible">  
    <button type="button" class="btn-close" data-bs-dismiss="alert"></button>  
    Amount has been transferred successfully.  
</div>
```

Amount has been transferred successfully.

Amount has been transferred successfully. 

contextual classes for
alert messages:
`alert-success`
`alert-info`
`alert-danger`
`alert-warning`

Standing Out : Buttons

- It's easy to convert an a, button, or input element into a fancy bold button in Bootstrap; just have to add the btn class

```
<a href="#" class="btn btn-primary">Button-1</a>
<button type="button" class="btn btn-primary">Button-2</button>
<input type="button" class="btn btn-info" value="Button-3">
```

Button-1 Button-2 Button-3

- You can also create outline buttons by replacing the button modifier classes

```
<button type="button" class="btn btn-outline-primary">Primary</button>
<button type="button" class="btn btn-outline-warning">Warning</button>
```

Primary Warning

- Buttons come in various color options:

- btn-primary for dark blue
- btn-secondary for dark gray
- btn-success for green
- btn-info for teal blue
- btn-warning for orange
- btn-danger for red
- btn-dark for dark

Basic Primary Secondary Success Info Warning Danger Dark

- And in various sizes:

- o btn-lg for large buttons
- o btn-sm for small buttons

Large Default Small

```
<button type="button" class="btn btn-primary btn-lg">
Large</button>
<button type="button" class="btn btn-primary">
Default</button>
<button type="button" class="btn btn-primary btn-sm
btn-success"> Small</button>
```

Standing Out : Buttons

- Button groups allow multiple buttons to be stacked together
 - To create a button groups just wrap a series of buttons with `.btn` class in a `<div>` element and apply the class `.btn-group` on it

```
<div class="btn-group">
  <button class="btn btn-primary">Save</button>
  <button class="btn btn-primary">Contact Us</button>
  <button class="btn btn-primary">About</button>
</div>
```

Save Contact Us About

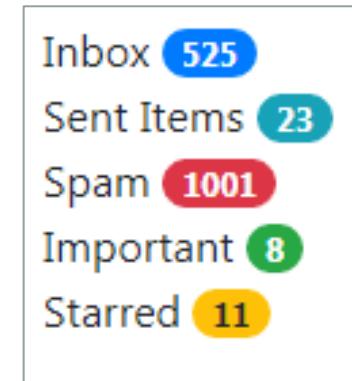
- Use the class `.btn-group-vertical` to create a vertical button group
 - `<div class="btn-group-vertical">`
- Use class `.btn-group-lg` for a large button group or `.btn-group-sm` for a small button group

```
<div class="btn-group btn-group-lg">
  <button class="btn btn-primary">1</button>
  <button class="btn btn-primary">2</button>
  <button class="btn btn-primary">3</button>
</div>
<div class="btn-group btn-group-sm">
  <button class="btn btn-primary">1</button>
  <button class="btn btn-primary">2</button>
  <button class="btn btn-primary">3</button>
</div>
```

1 2 3 1 2 3

Standing Out : badges

- Badges are generally used to indicate some valuable information on the web pages such as important heading, warning messages, notification counter, etc.
 - They are mostly used to display numbers such as unread items, notifications, and so on, rather than text
 - Use the `.badge` class together with a contextual class (like `.bg-secondary`) within `` elements to create badges.
 - Badges are **self-collapsing** components, ie when there's no content inside the badge it will not be visible on the website



```
<a href="#" class="btn btn-primary btn-lg">Inbox
  <span class="badge">23</span>
</a>
```

Inbox 23

```
<span class="badge bg-primary">Primary</span>
<span class="badge bg-secondary">Secondary</span>
<span class="badge bg-success">Success</span>
<span class="badge bg-danger">Danger</span>
```

Primary

Secondary

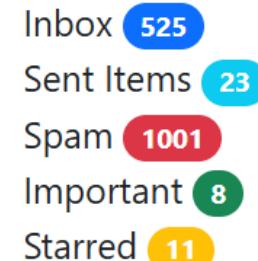
Success

Danger

Standing Out : badges

.rounded-pill: for creating rounded badges

```
<div>Inbox <span class="badge rounded-pill bg-primary">525</span></div>
<div>Sent Items <span class="badge rounded-pill bg-info">23</span></div>
<div>Spam <span class="badge rounded-pill bg-danger">1001</span></div>
<div>Important <span class="badge rounded-pill bg-success">8</span></div>
<div>Starred <span class="badge rounded-pill bg-warning">11</span></div>
```



```
<button type="button" class="btn btn-primary">
    Inbox <span class="badge bg-warning">4</span>
</button>
<button type="button" class="btn btn-info">
    Sent <span class="badge rounded-pill bg-dark">4</span>
</button>
<button type="button" class="btn btn-danger">
    Spam <span class="badge bg-success">4</span>
</button>
```



Progress bar

- A progress bar is a visual representation to show the progress of a task or an operation.
 - Used in many web applications to show the progress of action to users.
 - Bootstrap offers different progress bars, depending upon multiple parameters like color, height, shape, or style.
 - To create a default progress bar, use the `.progress` class with a `.progress-bar` nested inside
 - To create progress bars with different colors and styles, use utility (contextual) classes

```
<div class="progress">
    <div class="progress-bar" style="width:70%"></div>
<div>
```



- You can include a label that displays a percentage value on the progress bar (to explicitly state, in percentage terms, the completeness of the task).
 - To display a label on the progress bar, insert text as contents of the `<div>` element that has the `.progress-bar` class applied.

```
<div class="progress">
    <div class="progress-bar" role="progressbar" style="width:70%">
        <span>70% Complete</span>
    </div>
</div>
```



Progress bar

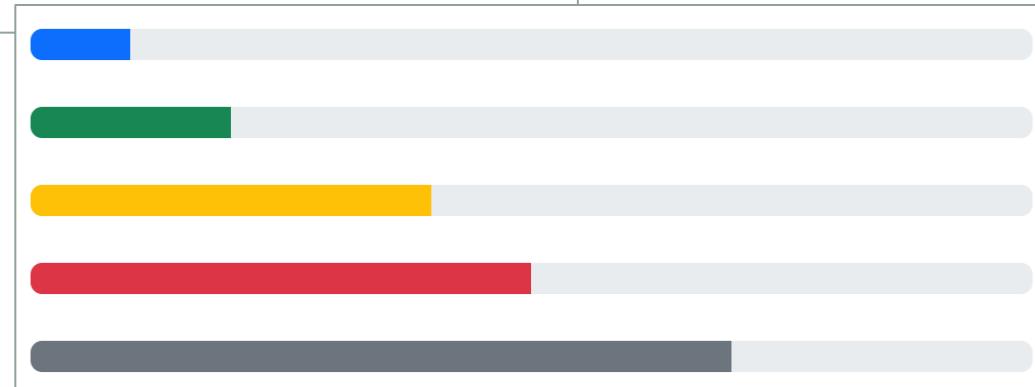
```
<div class="progress"> <!-- Blue -->
  <div class="progress-bar" style="width:10%"></div>
</div>

<div class="progress"> <!-- Green -->
  <div class="progress-bar bg-success" style="width:20%"></div>
</div>

<div class="progress"> <!-- Orange -->
  <div class="progress-bar bg-warning" style="width:40%"></div>
</div>

<div class="progress"> <!-- Red -->
  <div class="progress-bar bg-danger" style="width:50%"></div>
</div>

<div class="progress"> <!-- Grey -->
  <div class="progress-bar bg-secondary" style="width:70%"></div>
</div>
```



Pagination

- Pagination are those numbered links we find either below a webpage or on top of a webpage.
 - They are used to select pages between pages of a website.
- Pagination is used to enable navigation between pages in a website.
 - The pagination used in Bootstrap has a large block of connected links that are hard to miss and are easily scalable.
- To create a basic pagination, add the `.pagination` class to an `` element.
- Then add the `.page-item` to each `` element and a `.page-link` class to each link inside ``
 - The `.pagination` class is used to specify pagination on a list group.
 - The `.page-item` class is used to specify each pagination item in the group.
 - The `.page-link` class is used to specify the link in the pagination item.

```
<div class="container">
  <h2>Pagination</h2>
  <p>The .pagination class provides pagination links:</p>
  <ul class="pagination">
    <li class="page-item"><a href="#" class="page-link">1</a></li>
    <li class="page-item"><a href="#" class="page-link">2</a></li>
    <li class="page-item"><a href="#" class="page-link">3</a></li>
    <li class="page-item"><a href="#" class="page-link">4</a></li>
    <li class="page-item"><a href="#" class="page-link">5</a></li>
  </ul>
</div>
```

Pagination

The `.pagination` class provides pagination links:

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|

Bootstrap card

- A card in bootstrap is a bordered box with some padding around its content
 - It includes options for headers and footers, a wide variety of content, contextual background colors, and powerful display options.
 - If you're familiar with Bootstrap 3, cards replace old panels, wells, and thumbnails.
 - Cards support a wide variety of content, including text, list groups, links, and more within a box with rounded corners
 - The card div also can have optional parts like header, footer etc.
 - A basic card is created with the `.card` class. Content inside the card has a `.card-body` class

```
<h4>Simple card</h4>
<div class="card">
  <div class="card-body">Basic card</div>
</div>
```

Simple card

Basic card

```
<h4>Card with header and footer</h4>
<div class="card">
  <div class="card-header">Header</div>
  <div class="card-body">Content</div>
  <div class="card-footer">Footer</div>
</div>
```

Card with header and footer

Header

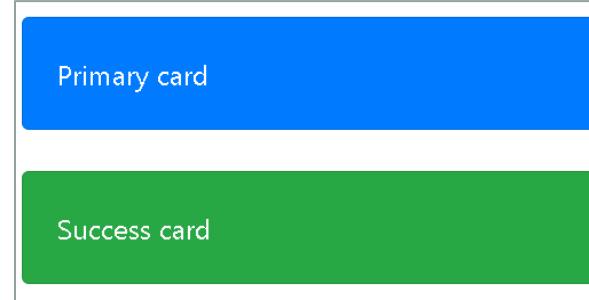
Content

Footer

Bootstrap card

- To add a background color the card, use contextual classes (.bg-primary, .bg-success, .bg-info, .bg-warning, .bg-danger, .bg-secondary, .bg-dark and .bg-light).

```
<div class="card bg-primary text-white">
  <div class="card-body">Primary card</div>
</div>
<br>
<div class="card bg-success text-white">
  <div class="card-body">Success card</div>
</div>
```



- Titles, text, and links
 - Use `.card-title` to add card titles to any heading element.
 - The `.card-text` class is used to remove bottom margins for a `<p>` element if it is the last child (or the only one) inside `.card-body`.
 - The `.card-link` class adds a blue color to any link, and a hover effect.

```
<div class="card bg-warning" style="width:400px">
  <div class="card-body">
    <h4 class="card-title">Anita Patil</h4>
    <p class="card-text">Anita is a freelance web designer</p>
    <a href="#" class="card-link">View Profile</a>
    <a href="#" class="btn btn-primary">Email Anita</a>
  </div>
</div>
```

Anita Patil

Anita is a freelance web designer and developer based in Pune. She is specialized in HTML5, CSS3, JavaScript, Bootstrap, etc.

[View Profile](#) [Email Anita](#)

Bootstrap Lists

- Unstyled Ordered and Unordered Lists

- Sometimes you might need to remove the default styling from the list items. You can do this by simply applying the class `.list-unstyled` to the respective `` or `` elements

```
<!-- normal HTML List -->
<ul>
  <li>Home</li>
  <li>Products
    <ul>
      <li>Gadgets</li>
      <li>Accessories</li>
    </ul>
  </li>
  <li>About Us</li>
  <li>Contact</li>
</ul>
```

- Home
- Products
 - Gadgets
 - Accessories
- About Us
- Contact

```
<ul class="list-unstyled">
  <li>Home</li>
  <li>Products
    <ul>
      <li>Gadgets</li>
      <li>Accessories</li>
    </ul>
  </li>
  <li>About Us</li>
  <li>Contact</li>
</ul>
```

- Home
- Products
 - Gadgets
 - Accessories
- About Us
- Contact

- If you want to create a horizontal menu using ordered or unordered list you need to place all list items in a single line i.e. side by side.

- You can do this by simply applying the class `.list-inline` to the respective `` or ``, and the class `.list-inline-item` to the `` elements.

```
<ul class="list-inline">
  <li class="list-inline-item">Home</li>
  <li class="list-inline-item">Products</li>
  <li class="list-inline-item">About Us</li>
  <li class="list-inline-item">Contact</li>
</ul>
```

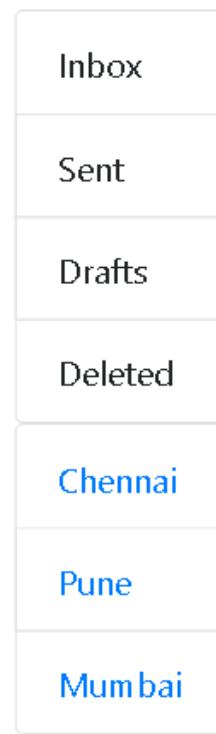
Home Products About Us Contact

Page Components : List Group

- List group is used for creating lists, like a list of useful resources/ list of recent activities.
 - You can also use it for a complex list of large amounts of textual content.
 - Add class `list-group` to a `` or `<div>` element to make its children appear as a list.
 - The children can be `li` or `a` element, depending on your parent element choice.
 - The child should always have the class `list-groupitem`.

```
<!-- List group -->
<ul class="list-group">
  <li class="list-group-item">Inbox</li>
  <li class="list-group-item">Sent</li>
  <li class="list-group-item">Drafts</li>
  <li class="list-group-item">Deleted</li>
</ul>

<div class="list-group">
  <a href="#" class="list-group-item">Chennai</a>
  <a href="#" class="list-group-item">Pune</a>
  <a href="#" class="list-group-item">Mumbai</a>
</div>
```



Page Components : List Group

- We can also apply various colors to each list item by adding `list-group-item-*` classes along with `list-group-item`.

```
<ul class="list-group">
<li class="list-group-item list-group-item-success">Success item</li>
<li class="list-group-item list-group-item-secondary">Secondary item</li>
<li class="list-group-item list-group-item-info">Info item</li>
<li class="list-group-item list-group-item-warning">Warning item</li>
<li class="list-group-item list-group-item-danger">Danger item</li>
<li class="list-group-item list-group-item-primary">Primary item</li>
<li class="list-group-item list-group-item-dark">Dark item</li>
<li class="list-group-item list-group-item-light">Light item</li>
</ul>
```

Success item

Secondary item

Info item

Warning item

Danger item

Primary item

Dark item

Light item

Dropdowns

- A dropdown is a toggleable menu allowing user to choose one value from a predefined list
 - `.dropdown` class indicates a dropdown menu.
 - To open the dropdown menu, use a button or a link with a class of `.dropdown-toggle` and the `data-bs-toggle="dropdown"` attribute.
 - Add the `.dropdown-menu` class to a `` element to actually build the dropdown menu.

```
<div class="dropdown">
  <button type="button"
    class="btn btn-primary dropdown-toggle"
    data-bs-toggle="dropdown">
    Folders
  </button>
  <ul class="dropdown-menu">
    <li><a class="dropdown-item" href="#">Inbox</a></li>
    <li><a class="dropdown-item" href="#">Drafts</a></li>
    <li><a class="dropdown-item" href="#">Sent Mail</a></li>
  </ul>
</div>
```

Dropdowns

Folders ▾

Dropdowns

Folders ▾

Inbox

Drafts

Sent Mail

BOOTSTRAP NAVIGATION BAR

Bootstrap 5 Navs

- If you want to create a simple horizontal menu, add the `.nav` class to a `/<div>/<nav>` element, followed by `.nav-item` for each `/<a>` and add the `.nav-link` class to their links

```
<div> Simple nav without bootstrap nav/a </div>
<nav>
  <a href="#">Home</a>
  <a href="#">About</a>
  <a href="#">Activity</a>
</nav>
```

Simple nav without bootstrap nav/a
[Home](#) [About](#) [Activity](#)

```
<div> Simple nav with nav/a </div>
<nav class="nav">
  <a href="#" class="nav-item nav-link">Home</a>
  <a href="#" class="nav-item nav-link">About</a>
  <a href="#" class="nav-item nav-link">Activity</a>
</nav>
```

Simple nav with nav/a
Home About Activity

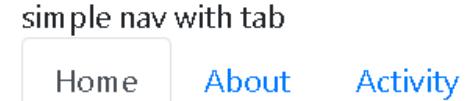
```
<div> Simple nav with ul</div>
<ul class="nav">
  <li><a href="#" class="nav-item nav-link">Home</a></li>
  <li><a href="#" class="nav-item nav-link">About</a></li>
  <li><a href="#" class="nav-item nav-link">Activity</a></li>
</ul>
```

Simple nav with ul
Home About Activity

Bootstrap 4 Navs

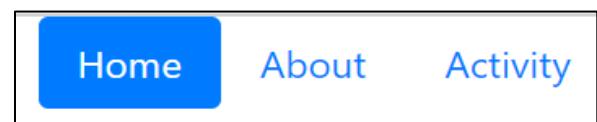
- Add the class **.nav-tabs** to the basic nav to generate a tabbed navigation.

```
<!-- simple nav with tab -->
<nav class="nav nav-tabs">
  <a href="#" class="nav-item nav-link active">Home</a>
  <a href="#" class="nav-item nav-link">About</a>
  <a href="#" class="nav-item nav-link">Activity</a>
</nav>
```



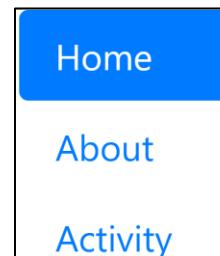
- Similarly, you can create pill based navigation by adding the class **.nav-pills** on the basic nav instead of class .nav-tabs

```
<!-- simple nav with pills -->
<div class="container">simple nav with pills
  <nav class="nav nav-pills">
    <a href="#" class="nav-item nav-link active">Home</a>
    <a href="#" class="nav-item nav-link">About</a>
    <a href="#" class="nav-item nav-link">Activity</a>
  </nav>
```



- Vertically stack these pills by attaching an additional class **flex-column**

```
<!-- simple nav with pills and stacked with flex -->
<nav class="nav nav-pills flex-column">
  ...
</nav>
```



Navbar

- The .nav class is for simple navigation. Whereas .nav-bar is intended for the main page navigation on top of the page and has a whole bunch of styling attributes available.
- First build a <div> or <nav> element, with class navbar.
 - A standard navigation bar is created with the .navbar class, followed by a responsive collapsing class: .navbar-expand-xxl|xl|lg|md|sm (becomes vertical on small screens).
 - To add links inside the navbar, use a element with class .navbar-nav
 - Then add the usual and <a> elements with a .nav-item and a .nav-link class

```
<div>A grey horizontal navbar that becomes vertical on small screens</div>
<nav class="navbar navbar-expand-sm bg-light">
  <!-- Links -->
  <ul class="navbar-nav">
    <li class="nav-item">
      <a class="nav-link" href="#">Link 1</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="#">Link 2</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="#">Link 3</a>
    </li>
  </ul>
</nav>
```

```
<div class="navbar">
</div>
```

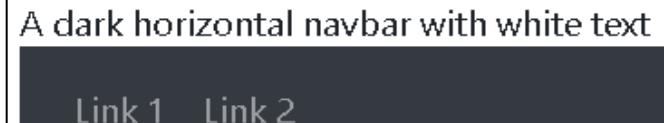
Link 1 Link 2 Link 3

Link 1
Link 2
Link 3

Colored Navbar

- Use any of the `.bg-color` classes to change the background color of the navbar (`.bg-primary`, `.bg-success`, `.bg-info`, `.bg-warning`, `.bg-danger`, `.bg-secondary`, `.bg-dark` and `.bg-light`)
- Tip: Add a white text color to all links in the navbar with the `.navbar-dark` class, or use the `.navbar-light` class to add a black text color.

```
<div>A dark horizontal navbar with white text</div>
<nav class="navbar navbar-expand-sm bg-dark navbar-dark">
  <ul class="navbar-nav">
    <li class="nav-item">
      <a class="nav-link" href="#">Link 1</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="#">Link 2</a>
    </li>
  </ul>
</nav>
```



Link 1 Link 2

```
<nav class="navbar navbar-expand-sm bg-light navbar-light">
```

Navbar with brand/logo

- The .navbar-brand class is used to highlight the brand/logo/project name of your page

```
<!-- navbar with text logo -->
<nav class="navbar navbar-dark bg-secondary">
  <a class="navbar-brand">Pawsome</a>
</nav>
```

navbar with text logo

Pawsome

```
<!-- navbar with image logo -->
<nav class="navbar navbar-dark bg-secondary">
  <a href="#" class="navbar-brand">
    
  </a>
</nav>
```

navbar with image logo



Collapsing The Navigation Bar

- Very often, especially on small screens, you want to hide the navigation links and replace them with a button that should reveal them when clicked on.
- To create a collapsible navigation bar, use a button with class="navbar-toggler", data-bs-toggle="collapse" and data-bs-target="#thetarget".
- Then wrap the navbar content (links, etc) inside a div element with class="collapse navbar-collapse", followed by an id that matches the data-target of the button: "thetarget".

```
<nav class="navbar navbar-expand-md navbar-light bg-light">
  <button type="button"
    class="navbar-toggler"
    data-bs-toggle="collapse"
    data-bs-target="#nb1">
    <span class="navbar-toggler-icon"></span>
  </button>  <!-- when navbar collapses on small dev-->
  <div class="collapse navbar-collapse justify-content-between" id="nb1">
    <div class="navbar-nav">
      <a href="#" class="nav-item nav-link active">Home</a>
      <a href="#" class="nav-item nav-link">Profile</a>
      <a href="#" class="nav-item nav-link">Contact us</a>
    </div>
  </div>
</nav>
```

Home Profile Contact us



Home

Profile

Contact us

Navbar With Dropdown

```
<!-- Navbar With Dropdown -->
<div>Navbar With Dropdown</div><br>
<nav class="navbar navbar-expand-sm bg-dark navbar-dark">
  <a class="navbar-brand" href="#">Logo</a>
  <ul class="navbar-nav">
    <li class="nav-item">
      <a class="nav-link" href="#">Link 1</a>
    </li>
    <!-- Dropdown -->
    <li class="nav-item dropdown">
      <a class="nav-link dropdown-toggle"
         href="#" id="navbardrop"
         data-toggle="dropdown">
        Dropdown link
      </a>
      <div class="dropdown-menu">
        <a class="dropdown-item" href="#">sub-Link 1</a>
        <a class="dropdown-item" href="#">sub-Link 2</a>
        <a class="dropdown-item" href="#">sub-Link 3</a>
      </div>
    </li>
  </ul>
</nav>
```

Logo Link 1 Dropdown link ▾

Logo Link 1 Dropdown link ▾

sub-Link 1
sub-Link 2
sub-Link 3

FORMS

Creating Forms

- By default, forms are aligned vertically as the default value for display is block and is set to 100%.
- We can use additional classes to change this default placement

The form consists of the following fields:

- Name: A text input field with placeholder "Your Name".
- Enter Email Address as the Username: A text input field with placeholder "Enter email".
- Password: A text input field with placeholder "Password".
- Browse to find file: A file input field with a "Browse..." button and a message "No file selected."
- Keep me signed in: A checkbox checked.
- Male: A radio button unselected.
- Female: A radio button selected.
- Login: A primary button labeled "Login".

- There are no classes to make a layout form but we can use some classes to change the alignment of forms.
- Syntax:

```
<form>
  <div class = row|mb-*|col-*>
    <label for="exampleInput"
      class="form-label">
      ...
    </label>
    <input type="text"
      class="form-control"
      id="exampleInput">
  </div>
</form>
```

add a .form-label class to each label element to ensure correct padding

Creating Forms : Vertical Form Layout

- The form controls in this layout are stacked with left-aligned labels on the top.

```
<div class="container">
<form>
  <div class="mb-3">
    <label for="mail" class="form-label">Email</label>
    <input type="email" class="form-control" id="mail" >
  </div>
  <div class="mb-3">
    <label for="pass" class="form-label">Password</label>
    <input type="password" class="form-control" id="pass" >
  </div>
  <div class="mb-3 form-check">
    <input type="checkbox" class="form-check-input" id="c1" >
    <label class="form-check-label" for="c1">Check me out</label>
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
</div>
```

add a .form-label class to each label element to ensure correct padding

Add class .form-control get proper form styling

Email

Password

Check me out

Submit

Inline Forms

- In an inline form, form elements to appear side by side.
 - Add class .row and .col

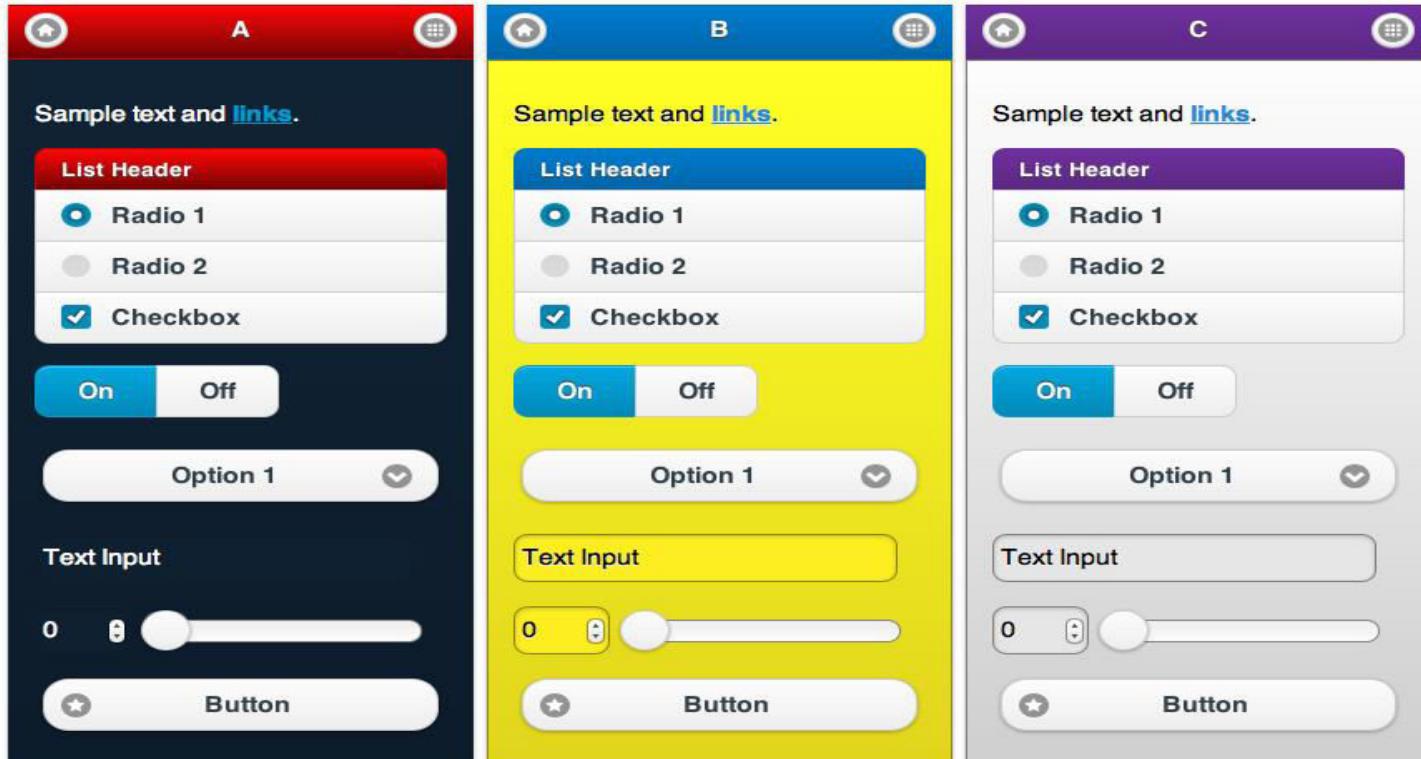
```
<div class="container">
  <div class="row">
    <div class="col col-md-6 col-sm-12">
      <input type="text" class="form-control" placeholder="First name">
    </div>
    <div class="col col-md-6 col-sm-12">
      <input type="text" class="form-control" placeholder="Last name">
    </div>
  </div>
</div>
```

The screenshot shows a horizontal row of two input fields. The first input field is labeled "First name" and the second is labeled "Last name". Both fields are styled with rounded corners and a light gray background.

The screenshot shows a vertical stack of two input fields. The top input field is labeled "First name" and the bottom input field is labeled "Last name". Both fields are styled with rounded corners and a light gray background.

Bootstrap theme

- bootstrap.css is the core css for BootStrap that defines all the style for various controls/components, where as bootstrap-theme.css defines the themes (gradient/animation) for buttons, dropdown, menu, navbar, progressbar, panels.
 - Most of the times adding bootstrap.css is enough for bootstrap to work, but for gradient/animation, you should use bootstrap-theme.css



NODE.JS

Server Side Javascript

Node.js – an intro

- In 2009 Ryan Dahl created Node.js or Node, a framework primarily used to create highly scalable servers for web applications.
 - Node.js is an open source, cross-platform runtime environment for server-side JavaScript.
 - Node.js is required to run JavaScript without a browser support. It uses Google V8 JavaScript engine to execute code.
 - It is written in C++ and JavaScript.
 - Node.js is a development framework that is based on Google's V8 JavaScript engine that powers Google's Chrome web browser.
 - You write Node.js code in JavaScript, and then V8 compiles it into machine code to be executed.
- It's a highly scalable system that uses **asynchronous, non-blocking** I/O model (input/output), rather than threads or separate processes
- It is not a framework like jQuery nor a programming language like C# or JAVA; Its primarily a **Javascript engine**

Node.js is really two things: a runtime environment and a library

Traditional Programming vs Event-driven programming

- In traditional programming I/O is performed in the same way as it does local function calls. i.e. Processing cannot continue until the operation is completed.
 - When the operation like executing a query against database is being executed, the whole process/thread idles, waiting for the response. This is termed as “Blocking”
 - Due to this blocking behavior we cannot perform another I/O operation, the call stack becomes frozen waiting for the response.
- Event-driven programming or Asynchronous programming is a programming style where the flow of execution is determined by events.
- Events are handled by **event handlers or event callbacks**
 - An event callback is a function that is invoked when something significant happens like when the user clicks on a button or when the result of a database query is available.
- This style of programming — whereby instead of using a return value you define functions that are called by the system when interesting events occur — is called event-driven or asynchronous programming.

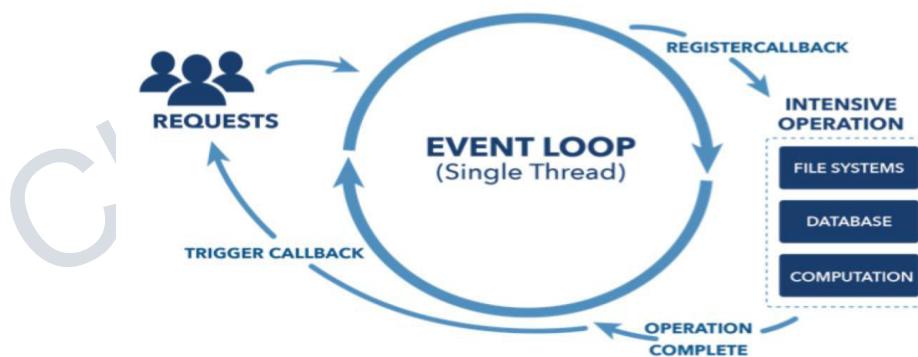
```
result = query('SELECT * FROM posts WHERE id = 1');  
do_something_with(result);
```

Typical blocking I/O
programming

```
query_finished = function(result) {  
    do_something_with(result);  
}  
query('SELECT * FROM posts WHERE id = 1', query_finished);
```

Event loop

- An event loop is a construct that mainly performs two functions in a continuous loop — **event detection and event handler triggering**.
 - In any run of the loop, it has to detect which events just happened.
 - Then, when an event happens, the event loop must determine the event callback and invoke it.
- This event loop is just one thread running inside one process, which means that, when an event happens, the event handler can run without interruption. This means the following:
 - There is at most one event handler running at any given time.
 - Any event handler will run to completion without being interrupted.
 - Node.js uses the “Single Threaded Event Loop” architecture to handle multiple concurrent clients.



Asynchronous and Event Driven

- All APIs of Node.js library are asynchronous that is, non-blocking.
 - It essentially means a Node.js based server never waits for an API to return data.
 - The server moves to the next API after calling it and a notification mechanism of Node.js helps the server to get a response from the previous API call.
 - It is non-blocking, so it doesn't make the program wait, but instead it registers a callback and lets the program continue.
- Node.js is not fit for an application which performs CPU-intensive operations like image processing or other heavy computation work because it takes time to process a request and thereby blocks the single thread.
- Node.js is great for data-intensive applications.
 - Using a single thread means that Node.js has an extremely low-memory footprint when used as a web server and can potentially serve a lot more requests.
 - Eg, a data intensive application that serves a dataset from a database to clients via HTTP
- **What Node is NOT!**

Node is not a webserver. By itself it doesn't do anything. It doesn't work like Apache. There is no config file where you point it to your HTML files. If you want it to be a HTTP server, you have to write an HTTP server (with the help of its built-in libraries). Node.js is just another way to execute code on your computer.

It is simply a JavaScript runtime.

Setting up Node

- To install and setup an environment for Node.js :
 - Download the latest version of Node.js installable archive file from <https://nodejs.org/en/>
 - Double click to run the msi file
 - Verify if the installation was successful : node –v in command window.

The screenshot shows the official Node.js website at <https://nodejs.org/en/>. The page features the Node.js logo and navigation links for HOME, ABOUT, DOWNLOADS, DOCS, GET INVOLVED, SECURITY, NEWS, and FOUNDATION. A prominent green button labeled '8.11.3 LTS' is highlighted as 'Recommended For Most Users'. Another button labeled '10.5.0 Current' is shown as 'Latest Features'. At the bottom, there are links for 'Other Downloads | Changelog | API Docs' on both the left and right sides.

```
C:\Users\Administrator>node -v  
v8.11.3
```

Using the Node CLI : REPL (Read-Eval-Print-Loop)

- There are two primary ways to use Node.js on your machines: by using the Node Shell or by saving JavaScript to files and running those.
 - Node shell is also called the Node REPL; a great way to quickly test things in Node.
 - When you run “node” without any command line arguments, it puts you in REPL

```
c:\ node
Your environment has been set up for
using Node.js 4.4.0 (x64) and npm.

C:\Users\DELL>node
> REPL
```

```
c:\ node
C:\Users\DELL>node
> console.log("hello world");
hello world
undefined
>
```

```
c:\ node
> 10+20
30
> x=50
50
> x
50
> -
```

```
> var foo = [];
undefined
> foo.push(123);
1
> foo
[ 123 ]
>
```

```
> function add(a,b){
...   return (a+b);
...
}
undefined
> add(10,20)
30
>
```

```
> var x = 10, y = 20;
undefined
> x+y
30
>
```

- You can also create a js file and type in some javascript.

```
C:\Users\DELL>node helloworld.js
Hello World!
```

```
//helloworld.js
console.log("Hello World!");
```

Using the REPL

- To view the options available to you in REPL type .help and press Enter.

```
C:\Users\DELL>node
> .help
break  Sometimes you get stuck, this gets you out
clear  Alias for .break
exit   Exit the repl
help   Show repl options
load   Load JS from a file into the REPL session
save   Save all evaluated commands in this REPL session to a file
>
```

```
> .load helloworld.js
> console.log('Hello World!!');
Hello World!!
undefined
>
```

```
for (var i = 1; i < 11; i++)
  console.log(i);
var arr1 = [10, 20, 30];
arr1.push(40,50);
console.log('arr length: ' + arr1.length); //5
console.log('arr contents: ' + arr1); //10,20,30,40,50
```

```
//functionEx.js
function foo() {
  return 123;
}
console.log(foo()); // 123

function bar() { }
console.log(bar()); // undefined
```

```
//JSObjEx.js
var person = {
  name: "Amit",
  age: 23,
  addr: {
    city: 'Pune',
    state: 'Mah'
  },
  hobbies: ['Reading', 'Swimming']
};
console.log(person);
```

```
E:\FreelanceTrg\Node.js\Demo\Intro>node JSObjEx.js
{
  name: 'Amit',
  age: 23,
  addr: { city: 'Pune', state: 'Mah' },
  hobbies: [ 'Reading', 'Swimming' ]
}
```

Node.js Modules

- A module in Node.js is a logical encapsulation of code in a single unit.
 - Since each module is an independent entity with its own encapsulated functionality, it can be managed as a separate unit of work.
- Consider modules to be the same as JavaScript libraries.
 - A set of functions you want to include in your application.
 - Module in Node.js is a simple or complex functionality organized in JavaScript files which can be reused throughout a Node.js application.
 - A module is a discrete program, contained in a single file in Node.js. Modules are therefore tied to files, with one module per file.
- Node.js has a set of built-in modules which you can use without any further installation.
 - Built-in modules provide a core set of features we can build upon.
 - Also, each module can be placed in a separate .js file under a separate folder.
 - To include a module, use the **require()** function with the name of the module.
- In Node, modules are referenced either by file path or by name
 - For example, we can require some native modules:

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

```
var myFile = require('./myFile'); // loads myFile.js
```

Node.js Web App

```
//RunServer.js
```

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

function process_request(req, res) {
    var body = 'Hello World\n';
    var content_length = body.length ;
    res.writeHead(200, {
        'Content-Length': content_length,
        'Content-Type': 'text/plain' });
    res.end(body);
}

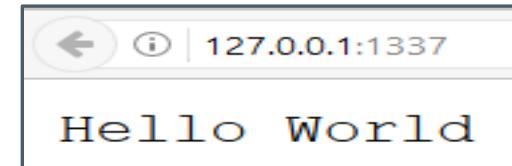
var srv = http.createServer(process_request);
srv.listen(1337, '127.0.0.1');

console.log('Server running at http://127.0.0.1:1337');
```

Import required module using **require**; load http module and store returned HTTP instance into http variable

createServer() : turns your computer into an HTTP server
Creates an HTTP server which listens for request over 1337 port on local machine

```
G:\FreeLanceTrg\Node.js\Demo\Intro>node runserver.js
Server running at http://127.0.0.1:1337/
```



```
var http = require('http');
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
})
.listen(1337, '127.0.0.1');
```

Node.js Module

- Node.js includes three types of modules:
 - Core Modules
 - Local Modules
 - Third Party Modules
- **Loading a core module**
 - Node has several modules compiled into its binary distribution called core modules.
 - It is referred solely by the module name, not by the path and are preferentially loaded even if a third-party module exists with the same name.
 - `var http = require('http');`
- Some of the important core modules in Node.js

| Core Module | Description |
|------------------------------------|---------------------------------------------------------------------------------|
| <u>http</u> | http module includes classes, methods and events to create Node.js http server. |
| <u>url</u> | url module includes methods for URL resolution and parsing. |
| <u>querystring</u> | querystring module includes methods to deal with query string. |
| <u>path</u> | path module includes methods to deal with file paths. |
| <u>fs</u> | fs module includes classes, methods &events to work with file I/O. |
| <u>util</u> | util module includes utility functions useful for programmers. |

Timers module

- Timer module is a core module.
 - Timers module contains several functions that allow you to execute a block of code or a function after a specified amount of time.
 - You don't need to use require() to import the Timers module because it's global.
- Timer functions:
 - setImmediate(): executes code at the end of the current event loop cycle
 - setInterval(): defines a time interval.
 - setTimeout(): executes a one-time callback after delay milliseconds.
 - clearImmediate(immediateObject)
 - clearInterval(intervalObject)
 - clearTimeout(timeoutObject)

```
E:\FreelanceTrg\Node.js\Demo\Modules>node TimerMod
before immediate
after immediate
before timeout
after timeout
executing immediate: so immediate
executing timeout
```

```
console.log('before immediate');

setImmediate((arg) => {
  console.log(`executing immediate: ${arg}`);
}, 'so immediate');

console.log('after immediate');

console.log('before timeout');

setTimeout(() => {
  console.log(`executing timeout`);
}, 3000);

console.log('after timeout');
```

Node.js Local Module

- The local modules are custom modules that are created locally by developer in the app
 - These modules can include various functionalities bundled into distinct files and folders
 - You can also package it and distribute it via NPM, so that Node.js community can use it.
 - For example, if you need to connect to MongoDB and fetch data then you can create a module for it, which can be reused in your application.

```
//Module1.js : exporting variable  
exports.answer=50;
```

```
//UseModule1.js  
var ans=require("./module1");  
console.log(ans.answer);
```

exports object is a special **object** created by the Node module system which is returned as the value of the require function when you include that module.

```
E:\FreelanceTrg\Node.js\Demo\Modules>node module1.js  
  
E:\FreelanceTrg\Node.js\Demo\Modules>node UseModule1.js  
50
```

```
module2.js > ...  
1 //exporting function  
2 exports.sayHelloInEnglish = function(){  
3   return "Hello";  
4 };  
5 exports.sayHelloInSpanish = function(){  
6   return "Hola";  
7 };
```

```
UseModule2.js > ...  
1 var greetings=require("./module2");  
2 console.log(greetings.sayHelloInSpanish());  
3 console.log(greetings.sayHelloInEnglish());  
4 /*  
5 This is equivalent to:  
6 var greetings=require("./module2").sayHelloInSpanish();  
7 console.log(greetings);  
8 */
```

Create Your Own Modules

```
module1a.js > ...
1  exports.bname = 'Core Node.js';
2  exports.read = function() {
3    console.log('I am reading ' + exports.bname);
4 }
```

```
UseModule1a.js > ...
1  var book = require('./module1a.js');
2  console.log('Book name: ' + book.bname);
3  book.read();
```

```
Book name: Core Node.js
I am reading Core Node.js
```

- Exports is just module.exports's little helper. Your module returns module.exports to the caller ultimately, not exports. All exports does is collect properties and attach them to module.exports

```
module2a.js > ...
1  exports.func1 = function() {
2    console.log('in function func1()');
3  };
4
5  module.exports.func2 = function() {
6    console.log('in function func2()');
7  };

```

```
UseModule2a.js > ...
1  const f1 = require('./module2a');
2  console.log(f1);
3
4  f1.func1();
5  f1.func2();
```

```
{ func1: [Function], func2: [Function] }
in function func1()
in function func2()
```

Example

```
module6.js > ...
1 //export methods and values as you go, not just at the end of the file.
2 /*exports.getName = () => {
3     return 'Jim';
4 };
5 exports.getLocation = () => {
6     return 'Munich';
7 };
8 exports.dob = '12.01.1982';
9 */
0 //We can export multiple methods and values in the same way
1 const getName = () => {
2     return 'Jim';
3 };
4 const getLocation = () => {
5     return 'Munich';
6 };
7 const dateOfBirth = '12.01.1982';
8
9 exports.getName = getName;
0 exports.getLocation = getLocation;
1 exports.dob = dateOfBirth;
```

```
UseModule6.js > ...
```

```
1 //we can cherry-pick what we want to import
2 const { getName, dob } = require('./module6');
3 console.log(
4     `${getName()} was born on ${dob}.`
5 ); //Jim was born on 12.01.1982.
```

Alternatively using module.exports

```
//exports.getName = getName;
//exports.getLocation = getLocation;
//exports.dob = dateOfBirth;
...
module.exports = {getName, getLocation, dateOfBirth}
```

```
//module7.js
function sayHello(){
  console.log("Hello World!")
}

const username = 'Joan'

module.exports = {username, sayHello}

//or, using dot notation:
module.exports.username = username
module.exports.sayHello = sayHello

//alternatively
exports.username = username
exports.sayHello = sayHello
```

```
//import the assigned properties with a destructuring assignment:
const { username, sayHello } = require('./module7')

//Or with a regular assignment and dot notation:

const helloModule = require('./module7')

helloModule.sayHello()
console.log(helloModule.username)
```

```
class User {
  constructor(name, age, email) {
    this.name = name;
    this.age = age;
    this.email = email;
  }

  getUserStats() {
    return {
      Name: `${this.name}`,
      Age: `${this.age}`,
      Email: `${this.email}`
    };
  }
}

module.exports = User;
```

```
const User = require('./user');
const jim = new User('Jim', 37, 'jim@example.com');

console.log(jim.getUserStats());
```

```
module3a.js > ...
```

```
1 //exposing JSON object
2 module.exports = {
3   ...
4   firstName: 'James',
5   lastName: 'Bond',
6   display: function(){
7     console.log(this.firstName);
8 }
8 exports.ans=40;
```

```
UseModule3a.js > ...
```

```
1 var person =...require("./module3a");
2 console.log("First Name : " + person.firstName);
3 console.log("Last Name : " + person.lastName);
4 person.display();
5 console.log(person.ans);
```

```
First Name : James
Last Name : Bond
James
undefined
```

```
module4.js > ...
```

```
1 //exporting result of a function that takes args
2 exports.add = function() {
3   ...
4   var sum = 0, i = 0;
5   while (i < arguments.length) {
6     sum += arguments[i++];
7   }
8   return sum;
8 };
```

```
UseModule4.js > ...
```

```
1 var result=require("./module4").add(10,20,30,40);
2 console.log(result);
3 /*
4 This is equivalent to:
5 var addNumbers=require("./module4");
6 console.log(addNumbers.add(10,20,30,40));
7 */
```

```
module5.js > ...
```

```
1 function printA() {
2   console.log('A');
3 }
4 function printB() {
5   console.log('B');
6 }
7 function printC() {
8   console.log('C');
9 }
0
1 module.exports.pi = Math.PI;
2 module.exports.printA = printA;
3 module.exports.printB = printB;
```

```
UseModule5.js > ...
```

```
1 var module5 = require('./module5');
2 module5.printA(); // -> A
3 module5.printB(); // -> B
4 console.log(module5.pi); // -> 3.141592653589793
```

NPM (Node Package Manager)

- Loading a module(Third party) installed via NPM
 - To use the modules written by other people in the Node community and published on the Internet (npmjs.com).
 - We can install those third party modules using the **Node Package Manager** which is installed by default with the node installation.
 - Node Package Manager (NPM) is a command line tool that installs, updates or uninstalls Node.js packages in your application.
 - It is also an online repository for open-source Node.js packages. The node community around the world creates useful modules and publishes them as packages in this repository.
 - NPM is a command line tool that [installs, updates or uninstalls](#) Node.js packages in your application.
 - After you install Node.js, verify NPM installation : [npm -v](#)

```
C:\Users\Shrilata>node -v  
v14.16.0  
  
C:\Users\Shrilata>npm -v  
6.14.11
```

NPM (Node Package Manager)

- Installing Packages
 - In order to use a module, you must install it on your machine.
 - To install a package, type **npm install**, followed by the package name
- There are two ways to install a package using npm: globally and locally.
- **Globally** – This method is generally used to install development tools and CLI based packages. To install a package globally, use the following code.
 - **npm install -g <package-name>**
 - Eg to install expressJS : **npm install -g express**
 - Eg to install Typescript : **npm install -g typescript**
 - Eg to install Angular : **npm install -g @angular/cli**
- **Locally** – This method is generally used to install frameworks and libraries. A locally installed package can be used only within the directory it is installed.
 - To install a package locally, use the same command as above without the -g flag.
 - **npm install <package-name>**
 - Eg : To install cookie parser in Express : **npm install --save cookie-parser**
 - Eg: to install bootstrap : **npm install bootstrap@4.1.1**

NPM (Node Package Manager)

- When packages are installed, they are saved on local machine
- npm installs module packages to the node_modules folder.
- **Installing a package using NPM :** *\$ npm install [g] <Package Unique Name>*
- **To remove an installed package :** *npm uninstall [g] < Package Unique Name>*
- **To update a package to its latest version :** *npm update [g] < Package Unique Name>*

JSInt

- A linter is basically a parser that parses your code and looks for mistakes. It's a great way to save time, maintain quality and write safer code.
 - There are many linter tools for Javascript – JsLint, JsHint, EsLint, Jscs etc
- JSInt is a static code analysis tool used in software development for checking if JavaScript source code complies with coding rules
 - Static Code Analysis :a method of analyzing the source code without running them.
- JSInt is a JavaScript program that looks for problems in JavaScript programs. It is a code quality tool.
 - It takes a JavaScript source and scans it. If it finds a problem, it returns a message describing the problem.
- Install JSInt

```
Command Prompt
Microsoft Windows [Version 10.0.19045.2965]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Shrilata>npm install -g jslint
C:\Users\Shrilata\AppData\Roaming\npm\jslint -> C:\Users\Shrilata\AppData\Roaming\npm\node_modules\jslint\bin\jslint.js
+ jslint@0.12.1
added 25 packages from 23 contributors in 10.97s
```

Static analysis, also called static code analysis, is a method of computer program [debugging](#) that is done by examining the [code](#) without [executing](#) the program. The process provides an understanding of the code structure and can help ensure that the code adheres to industry standards. Static analysis is used in software engineering by software development and [quality assurance](#) teams. Automated tools can assist programmers and developers in carrying out static analysis.

Working with JSLint

- Run JSLint on a JavaScript File

```
//lintdemo.js
function equal(a,b){
    if(a==b)
        return true
    return false
}
```

- After resolving issues:

```
//lintdemo.js
"use strict";
function equal(a, b) {
    if (a === b) {
        return true;
    }
    return false;
}
```

TERMINAL

```
D:\DAC-2023>jslint lintdemo.js
(node:5000) Warning: Accessing non-existent property 'ru
ency
(Use `node --trace-warnings ...` to show where the warni

lintdemo.js
#1 Missing space between ',' and 'b'.
    function equal(a,b){ // Line 1, Pos 18
#2 Expected exactly one space between ')' and '{'.
    function equal(a,b){ // Line 1, Pos 20
#3 Missing 'use strict' statement.
    if(a==b) // Line 2, Pos 5
#4 Expected exactly one space between 'if' and '('.
    if(a==b) // Line 2, Pos 7
#5 Missing space between 'a' and '=='.
    if(a==b) // Line 2, Pos 9
#6 Missing space between '==' and 'b'.
    if(a==b) // Line 2, Pos 11
#7 Expected exactly one space between ')' and 'return'.
    return true // Line 3, Pos 7
#8 Expected '{' and instead saw 'return'.
    return true // Line 3, Pos 7
```

```
D:\DAC-2023>jslint lintdemo.js
(node:16192) Warning: Accessin
dency
(Use `node --trace-warnings ..

lintdemo.js is OK.
```

Loading a third party module : package.json

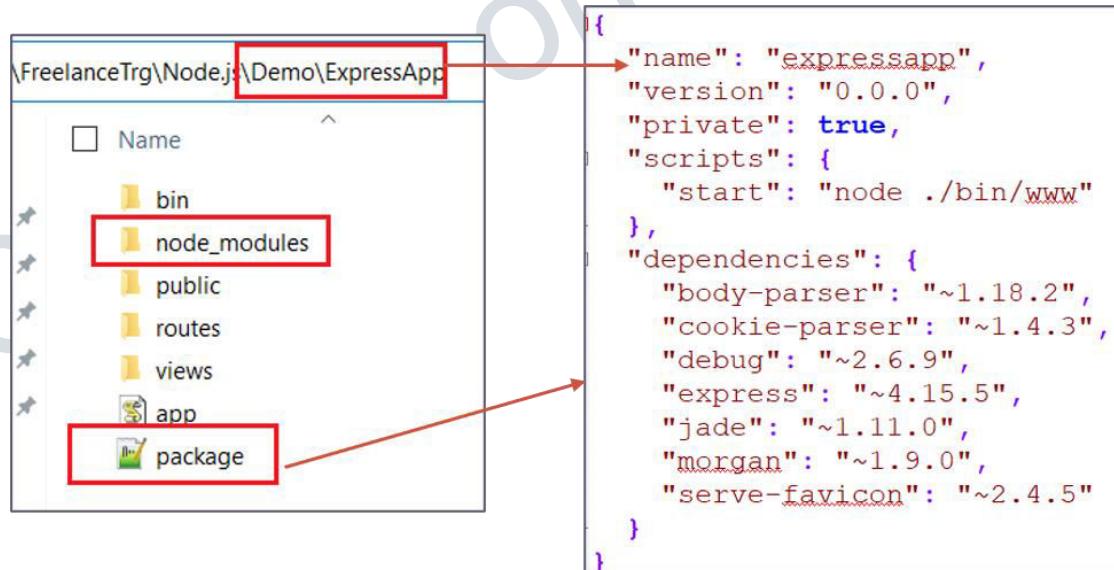
- The package.json file in Node.js is the heart of the entire application.
 - It is basically the manifest file that contains the metadata of the project.
 - package.json is a configuration file from where the npm can recognize dependencies between packages and installs modules accordingly.
 - It must be located in project's root directory.
 - It contains human-readable metadata about the project (like the project name and description) as well as functional metadata like the package version number and a list of dependencies required by the application.
 - Your project also must include a package.json before any packages can be installed from NPM.
 - Eg : a minimal package.json:

```
{  
  "name": "my-app",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "keywords": [],  
  "author": "shrilata",  
  "license": "ISC",  
  "dependencies": {  
    "express": "^4.17.1"  
  }  
}
```

- package-lock.json file
 - Introduced in version 5; keeps track of the exact version of every package that is installed so that a product is 100% reproducible in the same way even if packages are updated by their maintainers.
 - The package-lock.json sets your currently installed version of each package in stone, and npm will use those exact versions when running npm install.

Loading a third party module

- Lets say I want to create a ExpressJS application. I will install ExpressJs locally.
 - Step-1) choose a empty folder
 - Step-2) run npm init to create a package.json file
 - Step-3) install express : npm install express –save (to update package.json)
 - Step-4) check the updated json file to see new dependencies
 - See how package.json is installed in root folder along with node_modules folder
 - My Express app is dependent on a number of other modules
 - All these dependencies will have an entry in package.json



Node.js fs (File System) Module

- The fs module provides a lot of very useful functionality to access and interact with the file system.
 - There is no need to install it. Being part of the Node.js core, it can be used by simply requiring it:
 - `const fs = require('fs')`
- This module provides a wrapper for the standard file I/O operations.
- All the methods in this module has asynchronous and synchronous forms.
 - synchronous methods in this module ends with 'Sync'. For instance `renameSync()` is the synchronous method for `rename()` synchronous method.
 - The asynchronous form always take a completion callback as its last argument.
 - The arguments passed to the completion callback depend on the method, but the first argument is always reserved for an exception. If the operation was completed successfully, then the first argument will be null or undefined.
 - When using the synchronous form any exceptions are immediately thrown. You can use try/catch to handle exceptions or allow them to bubble up.

Asynchronous method is preferred over synchronous method because it never blocks the program execution where as the synchronous method blocks.

Node.js File System

- Node fs module provides an API to interact with FileSystem and to perform some IO operations like create a file, read a File, delete a File etc..
 - fs module is responsible for all the **async or synchronous** file I/O operations.

```
var fs = require('fs');
// write
fs.writeFileSync('test.txt', 'Hello fs!');
// read
console.log(fs.readFileSync('test.txt')); //<Buffer 48 65 6c 6c 6f 20 66 73 21>
console.log(fs.readFileSync('test.txt').toString()); //Hello fs!

console.log(fs.readFileSync('test.txt', 'utf8')); //Hello fs!
```

```
var fs = require("fs");

// Asynchronous read
fs.readFile('test.txt', function (err, data) {
  if (err) {
    return console.error(err);
  }
  console.log("Asynchronous read: " + data.toString());
});

// Synchronous read
var data = fs.readFileSync('test.txt');
console.log("Synchronous read: " + data.toString());

console.log("Program Ended");
```

```
Synchronous read: Hello fs!
Program Ended
Asynchronous read: Hello fs!
```

```
var fs = require('fs');
fs.writeFile('test.txt', 'Hello World!', function (err) {
  if (err)
    console.log(err);
  else
    console.log('Write operation complete.');
});
```

Node.js File System

- `fs.readFile(fileName [,options], callback)` : read the physical file asynchronously.
- `fs.writeFile(filename, data [, options], callback)` : writes data to a file
- `fs.appendFile()`: appends the content to an existing file
- `fs.unlink(path, callback)`; delete an existing file

```
var fs = require("fs")
fs.unlink("test1.txt", function(err){
  if(err) console.log("Err : " , err);
  console.log("delete successful")
})
```

- `fs.exists(path, callback)` : determines if specified file exists or not
- `fs.close(fd, callback(err));`
- `fs.rename(oldPath, newPath, callback)`: rename a file or folder

```
fs.rename("src.json", "tgt.json", err => {
  if (err) {
    return console.error(err)
  }
  console.log('Rename operation complete.');
});
```

Working with Directories

- fs.readdir() : lists all the files and directories within a specified path:

```
const fs = require('fs')
fs.readdir('./', (err, files) => {
  if (err) {
    console.error(err)
    return
  }
  console.log('files: ', files)
})
```

```
if(!fs.existsSync('./newDir')){
  fs.mkdir('./newDir',(err) => {
    if(err) console.log(err)
    console.log("Directory created!");
  })
}
```

```
if(fs.existsSync('./newDir')){
  fs.rmdir('./newDir',(err) => {
    if(err) console.log(err)
    console.log("Directory removed!");
  })
}
```

```
files: [
  'bigsample.txt',
  'first.html',
  'firstpage.html',
  'module1-workingWithJSON.js',
  'module1.js',
  'module2.js',
  'module3.js',
  'module4.js',
  'module5.js',
  'module6-readFile1.js',
  'module6-readFile2.js',
  'module7-writeFile.js',
  'module8.js',
  'moduleWT.js',
  'person.json',
  'ReadFromDirectory.js',
  'RunServerAndServeFile.js',
  'sample.txt',
  'sampleout.txt',
  'student-2.json',
  'temp.txt',
  'test.txt',
  'tgt.json',
  'wt.txt'
]
```

Example

- Reading a file at server end and serving to browser!

```
var http = require('http');
var fs = require('fs');
function process_request(req, res) {
  fs.readFile('bigsample.txt', function(err, data) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.write(data);
    res.end();
  });
}
var s = http.createServer(process_request);
s.listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```



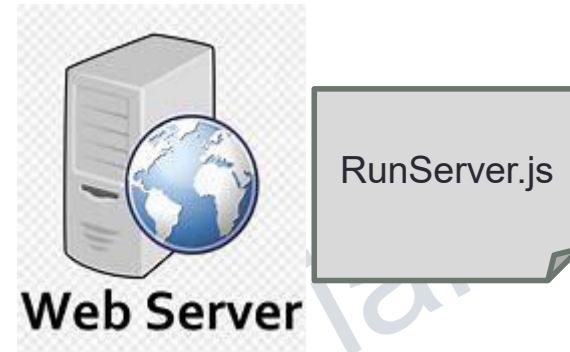
Sending dynamic content to browser

http://localhost:3000/



Browser

A screenshot of a web browser window. The address bar shows "localhost:3000". Below the address bar, there are links for "Apps", "Learn Spring Boot T...", and "Install Postman RES...". The main content area displays the text "Welcome to Node, today is" in large bold letters, followed by the date "Tue Nov 16 2021 17:06:59 GMT+0530 (India Standard Time)".



```
//RunServer.js
let http = require("http")

function handleReq(req, resp){
    var d1 = new Date();

    var body = "<html><body bgcolor='lightgreen'>";
    body += "<h1>Welcome to Node, today is </h1>" + d1;
    body += "</body></html>";
    var content_length = body.length ;
    resp.writeHead(200, {
        'Content-Length': content_length,
        'Content-Type': 'text/html'  });
    resp.end(body);
}

let server = http.createServer(handleReq);
server.listen(3000, "localhost");
```

Web development with Node : http.ServerRequest

- When listening for request events, the callback gets an `http.ServerRequest` object as the first argument (`function(req,res)`)
- This object contains some properties:
 - `req.url`: This property contains the requested URL as a string
 - It does not contain the schema, hostname, or port, but it contains everything after that.
 - Eg : if URL is `:http://localhost:3000/about?a=20` then `req.url` will return `/about?a=20`
 - `req.method`: This contains the HTTP method used on the request. It can be, for example, GET, POST, DELETE, or HEAD.
 - `req.headers`: This contains an object with a property for every HTTP header on the request.

Routing

- Routing refers to the mechanism for serving the client the content it has asked

```
var http = require('http');
var server = http.createServer(function(req,res){
  //console.log(req.url);
  var path = req.url.replace(/\/?(?:\?.*)?$/,'').toLowerCase();
  switch(path) {
    case '':
      res.writeHead(200, { 'Content-Type': 'text/html'});
      res.end('<h1>Home Page</h1>');
      break;
    case '/about':
      res.writeHead(200, { 'Content-Type': 'text/html'});
      res.end('<h1>About us</h1>');
      break;
    case '/admin':
      res.writeHead(200, { 'Content-Type': 'text/html'});
      res.end('<h1>Admin page</h1>');
      break;
    default:
      res.writeHead(404, { 'Content-Type': 'text/plain' });
      res.end('Not Found');
      break;
  }
});
server.listen(3000);
```

```
var http = require("http");
http.createServer(function(request, response) {
  if (request.url === "/" && request.method === "GET") {
    response.writeHead(200, { "Content-Type": "text/html" });
    response.end("Hello <strong>home page</strong>");
  }
  else if (request.url === "/foo" && request.method === "GET") {
    response.writeHead(200, { "Content-Type": "text/html" });
    response.end("Hello <strong>foo</strong>");
  }
  else if (request.url === "/bar" && request.method === "GET") {
    response.writeHead(200, { "Content-Type": "text/html" });
    response.end("Hello <strong>bar</strong>");
  }
  else {
    response.writeHead(404, { "Content-Type": "text/html" });
    response.end("404 Not Found");
  }
}).listen(8000);
```

Handling GET req

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

function process_req(req, res) {
  if (req.method == 'GET' && req.url == '/') {
    fs.readFile('radius.html', function(err, data) {
      res.writeHead(200, {'Content-Type': 'text/html'});
      res.write(data);
      res.end();
    });
  }
  else if(req.method == 'GET' && req.url.substring(0,8) == '/process'){
    var q = url.parse(req.url, true);
    var qdata = q.query;
    var r = qdata.radius;

    var rad = Math.PI + r * r;
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write("The area is : " + rad);
    res.end();
  }
  else
    res.end("not found");
}
var server = http.createServer(process_req)
server.listen(3000);
console.log('server listening on localhost:3000');
```

```
<html>
<body>
  <h1>Login</h1>
  <form action="process">
    Radius : <input name="radius">
    <input type="submit"
           value="Calc Radius">
  </form>
</body></html>
```

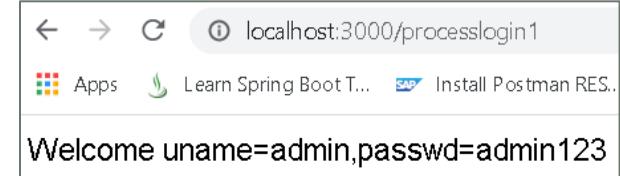
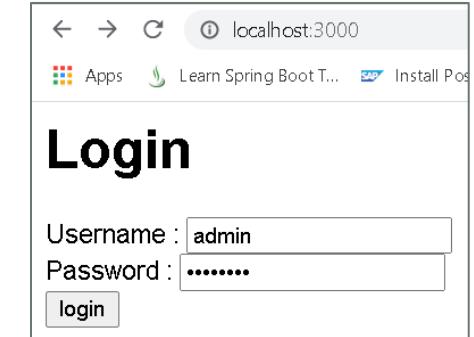
The screenshot shows two browser windows. The top window displays a simple HTML form with a single input field labeled 'Radius:' and a submit button labeled 'Calc Radius'. The bottom window shows the results of the calculation: 'The area is : 10003.14159265359'.

Handling POST req

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

function process_req(req, res) {
  if (req.method == 'GET' && req.url == '/') {
    fs.readFile('loginPost.html', {encoding: 'utf-8'}, function(err, data) {
      res.writeHead(200, {'Content-Type': 'text/html'});
      res.write(data);
      res.end();
    });
  }
  else if(req.method == 'POST'){
    var body = "";
    req.on("data",function(data){
      body += data;
      res.writeHead(200, {'Content-Type': 'text/html'});
      var arr = body.split("&");
      res.write("Welcome " + arr);
      res.end();
    })
  }
}
var server = http.createServer(process_req)
server.listen(3000);
console.log('server listening on localhost:3000');
```

```
<body> //loginpost.html
<h1>Login</h1>
<form action="processlogin"
method="post">
  Username :
  <input name="uname"><br>
  Password :
  <input type="password"
         name="passwd"><br>
  <input type="submit"
         value="login">
</form>
</body>
```



Promise

- Promises are a new feature of ES6.
 - It's a method to write asynchronous code; it represents the completion of an asynchronous function.
 - It is a Javascript object that might return a value in the future.
 - It accomplishes the same basic goal as a callback function, but with many additional features and a more readable syntax.
- Creating a Promise
 - Promises are created by using a constructor called Promise and passing it a function that receives two parameters, resolve and reject, which allow us to indicate to it that it was resolved or rejected.

```
let promise = new Promise(function(resolve, reject) {  
    // executor code - things to do to accomplish your promise  
});
```

```
let promise = new Promise(function(resolve, reject) {  
    // things to do to accomplish your promise  
  
    if(/* everything turned out fine */) {  
        resolve('Stuff worked')  
    } else { // for some reason the promise doesn't fulfilled  
        reject(new Error('it broke'))  
    }  
})
```

Consuming a Promise

- The promise we created earlier has fulfilled with a value, now we want to be able to access the value.
- Promises have a method called `then()` that will run after a promise reaches resolve in the code.
- The `then()` method returns a Promise. It takes up to two arguments: callback functions for the success and failure cases of the Promise.
- Syntax : `p.then(onFulfilled[, onRejected]);`
 - onFulfilled function called if the Promise is fulfilled. This function has one argument, the fulfillment value.
 - Example

```
promise.then(  
  function(result) { /* handle a successful result */ },  
  function(error) { /* handle an error */ }  
)
```

```
promise.then(function(result) {  
  console.log("Promise worked");  
}, function(err) {  
  console.log("Something broke");  
});
```

Consuming a Promise

- Full example

```
const promise = new Promise((resolve, reject) => {
  if(true)
    resolve("resolved!!")
});

promise.then(msg => console.log("In then - "+ msg))
```

Console

"In then - resolved!!"

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 3000);
});

// resolve runs the first function in .then
promise.then(
  result =>console.log(result), //shows "done!" after 3 seconds
  error =>console.log(error) // doesn't run
);
```

D:\>node one.js
done!

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// reject runs the second function in .then
promise.then(
  result => alert(result), // doesn't run
  error => alert(error) // shows "Error: Whoops!" after 1 second
);
```

Consuming a Promise

- Alternatively, instead of using this syntax of `then(fulfilled, rejected)`, you can also use `catch()`
- The **then** and **catch** handlers are asynchronous.
 - Basically, then and catch will be executed once Javascript finished reading the code

```
promise.then(function(result) {  
    console.log(result)  
}).catch(function(err) {  
    console.log(err)  
})  
  
console.log('Hello world')
```

```
const promise = new Promise((resolve, reject) => {  
    // Note: only 1 param allowed  
    return reject('Hi')  
})  
  
// Parameter passed into reject would be the arguments passed into  
// catch.  
promise.catch(err => console.log(err)) // Hi
```

```
function randomDelayed(max = 10, expected = 5, delay = 1000) {  
    return new Promise((resolve, reject) => {  
        const number = Math.floor(Math.random() * max)  
  
        setTimeout(  
            () => number > expected  
            ? resolve(number)  
            : reject(new Error('lower than expected number')), 1000  
        );  
    });  
}  
randomDelayed(100, 75, 2500)  
    .then(number => console.log(number))  
    .catch(error => console.error(error.toString()));
```

Console

```
x "Error: lower than expected number"  
78  
x "Error: lower than expected number"  
89  
x "Error: lower than expected number"  
x "Error: lower than expected number"  
x "Error: lower than expected number"  
85  
x "Error: lower than expected number"
```

Chained Promises

- The methods `promise.then()`, `promise.catch()`, and `promise.finally()` can be used to associate further action with a promise that becomes settled.
 - Each `.then()` returns a newly generated promise object, which can optionally be used for chaining

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('foo');
  }, 300);
});

myPromise
  .then(handleResolvedA, handleRejectedA)
  .then(handleResolvedB, handleRejectedB)
  .then(handleResolvedC, handleRejectedC);
```

```
const myPromise = new Promise((resolve) => {
  setTimeout(() => {
    resolve('foo');
  }, 300);
});

myPromise
  .then((x)=>{console.log("in A - " + x)})
  .then((x)=>{console.log("in B - " + x)})
  .then((x)=>{console.log("in C - " + x)});
```

Console

```
"in A - foo"
"in B - undefined"
"in C - undefined"
```

```
import fs from 'fs';

function readAFile(path) {
  return new Promise((resolve, reject) => {
    fs.readFile(path, 'utf8', (error, data) => {
      if (error) return reject(error);
      return resolve(data);
    });
  });
}

readAFile('./file.txt')
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

```
function myAsyncFunction(url) {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest()
    xhr.open("GET", url)
    xhr.onload = () => resolve(xhr.responseText)
    xhr.onerror = () => reject(xhr.statusText)
    xhr.send()
  });
}
```

Ajax example with promise

```
<!DOCTYPE html>
<html>
<body>
  <div id="msg"></div>
  <button id="btnGet">Get Message</button>
</body>
</html>
```

- You'll see Promises used a lot when fetching data from an API
- Promises can be confusing, both for new developers and experienced programmers that have never worked in an asynchronous environment before.
- However, it is much more common to consume promises than create them. Usually, a browser's Web API or third party library will be providing the promise, and you only need to consume it

```
function load(url) {
  return new Promise(function (resolve, reject) {
    const request = new XMLHttpRequest();
    request.onreadystatechange = function (e) {
      if (this.readyState === 4) {
        if (this.status == 200) {
          resolve(this.response);
        } else {
          reject(this.status);
        }
      }
    }
    request.open('GET', url, true);
    request.send();
  });
}
```

```
btn.onclick = function () {
  load('data.json')
    .then(
      response => {
        const result = JSON.parse(response);
        $('#msg').text(result.message);
      },
      error => $('#msg').text(`Error getting message,
HTTP status: ${error}`);
    )
}
```

Using Fetch

- One of the most useful and frequently used Web APIs that returns a promise is the Fetch API.
 - It allows you to make an asynchronous resource request over a network.
 - `fetch()` is a two-part process, and therefore requires chaining `then()`

```
// Fetch a user from the GitHub API
fetch('https://api.github.com/users/octocat')
  .then((response) => {
    return response.json()
  })
  .then((data) => {
    console.log(data)
  })
  .catch((error) => {
    console.error(error)
  })
```

Console

```
[object Object] {
  avatar_url: "https://avatars.githubusercontent.com/u/1?v=4",
  bio: null,
  blog: "https://github.blog",
  company: "@github",
  created_at: "2011-01-25T18:44:36Z",
  email: null,
  events_url: "https://api.github.com/u...
```

async/await

- async/await added in ECMAScript 2017 (ES8)
- async/await is built on promises
- **async** keyword put in front of a function declaration turns it into an async function
- An async function is a function that knows how to expect the possibility of the **await** keyword being used to invoke asynchronous code.
 - We use the **async** keyword with a function to represent that the function is an asynchronous function.
 - The **async** function returns a promise.

```
//simple synchronous JS function and invocation
function hello() { return "Hello" };
hello();
```

```
//converting above JS function into async function
async function hello() { return "Hello" };
hello();
```

```
// You can also create an async function expression like this:
let hello = async function() { return "Hello" };
hello();
```

```
//you can also use arrow functions:
let hello = async () => { return "Hello" };
```

async/await

- To consume the value returned when the promise fulfills (since it is returning a promise) use a `.then()` block

```
hello().then((value) => console.log(value))  
//or even just shorthand such as  
hello().then(console.log)
```

```
//complete example  
async function hello(){  
  return "hello-1"  
}  
hello().then((x)=>console.log(x))  
// returns"hello-1"
```

```
let f = async () => {  
  console.log('Async function.');//function returns a promise  
  return Promise.resolve("Hello-1");  
}  
  
f().then(function(result) {  
  console.log(result)  
});
```

Console
"Async function."
"Hello-1"

The await keyword

- The **await** keyword is used inside the `async` function to wait for the asynchronous operation
 - `await` can be put in front of any `async` promise-based function to pause your code on that line until the promise fulfills, then return the resulting value.
 - Syntax: `let result = await promise;`
 - You can use `await` when calling any function that returns a `Promise`, including web API functions

```
// a promise
let promise = new Promise(function (resolve, reject) {
  setTimeout(function () {
    resolve('Promise resolved'), 4000);
  });
}

async function asyncFunc() { // async function
  // wait until the promise resolves
  let result = await promise;

  console.log(result);
  console.log('hello');
}

asyncFunc(); // calling the async function
```

Console
"Promise resolved"
"hello"

```
function logFetch(url) {
  return fetch(url)
    .then(response => response.text())
    .then(text => {
      console.log(text);
    }).catch(err => {
      console.error('fetch failed', err);
    });
}
```

```
async function logFetch(url) {
  try {
    const response = await fetch(url);
    console.log(await response.text());
  }
  catch (err) {
    console.log('fetch failed', err);
  }
}
```

```
var fs = require('fs');
fs.writeFile('test.txt', 'Hello World!', function (err) {
  if (err) console.log(err);
  console.log('Write operation complete.');

  fs.appendFile('test.txt', '\n\nAppending new Data!', function (err) {
    if (err) console.log(err);
    console.log('Append operation complete.');

    fs.rename('test.txt', 'newTest.txt', function (err) {
      if (err) console.log(err);
      console.log('Rename operation complete.');
    });
  });
});
```

```
var fsPromises = require('fs').promises;

var fileOps = async() => {
  try{
    await fsPromises.writeFile("test.txt", "Hello Promises!");
    await fsPromises.appendFile("test.txt", "\nSome more content!");
    await fsPromises.rename("test.txt", "newTest.txt");
    var data = await fsPromises.readFile("newTest.txt", "utf8");
    console.log(data);
  }
  catch(err){
    console.error(err);
  }
}
fileOps()
```

fs.promises contains a subset of the interface for what's on fs, but with promise-based interfaces instead of plain callback-style interfaces.

Axios

- Making API calls is integral to most applications
- Axios is a popular HTTP client available as a JavaScript library with more than 22 million weekly downloads as of May 2022.
 - We can make API calls with Axios from JavaScript applications irrespective of whether the JavaScript is running on the front-end or the server-side.
- Why do we need to use a library like Axios when JavaScript already provides built-in objects: XMLHttpRequest and the Fetch API for interacting with APIs
 - Axios in contrast to these **built-in objects** is an **open-source library** that we need to include in our application for making API calls over HTTP.
 - It is similar to the Fetch API and returns a JavaScript Promise object but also includes many powerful features.
 - Axios makes it easy to send asynchronous HTTP requests to REST endpoints and perform CRUD operations.
 - It can be used in **plain JavaScript** or with **a library such as Vue or React**.
 - Axios **works both in the browser and in a Node.js environment**.
 - On the server-side it uses the native node.js http module, while on the client (browser) it uses XMLHttpRequests

Installing Axios

- We can add Axios to our project/code with one of following simple ways:
 1. Using npm: `npm install axios`
 2. Using jsDelivr CDN: `<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>`
- Making an HTTP request - simply pass a config object to the `axios()` function.
- Example:

```
// Send a POST request
axios({
  method: 'post',
  url: '/user/12345',
  data: {
    firstName: 'Fred',
    lastName: 'Flintstone'
  }
});
```

- Axios also provides a set of shorthand methods for performing different types of requests.
- The methods include:
 - `axios.request(config)`
 - `axios.get(url[, config])`
 - `axios.delete(url[, config])`
 - `axios.head(url[, config])`
 - `axios.options(url[, config])`
 - `axios.post(url[, data[, config]])`
 - `axios.put(url[, data[, config]])`
 - `axios.patch(url[, data[, config]])`

```
axios.post('/login', {
  firstName: 'Finn',
  lastName: 'Williams'
});
```

Axios configurations

- Following are some of the available config options for making requests.
 - Only the url is required.
 - Requests will default to GET if method is not specified.

```
{  
  url: '/user', // the server URL that will be used for the request  
  method: 'get', // request method to make the request; default - get  
  transformRequest: [function (data, headers) {  
    // Do whatever you want to transform the data  
    // only applicable for 'PUT', 'POST', 'PATCH' & 'DELETE'  
    return data;  
  }],  
  transformResponse: [function (data) {  
    // Do whatever you want to transform the data  
    return data;  
  }],  
  // `data`:the data to be sent as the request body  
  // Only applicable for 'PUT', 'POST', 'DELETE', and 'PATCH'  
  data: {  
    firstName: 'Fred'  
  },...  
}
```

Axios configurations

- The response for a request contains the following information:

```
{  
  data: {}, //the response coming from server  
  status: 200, // response status code  
  statusText: 'OK', //response status message  
  headers: {}, //response headers  
  config: {}, //the config that was provided for the request  
}
```

- When using then, you will receive the response as follows:

```
axios.get('/user/12345')  
.then(function (response) {  
  console.log(response.data);  
  console.log(response.status);  
  console.log(response.statusText);  
  console.log(response.headers);  
  console.log(response.config);  
});
```

```
axios.post('/login', {  
  firstName: 'Anita',  
  lastName: 'Patil'  
})  
.then((response) => {  
  console.log(response);  
}, (error) => {  
  console.log(error);  
});
```

If the promise is fulfilled, the first argument of then() will be called; if the promise is rejected, the second argument will be called.

Example code

- Using Axios in a simple CRUD service component:

```
const baseUrl = 'http://localhost:8080/course/courses'

class CourseService{

    getCourses(){
        return axios.get(baseUrl)
    }
    getCourseById(id){
        return axios.get(baseUrl+"/"+id)
    }
    updateCourse(course){
        return axios.put(baseUrl+"/"+course.id, course)
    }
    addCourse(course){
        return axios.post(baseUrl, course)
    }
    deleteById(id){
        return axios.delete(baseUrl+"/"+id)
    }
}
```

Demo : axios.get()

- Lets see a simple example: See what console shows

```
function getTodos() {  
  axios  
    .get('https://jsonplaceholder.typicode.com/todos?_limit=5')  
    .then(response => console.log(response))  
}
```

```
▼ {data: Array(5), status: 200, statusText: '', headers: {...}, config: {...}, ...} ⓘ  
  ► config: {url: 'https://jsonplaceholder.typicode.com/todos?_limit=5', method: 'get', ...}  
  ▼ data: Array(5)  
    ► 0: {userId: 1, id: 1, title: 'delectus aut autem', completed: false}  
    ► 1: {userId: 1, id: 2, title: 'quis ut nam facilis et officia qui', completed: false}  
    ► 2: {userId: 1, id: 3, title: 'fugiat veniam minus', completed: false}  
    ► 3: {userId: 1, id: 4, title: 'et porro tempora', completed: true}  
    ► 4: {userId: 1, id: 5, title: 'laboriosam mollitia et enim quasi adipisci quia provi  
      length: 5  
    ► [[Prototype]]: Array(0)  
  ► headers: {cache-control: 'max-age=43200', content-type: 'application/json; charset=ut  
  ► request: XMLHttpRequest {readyState: 4, timeout: 0, withCredentials: false, upload: X  
    status: 200  
    statusText: ""  
  ► [[Prototype]]: Object
```

Demo : axios.post()

- `axios.post(url, data)` – takes 2 required parameters - url to connect to server and Data to post to server.

```
function addTodo() {
  axios
    .post('https://jsonplaceholder.typicode.com/todos', {
      title: 'New Todo',
      completed: false
    })
    .then(res => console.log(res))
}
```

data is a JS object which will be converted to JSON by axios

Response object in console

```
▼ {data: {...}, status: 201, statusText: '', headers: {...}, config: {...}, ...} ⓘ
  ► config: {url: 'https://jsonplaceholder.typicode.com/todos', method: 'post', data: '{"title": "New Todo", "completed": false}'}
  ▼ data:
    completed: false
    id: 201
    title: "New Todo"
  ► [[Prototype]]: Object
  ► headers: {cache-control: 'no-cache', content-length: '60', content-type: 'application/json'}
  ► request: XMLHttpRequest {readyState: 4, timeout: 0, withCredentials: false, upload: XMLHttpRequest}
  status: 201
  statusText: ""
  ► [[Prototype]]: Object
```

This is some dummy data that API returns for some pseudo successful save

Axios Error handling

- Use catch() for handling errors

```
function getTodos() {  
  axios  
    .get('https://jsonplaceholder.typicode.com/todos1?_limit=5')  
    .then(response => console.log(response))  
    .catch(err => console.error(err, err.response));  
}
```

Incorrect URL

✖ ► GET https://jsonplaceholder.typicode.com/todos1?_limit=5 spread.js:25
404 (Not Found)

✖ ► Error: Request failed with status code 404 main.
at e.exports (spread.js:25:1)
at e.exports (spread.js:25:1)
at d.onreadystatechange (spread.js:25:1)
► {data: {...}, status: 404, statusText: '', headers: {...}, config: {...}, ...}

>

```
.catch(function (error) {  
  if (error.response) { // get response with a status code not in range 2xx  
    console.log(error.response.data);  
    console.log(error.response.status);  
    console.log(error.response.headers);  
  }  
})
```

EXPRESS.JS

Node.js web application framework

Introduction

- If you write serious apps using only core Node.js modules you will find yourself reinventing the wheel by writing the same code continually for similar tasks, such as the following:
 - Parsing of HTTP request bodies and Parsing of cookies
 - Managing sessions
 - Organizing routes with a chain of if conditions based on URL paths and HTTP methods of the requests
 - Determining proper response headers based on data types



- Express is a **web application framework** for Node
 - It's built on top of Node.js.
 - It provides various features that make web application development fast and easy compared to Node.js.
- Express.js Installation
 - Create a new folder: eg E:\Express-app
 - E:\Express-app>npm init //for creating package.json
 - E:\Express-app>npm install express --save

```
{  
  "name": "express-app",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "express": "^4.17.1"  
  }  
}
```

First app

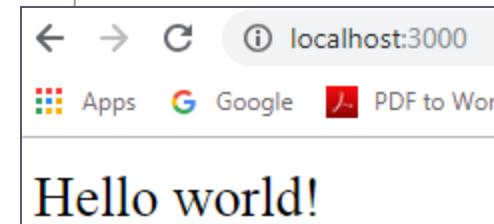
- An Express app is created by calling the `express()` function
 - `express()` : Creates an Express application; is a top-level function exported by the express module
 - The `app` object conventionally denotes the Express application.
 - This object, which is traditionally named `app`, has methods for routing HTTP requests, configuring middleware, rendering HTML views, registering a template engine, and modifying application settings that control how the application behaves

```
let express = require("express") // import the express module

let app = express() // create an Express application

app.get("/", function(req, resp){
  resp.send("Hello world")
})

app.listen(3000, function(){
  console.log("app running on port 3000")
})
```



The `app.listen()` method returns an `http.Server` object

Express Routes

- HTTP verb and URL combinations are referred to as routes, and Express has efficient syntax for handling them.

```
var express = require("express");
var http = require("http");
var app = express();

app.get("/", function(req, res, next) {
    res.send("Hello <strong>home page</strong>");
});

app.get("/foo", function(req, res, next) {
    res.send("Hello <strong>foo</strong>");
});

app.get("/bar", function(req, res, next) {
    res.send("Hello <strong>bar</strong>");
});
//app is passed to the http.createServer() method

http.createServer(app).listen(8000);
```

Open browser and type:
http://localhost:3000
http://localhost:3000/foo
http://localhost:3000/bar
http://localhost:3000/admin - gives error

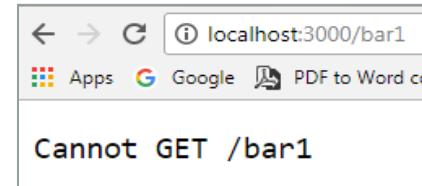
Express Routes

- If none of your routes match the request, you'll get a "**Cannot GET <your-request-route>**" message as response.
- This message can be replaced by a 404 not found page using this simple route

```
app.get('*', function(req, res){  
    res.send('Sorry, this is an invalid URL.');//  
});
```

```
app.get("/", function(req, res, next) {  
    res.send("Hello <strong>home page</strong>");//  
});
```

```
app.get('/course/:id', function(req, res){  
    var course = //code to retrieve course  
    If(!course){  
        res.status(404).send("course not found");//  
    };
```



- The `get()` method defines routes for handling GET requests.
- Express also defines similar methods for the other HTTP verbs (`put()`, `post()`, `delete()`, and so on).
 - **app.method(path, handler)** : This METHOD can be applied to any one of the HTTP verbs – `get`, `post`, `put`, `delete`.
 - The path is a string or regular expression representing the URL that the route responds to. Note that the query string is not considered part of the route's URL.
- Also notice that we haven't defined a `404` route, as this is the **default behavior** of Express when a request does not match any defined routes.

Routing basics

```
app.get("/", function(req, res, next) {  
    res.send("Hello <strong>home page</strong>");  
});
```

- Express also augments the request and response objects with additional methods. Example [response.send\(\)](#) .
 - send() is used to send a response status code and/or body back to the client.
 - If the first argument to send() is a number, then it is treated as the status code. If a status code is not provided, Express will send back a 200.
 - The response body can be specified in the first or second argument, and can be a [string](#), [Buffer](#), [array](#), or [object](#).
- send() also sets the Content-Type header unless you do so explicitly.
 - If the body is a string, Express will set the Content-Type header to text/html.
 - If the body is an array or object, then Express will send back JSON.
 - If the response body is a Buffer, the Content-Type header is also set to application/octet-stream

Routing basics

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

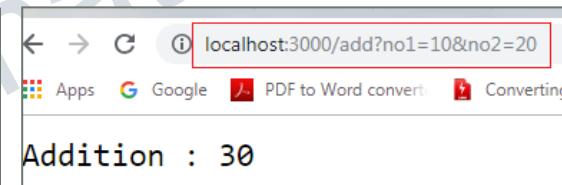
app.get("/buff", function(req, res, next) {
    var buff = Buffer.from("Hello World");
    res.send(buff.toString());
});
app.get("/string", function(req, res, next) {
    res.send("Hello <strong>String response</strong>");
});
app.get("/json", function(req, res, next) {
    res.send({name:'Soha',age:23});
});
app.get("/array", function(req, res, next) {
    res.send(['NodeJS','Angular','ExpressJS']);
});
app.get("/file", function(req, res, next) {
    res.sendFile(path.join(__dirname + '/summer.html'));
});
app.listen(3000);
```

Working with parameters using get

- **req.query**

- Express parses query string parameters by default, and puts them into the req.query property.
- If the request is GET /search?username=Shrilata
req.query.username returns "Shrilata"
- Lets say the incoming url is : <http://localhost:3000/add?no1=10&no2=20>
- Use req.query to query the request parameters

```
app.get("/add", function (req, res) {  
  n1 = parseInt(req.query.no1);  
  n2 = parseInt(req.query.no2);  
  res.end("Addition : " + (n1 + n2));  
});
```



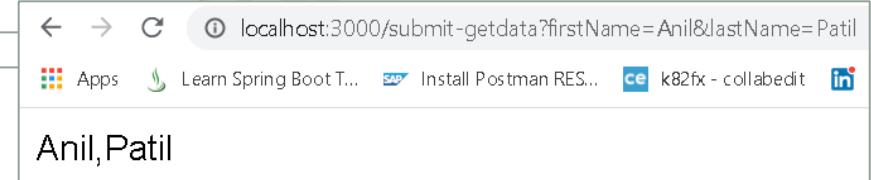
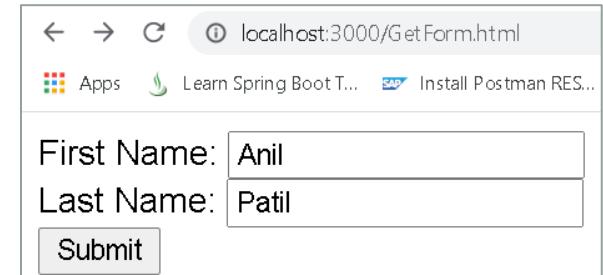
Handle GET Request

```
④ GetForm.html > ...
<!DOCTYPE html>
<html>
<body>
  <form action="/submit-getdata" method="get">
    First Name: <input name="firstName" type="text" />
    Last Name: <input name="lastName" type="text" />
    <input type="submit" />
  </form>
</body>
</html>
```

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

app.get('/GetForm.html', function (req, res) {
  res.sendFile('public/GetForm.html' , { root : __dirname});
});

app.get('/submit-getdata', function (req, res) {
  console.log(req.query.firstName);
  console.log(req.query.lastName);
  res.send(req.query.firstName+" "+req.query.lastName);
});
app.listen(3000);
```



Multiple different methods

- We can also have multiple different methods at the same route.

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

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

app.post('/hello', function(req, res){
  res.send("You just called the post method at '/hello'!\n");
});

app.delete('/hello', function(req, res){
  res.send("You just called the delete method at '/hello'!\n");
});

app.listen(3000);
```

Handle POST Request

- To handle HTTP POST request in Express.js version 4 and above, you need to install a middleware module called [body-parser](#).
 - This is used to parse the body of requests which have payloads attached to them.
 - Install it using : **npm install --save body-parser**
 - Mount it by including the following lines in your js

```
var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: false }));
```

- This body-parser module parses the JSON, buffer, string and url encoded data submitted using HTTP POST request.
- Eg :To parse json data: app.use(bodyParser.json())

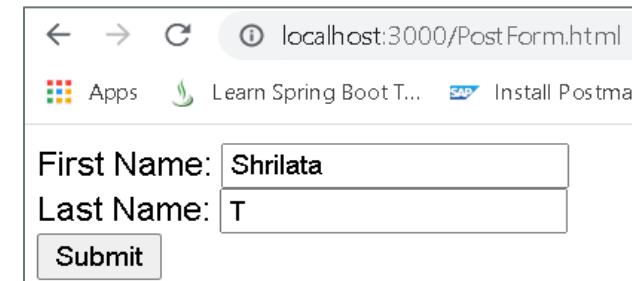
bodyParser.urlencoded(): Parses the text as URL encoded data (which is how browsers tend to send form data from regular forms set to POST) and exposes the resulting object (containing the keys and values) on [**req.body**](#)

```

<!DOCTYPE html>
<html>  <!-- PostForm.html -->
<body>
    <form action="/submit-data" method="post">
        First Name: <input name="firstName" type="text" />
        Last Name: <input name="lastName" type="text" />
        <input type="submit" />
    </form>
</body>
</html>

```

Handle POST Request



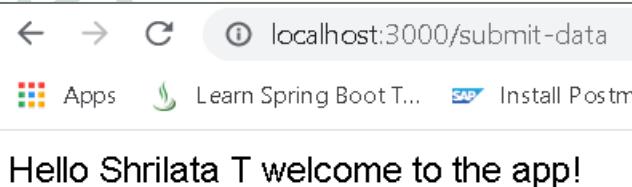
```

var express = require('express');
var app = express();
var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: false }));

app.get('/PostForm.html', function (req, res) {
    //res.sendFile('E:\\Node.js\\Demo\\expressPrj\\PostForm.html');
    res.sendFile('public/PostForm.html' , { root : __dirname});
});

app.post('/submit-data', function (req, res) {
    var name = req.body.firstName + ' ' + req.body.lastName;
    res.send("Hello " + name + ' welcome to the app!');
});
app.listen(3000);

```



Route Parameters

- One of the most powerful features of routing is the ability to use placeholders to extract named values from the requested route, marked by the colon (:) character.
 - When the route is parsed, express puts the matched placeholders on the req.params object for you.

```
app.get('/user/:id', function(req, res) {  
    res.send('user ' + req.params.id);  
});  
  
app.get('/product/:prodname', function(req, res) {  
    res.send('Product : ' + req.params.prodname);  
});
```

Placeholders match any sequence of characters except for forward slashes.



REST WITH EXPRESS

Example

- npm install body-parser –save
 - **body-parser** middleware helps us decode the body from an HTTP request: It parses the body of the request and lets us react to it
- npm install cors –save
 - **cross-origin resource sharing** : Since we are calling the API from different locations by hitting endpoints in the browser. We also have to install the CORS middleware.

```
var express = require('express');
var app = express();
const bodyParser = require('body-parser');
const cors = require('cors'); //cross-origin resource sharing

// Configuring body parser middleware
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

let courses = [{id:1,name:'Java'},
               {id:2,name:'AJAX'},
               {id:3,name:'Angular'}
];
```

```

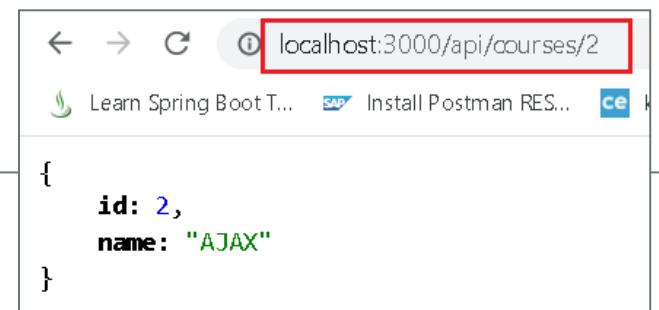
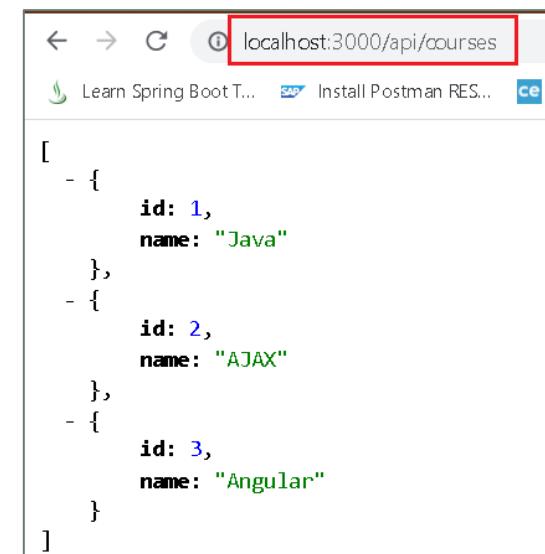
app.get('/', function(req, res){
  let str = "<h1>Hello World </h1>"
  str += "Do you want to <a href='AddCourse.html'>Add new course</a>"
  res.send(str);
});

app.get('/AddCourse.html', function(req, res){
  res.sendFile('AddCourse.html' , { root : __dirname});
});

app.get('/api/courses', function(req, res){
  res.send(courses);
});

//logic to look for a course with given id
app.get('/api/courses/:id', function(req, res){
  var course;
  for(i in courses){
    if(courses[i].id == parseInt(req.params.id)){
      console.log("found ");
      course = courses[i];
    }
  }
  res.send(course);
});

```



```

app.post('/api/courses', function(req, res){
  let obj = req.body
  obj.id = courses.length+1
  courses.push(obj);
  res.send("Added course : " + JSON.stringify(obj));
});

app.delete('/api/courses/:id', function(req, res){
  var course, index=0;
  for(c in courses){
    if(courses[c].id == parseInt(req.params.id)){
      console.log("course found for delete");
      course = courses[c];
      index++;
    }
  }
  courses.splice(index-1,1)
  res.send("course with id " + req.params.id
           + " deleted");
});

app.listen(3000);

```



```

<body>
  <form method="post" action="api/courses">
    New Course Name : <input name="name"><br>
    <input type="submit" value="Add course">
  </form>
</body>

```

| Method | URL endpoint | what it does |
|--------|----------------------------------------------------------------------------------|-----------------------------|
| get | localhost:3000 | fetch home page |
| get | localhost:3000/AddCourse.html | fetch AddCourse.html |
| get | localhost:3000/api/courses | fetch all courses |
| get | localhost:3000/api/courses/2 | fetch course with given ID |
| post | localhost:3000/api/courses | create course |
| delete | localhost:3000/api/courses/2 | delete course with given Id |

Mongodb with mongoose :CRUD Operations Using Mongoose

- Mongo is a document-oriented database that stores data in BSON formatted documents.
- To access Mongo from within a Node/Express application, a driver is required.
- There are a number of Mongo drivers available, but MongoDB and Mongoose is among the most popular
- Mongoose is a package that serves as a mediator between the NodeJS application and the MongoDB server.
 - It is an Object Document Mapper(ODM) that allows us to define objects with a **strongly-typed schema** that is mapped to a MongoDB document.
 - Mongoose supports all the CRUD operations
- Install mongoose module : `npm install mongoose`

```
show dbs;  
use trgdb;  
db.userdata.find()
```

Connect to local MongoDB with Mongoose

- **Step-0:** Install mongodb and mongoose : `npm install mongodb mongoose`
- **Step-1 :** Create a connection to the MongoDB database using Mongoose
 - This connects to a local MongoDB database named `trgdb` and logs a message to the console when the connection is successful.

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

const mongoose = require('mongoose');

mongoose.connect('mongodb://127.0.0.1:27017/trgdb',
{ useNewUrlParser: true, useUnifiedTopology: true })
.then(() => console.log('MongoDB connected'))
.catch((err) => console.log(err));

app.listen(3030);
```

useNewUrlParser: Set to true to use the new MongoDB connection string parser.

useUnifiedTopology: Set to true to use the new Server Discovery and Monitoring engine.

```
PS E:\FreelanceTrg\Node.js\Demo\MongoApp> node .\Mongoose-local-demo.js
MongoDB connected
```

Connect to local MongoDB with Mongoose

- **Step-2:** Define a Mongoose schema for the data that will be stored in the DB

```
const mongoose = require('mongoose');

const contactsSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true
  },
  email: {
    type: String,
    required: true
  },
  phoneno: {
    type: String,
    required: true
  },
  createdAt: {
    type: Date,
    default: Date.now
  }
});

module.exports = mongoose.model('Contact', contactsSchema);
```

Schema is a representation of the structure of the data. It allows us to decide exactly what data we want, and what options we want the data to have as an object

The Schema allows you to define the fields stored in each document along with their validation requirements and default values



Schemas are "compiled" into models using the `mongoose.model()` method. Once you have a model you can use it to find, create, update, and delete objects of the given type.

Connect to local MongoDB with Mongoose

- **Step-3** : Finally, use the defined schema to perform CRUD on documents in the DB

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

const mongoose = require('mongoose');
const Contact = require('./model/contactmodel')

mongoose.connect('mongodb://127.0.0.1:27017/trgdb',
  { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => console.log('MongoDB connected'))
  .catch((err) => console.log(err));

const newContact = new Contact({
  name:'Shrilata',
  email:'shri@gmail.com',
  phoneno:'9988776655'
});
```

The Contact model is imported from the previously defined schema file

The methods are all provided by Mongoose and simplify the process of interacting with the database

```
newContact.save()
  .then(() => console.log('Contact created'))
  .catch(err => console.log(err));

Contact.find()
  .then(contacts => console.log(contacts))
  .catch(err => console.log(err));

Contact.findOneAndUpdate({name:'Shrilata'}, {name:'soha'})
  .then(() => console.log('contact updated'))
  .catch(err => console.log(err));

Contact.deleteOne({name:'Shrilata'})
  .then(() => console.log('contact deleted'))
  .catch(err => console.log(err));

app.listen(3030);
```

Express Router

Express router is a class which helps us to create router

- Router is like a mini express application; capable only of performing middleware and routing functions. Every Express application has a built-in app router.

```
var router = express.Router();  
  
router.get('/', function(req, res) {  
  res.send('In the home page!');  
});  
  
router.get('/about', function(req, res) {  
  res.send('In the about page!');  
});  
  
// apply the routes to our application  
app.use('/', router);
```

We call an instance of the `express.Router()`, apply routes to it, and then tell our application to use those routes. We can now access the home page at `http://localhost:3000` and the about page at `http://localhost:3000/about`.

`app.use("/",router);` tells Express to use the Routes we have defined above.

- Notice how we set a default root for using these routes we just defined. If we had changed last line to `app.use('/app', router)`, then our routes would be `http://localhost:3000/app` and `http://localhost:3000/app/about`.

```
//bookRoutes.js
const express = require("express");
const router = express.Router();

router.get("/books/:bid", function (req, res) {
    res.send("<h1>Sending book with ID : " + req.params.bid
+ "</h1>");
});

router.get("/books", function (req, res) {
    res.send("<h1>Sending list of all books</h1>");
});

router.post("/books", function (req, res) {
    res.send("<h1>created new book</h1>");
})

module.exports = router;
```

```
//route-demoapp.js
const express = require('express')
const app = express()
const bookroutes = require("./routes/bookRoutes")

app.use("/book", bookroutes)

app.listen(3000, ()=>{
    console.log("Server listening on http://localhost:3000")
})
```

The image displays two browser windows side-by-side. The top window shows the response to the URL `localhost:3000/book/books`, which is the list of all books. The bottom window shows the response to the URL `localhost:3000/book/books/101`, which is the details of a specific book with ID 101.

localhost:3000/book/books

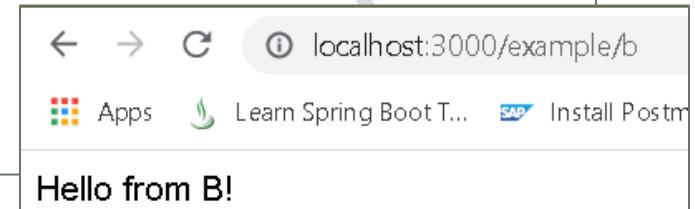
Sending list of all books

localhost:3000/book/books/101

Sending book with ID : 101

Middleware

```
//More than one callback function can handle a route
app.get('/example/b', function (req, res, next) {
  console.log('the response will be sent by the next function ...')
  next()
}, function (req, res) {
  res.send('Hello from B!')
})
```

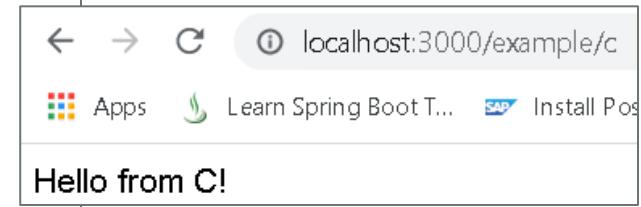


```
//An array of callback functions can handle a route.
var cb0 = function (req, res, next) {
  console.log('CB0')
  next()
}

var cb1 = function (req, res, next) {
  console.log('CB1')
  next()
}

var cb2 = function (req, res) {
  res.send('Hello from C!')
}

app.get('/example/c', [cb0, cb1, cb2])
```



Application level middleware

- Types of middleware: Application-level, Third-party, Router-level, Built-in, Error handling

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

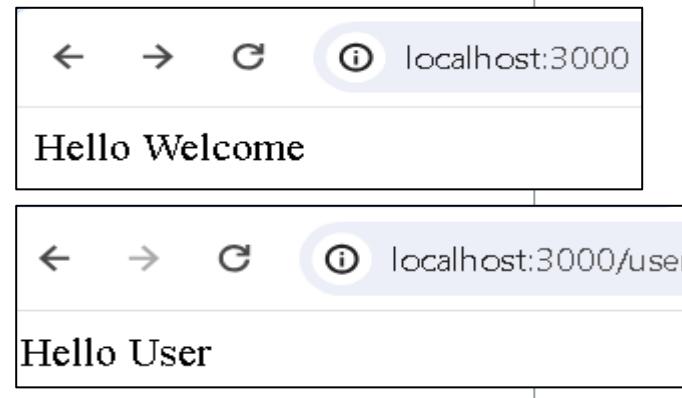
const loggerMiddleware = (req, resp, next) => {
  console.log(`[${new Date()}] -- Request [${req.method}] [${req.url}]`)
  next();
}

app.use(loggerMiddleware)

app.get("/", (req, resp) => {
  resp.send("Hello Welcome")
})

app.get("/user", (req, resp) => {
  resp.send("Hello User")
})

app.listen(3000);
```



```
PS E:\FreelanceTrg\Node.js\Demo\expresslocal> node .\express7-middleware-logger.js
Mon Jun 17 2024 17:29:29 GMT+0530 (India Standard Time) -- Request [GET] [/]
Mon Jun 17 2024 17:29:43 GMT+0530 (India Standard Time) -- Request [GET] [/user]
```

```

var express = require('express');
var app = express();
var router = express.Router();

const fakeAuth = (req, resp, next) => {
  const authStatus = false (/or true)
  if (authStatus) {
    console.log("Auth status : ", authStatus)
    next();
  }
  else {
    resp.status(401)
    throw new Error("User not authorized")
  }
}
router.use(fakeAuth)

router.get('/', function (req, res) {
  res.json({ message: "Get all users" })
});

router.post('/', function (req, res) {
  res.json({ message: "Create new user" })
});

app.use('/', router);

app.listen(3000);

```

Router level middleware

```

PS E:\FreelanceTrg\Node.js\Demo\expresslocal>
Auth status : true

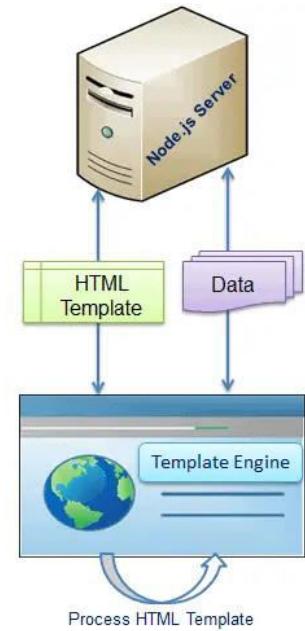
< → ⌂ ⓘ localhost:3000
{
  message: "Get all users"
}

< → ⌂ ⓘ localhost:3000
Error: User not authorized
  at fakeAuth (E:\FreelanceTrg\Node.js\Demo\expresslocal\index.js:12:11)
  at Layer.handle [as handle_request] (E:\FreelanceTrg\Node.js\Demo\expresslocal\node_modules\express\lib\router\layer.js:95:5)
  at trim_prefix (E:\FreelanceTrg\Node.js\Demo\expresslocal\node_modules\express\lib\router\index.js:317:13)
  at Object.router (E:\FreelanceTrg\Node.js\Demo\expresslocal\node_modules\express\lib\router\index.js:42:12)
  at Function.create (E:\FreelanceTrg\Node.js\Demo\expresslocal\node_modules\express\lib\application.js:51:9)
  at E:\FreelanceTrg\Node.js\Demo\expresslocal\index.js:10:5)
  at Object. (E:\FreelanceTrg\Node.js\Demo\expresslocal\index.js:14:3)
  at Module._compile (internal/modules/cjs/loader.js:955:30)
  at Object.Module._extensions..js (internal/modules/cjs/loader.js:989:10)
  at Module.load (internal/modules/cjs/loader.js:836:32)
  at Function.Module._load (internal/modules/cjs/loader.js:774:12)
  at Function.executeUserEntryPoint [as runMain] (internal/main/module_main_entry.js:4:9)

```

Views, templates, template engines

- A template engine facilitates you to use static template files in your applications.
 - At runtime, it replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client.
 - This approach makes it easier to design an HTML page.
 - Some popular template engines that work with Express are [Pug](#), [Mustache](#), [Haml](#), [EJS](#), [Jade](#), [Vash](#), [Handlebars](#) etc
- Using EJS:
 - install ejs : `npm install ejs --save`
 - Create a folder called “views” in main project folder



```
var express = require("express");
var path = require("path");

var app = express();

app.set("views", path.resolve(__dirname, "views"));
app.set("view engine", "ejs");
```

Tells Express that your views will be in a folder called views

Tells Express that you're going to use the EJS templating engine

Next gen Javascript : Javascript module

- As our application grows bigger, we want to split it into multiple files, so called “modules”.
 - A module may contain a class or a library of functions for a specific purpose.
 - A module is just a .js file
 - `export` keyword labels variables and functions that should be accessible from outside the current module.
 - `import` allows the import of functionality from other modules.

```
// module.js
export function greetUser(name) {
    return `Hello ${name}`;
}

// exporting a variable
export const name = 'Shrilata';

// exporting a function
export function sum(x, y) {
    return x + y;
}
```

```
//TestModule.js
import { greetUser, name, sum } from "./module.js";

console.log(name)
console.log(greetUser("Shrilata"))
console.log("Sum is ", sum(10,20))
```

To run this code, first create package.json and add a property : "type": "module"

```
PS E:\FreelanceTraining\ReactJS\Next-gen-JS-demos> node .\TestModule.js
Shrilata
Hello Shrilata
Sum is 30
```

Export default

```
//def_module.js
function greet(){
    return "Hello User"
}
export default greet;
```

```
//TestModule.js
import helloFunc from "./def_module.js"
console.log(helloFunc()) //Hello User
```

```
//MathModule.js
export function sum(a,b){
    return a + b
}

export function mul(a,b){
    return a * b
}

export const PI = 3.14
```

```
import * as math from "./MathModule.js"
console.log(math.sum(math.PI, math.PI)) //6.28

console.log(math.mul(10,20)) //200
```

While performing default export,

“module” is imported from greet.js. Since, module is not in greet.js, the default export (greet() in this case) is exported as “module”.

You can directly use the default export without enclosing curly brackets {}.

A file can contain multiple exports. However, you can only have one default export in a file.

Next gen Javascript : Classes, properties and methods

- To declare a class, you use the `class` keyword with the name of the class
- There can only be one "constructor" in a class; `SyntaxError` will be thrown if the class contains more than one occurrence of a constructor method.

```
class Rectangle {  
    constructor(height, width) {  
        this.height = height;  
        this.width = width;  
    }  
}
```

constructor method is
always defined with the
name "constructor"

Classes can have methods, which
defined as functions, albeit without
needing to use the function
keyword.

```
class Rectangle {  
    constructor(height, width) {  
        this.height = height;  
        this.width = width;  
    }  
  
    // Method  
    calcArea() {  
        return this.height * this.width;  
    }  
  
    const square = new Rectangle(10, 10);  
  
    console.log(square.calcArea()); // 100
```

Next gen Javascript : Class inheritance

```
class Person{
    fname = "Anil";
    lname = "Patil";
    getFullName = () => this.fname + " " + this.lname;
}

const p1 = new Person();
console.log(p1.getFullName()); //Anil Patil
```

```
class Stud{
    constructor(name){
        this.sname=name;
    }
    getName(){
        console.log(this.sname); //sunita
    }
}
const s1 = new Stud("sunita");
s1.getName();
```

```
class GradStud extends Stud{
    constructor(){
        super("Kavita");
        this.gpa = 4.9;
    }
    getDetails(){
        super.getName(); //Kavita
        console.log("GPA : " + this.gpa);
    }
}

const gs = new GradStud();
gs.getDetails();
```

Next gen Javascript : Spread and rest operators

- **spread:**

- The spread operator is three dots (...) that perform several different tasks. The spread operator allows us to combine the contents of arrays. And Objects too

```
var peaks = ["Tallac", "Ralston", "Rose"]
var canyons = ["Ward", "Blackwood"]
var tahoe = [...peaks, ...canyons]
console.log(tahoe.join(', '))
// Tallac, Ralston, Rose, Ward, Blackwood
```

```
const arr = [2, 8, 15, 4];
Math.max(...arr); // 15
Math.min(...arr); // 2
```

```
const nos = [1,2,3]
const nos1 = [nos,4]
console.log(nos1) // [ [ 1, 2, 3 ], 4 ]
const nos2 = [...nos, 4]
console.log(nos2) // [ 1, 2, 3, 4 ]
```

We can use for loop to loop through each value of the array and find the max or min value. But using a spread operator we can do it in a single line.

```
var enhancedUser = {
  isAdmin : true,
  ...user
}
console.log(enhancedUser)
// { isAdmin: true, uname: 'Soha', email: 'soha@gmail.com' }
```

```
var user = {
  uname : "Soha",
  email : "soha@gmail.com"
}
```

Next gen Javascript : Spread and rest operators

- rest operator : used to merge a list of function arguments into an array

```
//rest
function sortArgs(...args){
  console.log(args.sort());
}

sortArgs(5,2,7,3,9,1)                                     >
[1, 2, 3, 5, 7, 9]

function evenNos(...args){
  console.log(args.filter(ele => (ele%2 ==0)));
}

evenNos(5,2,7,4,9,1,8)                                     [2, 4, 8]
```

Next gen Javascript : Destructuring arrays and objects

- Destructuring : allows to easily extract array elements or object properties and store them in variables
 - Allows you to do in a single line, what would otherwise require multiple lines
 - We can assign values to multiple variables in one line with array destructuring.

```
//Longhand  
let a, b, c;  
a = 5;  
b = 8;  
c = 12;
```

```
//Shorthand  
let [a, b, c] = [5, 8, 12];
```

```
//Destructuring arrays  
const name = ["Anil", "Patil"]  
// To access , we use name[0] and name[1]
```

```
//Instead we can do this:  
const [fname, lname] = ["Anil", "Patil"]  
console.log(fname, lname) //Anil Patil
```

```
var rect = { x: 0, y: 10, width: 15, height: 20 };  
// Destructuring assignment  
var {x, y, width, height} = rect;  
console.log(x, y, width, height); // 0,10,15,20
```

```
const arr=[1,2,3,4,5]  
var [a,b] = arr  
console.log(a,b) //1,2
```

Next gen Javascript : Destructuring

- To swap two variables, we often use a third variable. We can swap two variables easily with array destructuring assignment.

```
let x = 'Hello', y = 55;  
//Longhand  
const temp = x;  
x = y;  
y = temp;  
  
//Shorthand  
[x, y] = [y, x];
```

- Object destructuring:

- If the variable name and object key name are the same then we can just mention the variable name in object literals instead of both key and value.
- JavaScript will automatically set the key same as the variable name and assign the value as a variable value.

```
let firstname = 'Amitav';  
let lastname = 'Mishra';  
//Longhand  
let obj = {firstname: firstname, lastname: lastname};  
//Shorthand  
let obj = {firstname, lastname};
```

Array functions

- **Array.filter**
 - You can filter arrays by using the `.filter(callback)` method.
 - The result will be another array that contains 0 or more elements based on the condition (or the "check") that you have in the callback.

```
var nums = [1, 2, 3, 21, 22, 30];
var evens = nums.filter(i => i % 2 == 0);
```

```
const grades = [10, 2, 21, 35, 50, -10, 0, 1];

//get all grades > 20
const result = grades.filter(grade => grade > 20); // [21, 35, 50];

// get all grades > 30
grades.filter(grade => grade > 30); // ([35, 50])
```

Array functions

- `Array.map()` : returns a new array containing the result of invoking the callback function for every item in the array.

```
const numbers = [1, 2, 3];

const doubleNumArray = numbers.map((num) => {
  return num * 2;
});

console.log(numbers);
console.log(doubleNumArray);
```

[1, 2, 3]

[2, 4, 6]

> 1

```
var numbers = [1, 4, 9];
var roots = numbers.map(Math.sqrt);
console.log("roots is : " + roots ); //1,2,3
```

```
var numbers = [1, 2, 3, 4];
var doubled = numbers.map(i => i * 2);
var doubled = [for (i of numbers) i * 2]; //same as above
console.log(doubled); // logs 2,4,6,8
```

Next gen Javascript

- Template strings provide us with an alternative to string concatenation. They also allow us to insert variables into a string.
 - Traditional string concatenation uses plus signs or commas to compose a string using variable values and strings.
 - `console.log(lastName + ", " + firstName + " " + middleName)`
- With a template, we can create one string and insert the variable values by surrounding them with `${variable}`.
 - `console.log(` ${lastName}, ${firstName} ${middleName}`)`
 - Any JavaScript that returns a value can be added to a template string between the `${ }` in a template string.

```
//Javascript : generating an html string
var msg1 = 'Have a great day';
var html = '<div>' + msg1 + '</div>';
document.write(html)

//Using template strings
var msg2 = 'Never give up';
var html1 = `<div>${msg2}</div>`;
document.write(html1)
```

REACT.JS

JavaScript library for building user interfaces

- HTML
- CSS
- JavaScript with ES6 preferably
- Basic understanding of how to use npm.

What is React?

- From the official React page : A JavaScript library for building user interfaces
- Its not a framework; React does only one thing – create awesome UI!
- React is used to build single page applications.
- React.js is a JavaScript library. It was developed by engineers at Facebook.
- React is a declarative, efficient, and flexible JavaScript library for building user interfaces.
- It lets you compose complex UIs from small and isolated pieces of code called “components”.
- React : released by Facebook in 2013. By this point, FB had already been using React to solve many of its problems internally.
 - React itself is not technically a framework; it's a library for rendering UI components.
 - React is used in combination with other libraries to make applications — React and React Native enable developers to make mobile applications; React and ReactDOM enable them to make web applications, etc.

Components

- A **Component** is one of the core building blocks of React.
- Its just a **custom HTML element!**
 - Components are just HTML code, CSS code for styling and some JS code for logic
 - Advantages of components? Components are **reusable**, so same component can be used again and again in same UI
- Every application you will develop in React will be made up of pieces called components.
 - Components make the task of building UIs much easier. You can see a UI broken down into multiple individual pieces called components and work on them independently and merge them all in a parent component which will be your final UI.

Requirements

- Ensure that NodeJS and typescript are installed
 - You'll need to have Node ≥ 22 on your local development machine
 - Install TypeScript as follows: `npm install -g typescript`

```
Administrator: C:\WINDOWS\system32\cmd.exe
C:\Windows\System32>node -v
v22.21.0

C:\Windows\System32>npm -v
10.9.4

C:\Windows\System32>tsc -v
Version 5.9.3
```

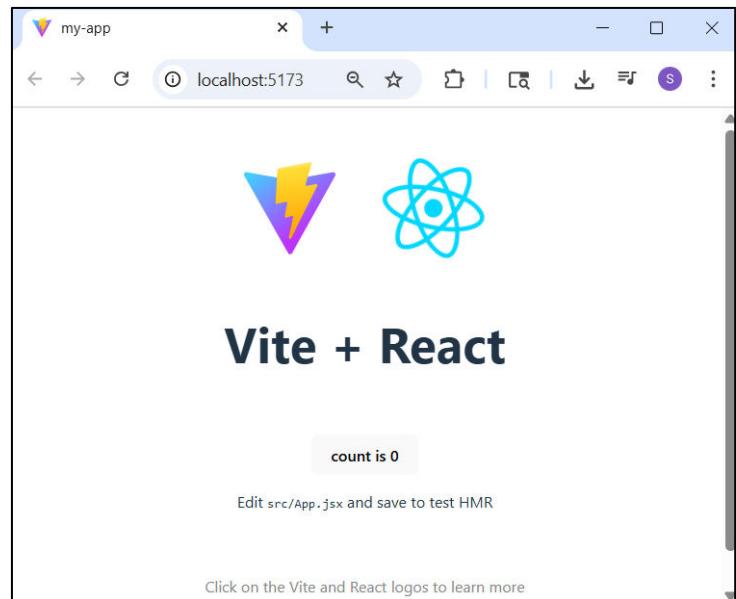
Using Vite

- Visit <https://vite.dev/guide/>

npm create vite@latest

```
> npx
> create-vite

| 
|  Project name:
| my-app
|
|  Select a framework:
| React
|
|  Select a variant:
| JavaScript
|
|  Use rolldown-vite (Experimental)
| Yes
|
 Install with npm and start now?
 Yes /  No
```



```
PS E:\FreelanceTraining\ReactJS\Demo-Sep2025\my-app> npm run dev

> my-app@0.0.0 dev
ROLLDOWN-VITE v7.1.14 ready in 225 ms

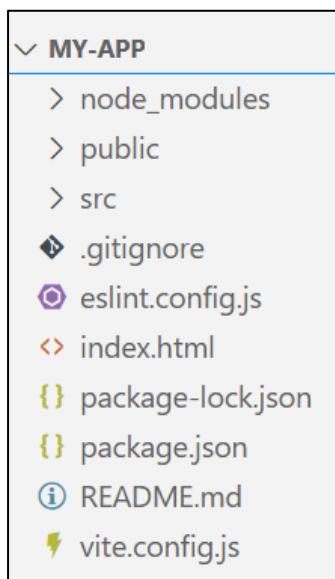
→ Local: http://localhost:5173/
```

```
npm create vite@latest my-vite-react-app -- --template react
```

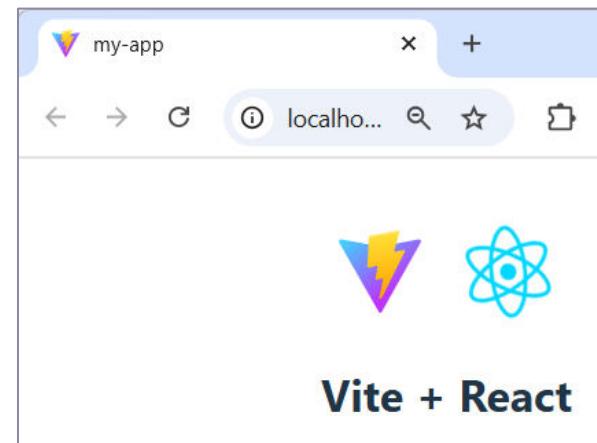
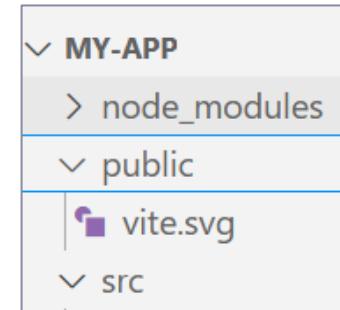
Rolldown is a modern, high-performance JavaScript bundler written in Rust. It aims to provide significant performance improvements while maintaining compatibility with the existing ecosystem.

Understanding the folder structure

- **public/**: This folder contains static assets that are served directly.
- **src/**: This is where your React components and application logic reside. By default, you'll find an **App.jsx** and **main.jsx**.
- **index.html**: The main HTML file that serves your application.
- **vite.config.js**: This configuration file allows you to customize Vite's behavior.



```
▶ Debug
"scripts": [
  "dev": "vite",
  "build": "vite build",
  "lint": "eslint .",
  "preview": "vite preview"
},
"dependencies": {
  "react": "^19.1.1",
  "react-dom": "^19.1.1"
},
```



Index.html

To ensure proper rendering and touch zooming, add this <meta> tag

- The width=device-width part sets the width of the page to follow the screen-width of the device (which will vary depending on the device).
- The initial-scale=1 part sets the initial zoom level when the page is first loaded by the browser.

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>my-app</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

tells the browser to load the main JSX file as a module, which allows for modern JavaScript features and module imports to be used directly in the browser.

- The root node is the HTML element where you want to display the result.
- It is like a container for content managed by React.
- It does NOT have to be a <div> element and it does NOT have to have the id='root'

Understanding the folder structure

```
MY-APP
  node_modules
  public
    vite.svg
  src
    assets
    App.css
    App.jsx
    index.css
    main.jsx
```

```
function App() {
  const [count, setCount] = useState(0)

  return (
    <>
      <div>
        <a href="https://vite.dev" target="_blank">
          | <img src={viteLogo} className="logo" alt="Vite logo" />
        </a>
        <a href="https://react.dev" target="_blank">...
        </a>
      </div>
      <h1>Vite + React</h1>
      <div className="card">
        <button onClick={() => setCount((count) => count + 1)}>
          | count is {count}
        </button>
        <p>...
        </p>
      </div>
      <p className="read-the-docs">
        | Click on the Vite and React logos to learn more
      </p>
    </>
  )
}

export default App
```

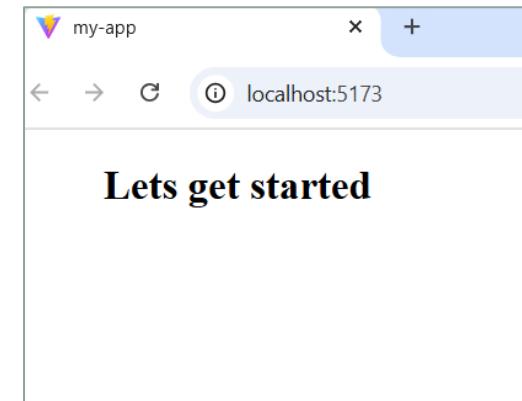
```
/* App.css */
.App {
  margin-left: 50px;
}
```

```
App.jsx > ...
import './App.css'

function App() {
  return (
    <>
      <div className='App'>
        <h2>Lets get started</h2>
      </div>
    </>
  )
}

export default App
```

```
<!DOCTYPE html>
<html lang="en">
  <head> ... </head>
  <body>
    <div id="root">
      <div class="App" > == $0
        <h2>Lets get started</h2>
      </div>
    </div>
    <script type="module" src="/src/main.jsx" >
  </body>
</html>
```



Understanding JSX

- Eg : const mytag = <h1>Hello React!</h1>;

```
const myelement = <h1>Understanding JSX!</h1>;
```

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

```
const myelement = (  
  <ul>  
    <li>Apples</li>  
    <li>Bananas</li>  
    <li>Cherries</li>  
  </ul>  
);  
ReactDOM.render(myelement, document.getElementById('root'));
```

To write HTML on multiple lines, put the HTML inside parentheses()

```
const myelement = (  
  <div>  
    <h1>I am a Header.</h1>  
    <h1>I am a Header too.</h1>  
  </div>  
);  
ReactDOM.render(myelement, document.getElementById('root'));
```

If we want to return more elements, we need to wrap it with one container element. Notice how we are using div as a wrapper for the two h1 elements.

JSX restrictions

```
function App() {
  return (JSX attribute) React.HTMLAttributes<HTMLDivElement>.className?: string
    <div className="App">
      <h2>Welcome to React!</h2>
    </div>
  );
}
export default App;
```

```
▼<div class="App">
  <h1> Hi, welcome to React</h1>
</div>
```

JSX clearly looks like HTML and it should allow us to write HTML-ish code in our JS file. But, since it is in a JS file and it IS a JS code, some words cant be used.

Eg, ‘class’ that we use normally use in HTML for bringing in CSS cant be used, since class is a ES6 feature too! And we can create React class components using “class” keyword

JSX vs. JavaScript (Only for Vite)

- The .js file is a file that contains standard JavaScript code.
 - These files do not contain the UI related code and are used for writing the logics and the functions.
 - Used for regular JavaScript code and logic.
 - Does not include JSX (HTML-like syntax).
 - Can be used for utility functions, state management, or simple React components.
 - No extra tools needed to run.
- JSX (JavaScript XML) is an extension of JavaScript that lets developers write HTML-like code inside JavaScript.
 - Files with the .jsx extension usually contain components written in JSX, which React then converts into JavaScript function calls.
 - Used for React components with JSX (HTML-like syntax).
 - Includes JSX to define the UI structure.
 - Mainly for creating and rendering the user interface.
 - Requires tools like Babel to convert JSX into JavaScript.

Creating a functional component

- Components are the essential building blocks of any app created with React
 - A single app most often consists of many components.
- A component is in essence, a piece of the UI - splitting the user interface into reusable and independent parts, each of which can be processed separately.
 - Component is independent and reusable bit of code.
 - It's an encapsulated piece of logic that returns HTML via a `render` function.

```
const ExpenseItem = () => {
  return <h2>Expense Item</h2>
}
export default ExpenseItem;
```

```
function ExpenseItem(){
  return <h2>Expense Item</h2>
}
export default ExpenseItem;
```

- Every React function component must follow 2 rules:
 - Function name MUST begin with capital
 - Function MUST return a “Renderable” value; ie return a value that can be rendered (displayed on screen) by React; typically returns JSX

Creating a functional component

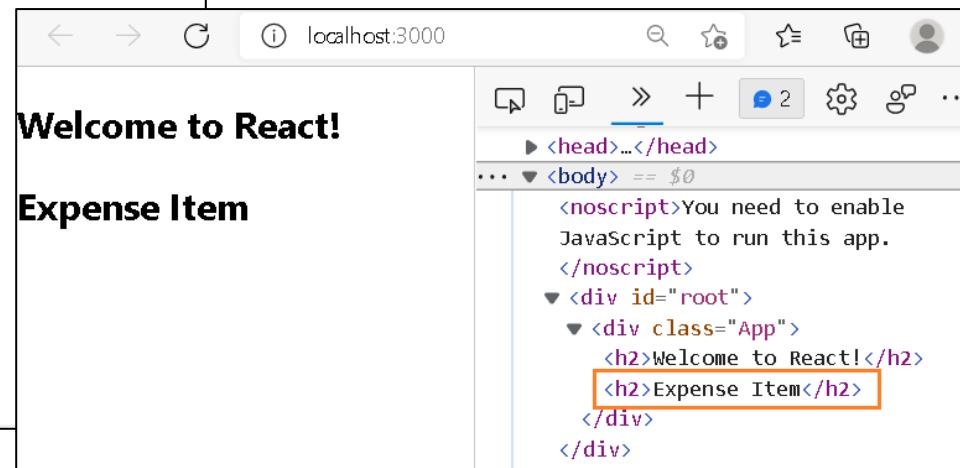
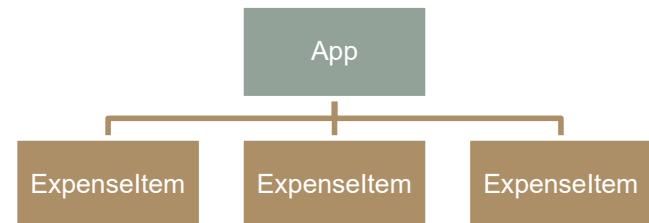
- Adding our component to main component

App.js > ...

```
import React from 'react';
import ExpenseItem from './components/ExpenseItem';

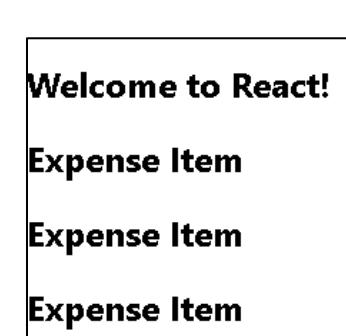
function App() {
  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <ExpenseItem />
    </div>
  );
}

export default App;
```



- Creating components makes them reusable and configurable.
- Reusing is simple. Eg, simply copy paste `<ExpenseItem/>` multiple times in App.js.

```
function App() {
  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <ExpenseItem />
      <ExpenseItem />
      <ExpenseItem />
    </div>
  );
}
```



Another example

> OPEN EDITORS

✓ MY-APP

> node_modules

> public

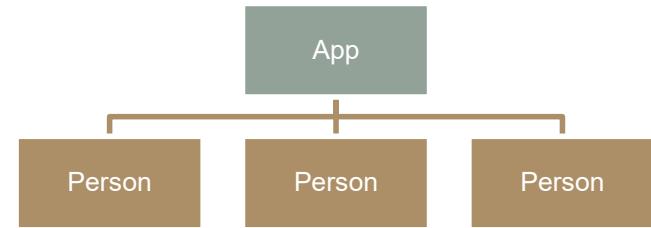
✓ SRC

> Person

JS Person.js U

```
src > Person > JS Person.js > [?] default
  1   import React from 'react';
  2
  3   const person = () => {
  4     return <p>Hi Person</p>
  5   }
  6   export default person;
```

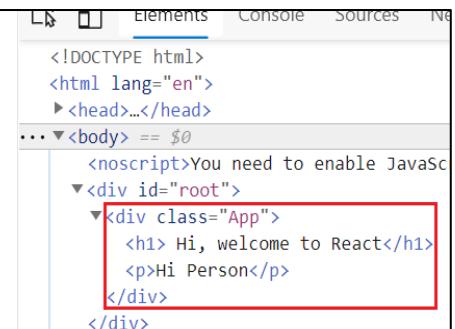
```
import React, {Component} from 'react';
import './App.css';
import Person from './Person/Person';
function App() {
  return (
    <div className="App">
      <h1> Hi, welcome to React</h1>
      <Person />
    </div>
  );
}
export default App;
```



```
return (
  <div className="App">
    <h1> Hi, welcome to React</h1>
    <Person />
    <Person />
    <Person />
  </div>
);
```

Hi, welcome to React

Hi Person



Components & JSX

- When creating components, you have the choice between two different ways:
- **Functional components** (also once referred to as "presentational", "dumb" or "stateless" components)

```
const cmp = () => {
  return <div>some JSX</div>
}
```

- **class-based components** (also referred to as "containers", "smart" or "stateful" components)

```
class Cmp extends Component {
  render () {
    return <div>some JSX</div>
  }
}
```

Making our functional component more complex

```
import "./ExpenseItem.css"

const ExpenseItem = () => {
  return (
    <div className="expense-item">
      <div>Oct 20th 2021</div>
      <div className="expense-item__description">
        <h2>Paid Carpenter</h2>
        <p className="expense-item__price">Rs 75000</p>
      </div>
    </div>
  )
}

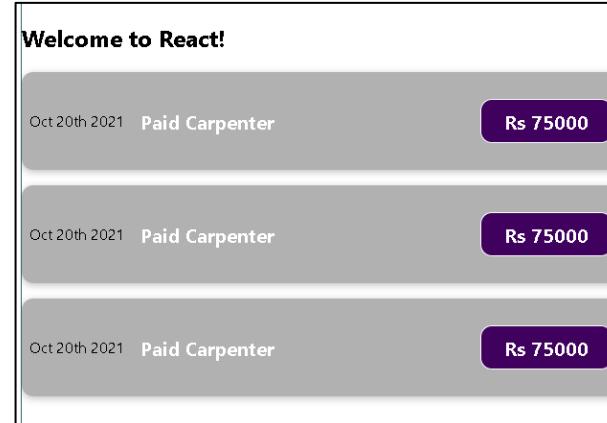
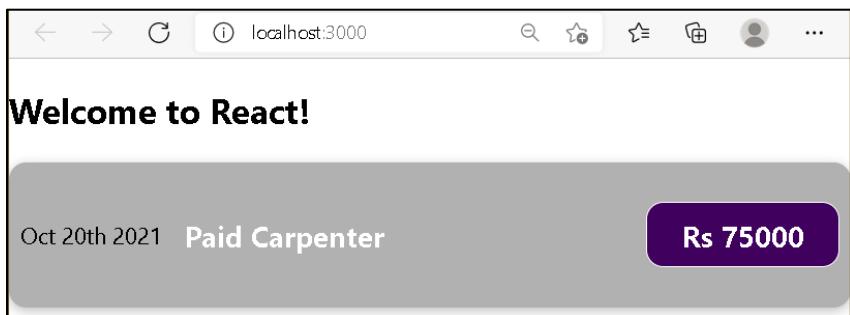
export default ExpenseItem;
```

```
.expense-item {
  display: flex;
  justify-content: space-between;
  align-items: center;
  box-shadow: 0 2px 8px rgba(0, 0, 0, 0.1);
  padding: 0.5rem;
  margin: 1rem 0;
  border-radius: 12px;
  background-color: #4b4b4b6e;
}

.expense-item__description { ... }

.expense-item h2 { ... }

.expense-item__price { ... }
```



Outputting dynamic content

- If we have some dynamic content in our jsx part which we want to run as JavaScript code and not interpret as text, we have to wrap it in **single curly braces**.

```
import "./ExpenseItem.css"

const ExpenseItem = () => {

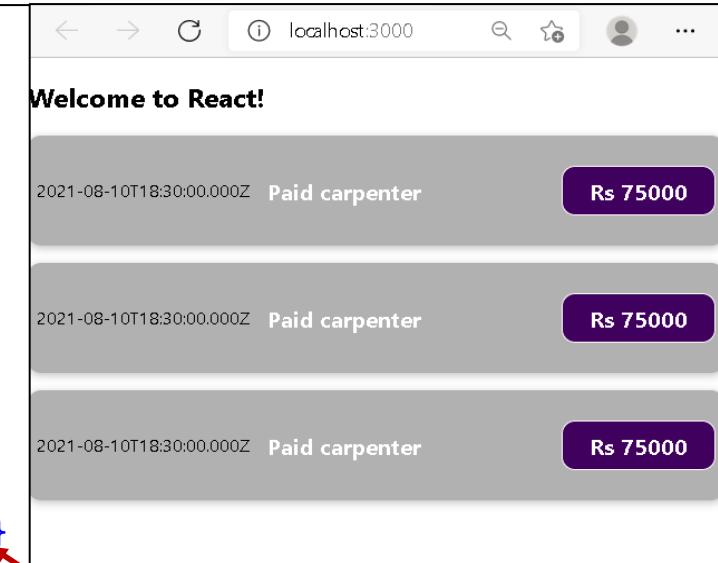
  const expDate = new Date(2021, 7, 11);
  const expTitle = "Paid carpenter";
  const expAmount = 75000

  return (
    <div className="expense-item">

      /* single and multiline comments in JSX */

      <div>{expDate.toISOString()}</div>
      <div className="expense-item__description">
        <h2>{expTitle}</h2>
        <p className="expense-item__price">Rs {expAmount}</p>
      </div>
    </div>
  )
}

export default ExpenseItem;
```



Comments in JSX

Wrap dynamic content in
JSX in {...}

Outputting dynamic content – another example

```
//Person.js
const person = () => {
  return <p>Hi Person i am {Math.floor(Math.random() * 30)} years old</p>
}
export default person;
```

Wrap dynamic content in JSX in {...}

```
function App() {
  return (
    <div className="App">
      <h1> Hi, welcome to React</h1>
      <Person />
      <Person />
      <Person />
    </div>
  );
}
```

Math.random() returns a random number between 0 and 1.
Math.floor(Math.random() * 30) Returns a random integer from 0 to 29

Hi, welcome to React

Hi Person i am 27 years old

Hi Person i am 29 years old

Hi Person i am 12 years old

PROPS

Passing data via 'props'

Parent component

Child
component

Parent

data flow

Child

- “props” stands for properties.
 - It is a special keyword in React used for passing data from one component to another.
 - Props are arguments passed into React components.
 - props are read-only. So, the data coming from a parent component can't be changed by the child component.
 - Props are passed to components via HTML attributes.
 - Props can be used to pass any kind of data such as: String, Array, Integer, Boolean, Objects or, Functions

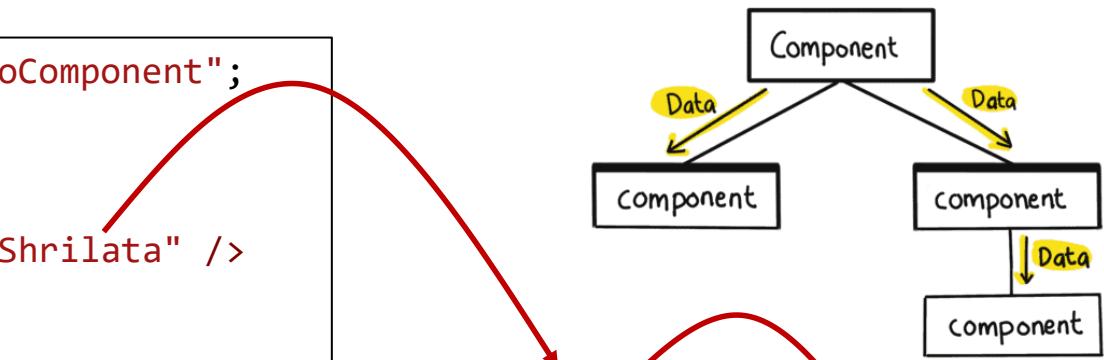
```
import HelloComponent from "./HelloComponent";
const MessageComponent = () => {
  return(
    <div>
      <HelloComponent name="Shrilata" />
    </div>
  );
}
export default MessageComponent;
```

← → ⌂ localhost:3000/?

Welcome to React!

Hello, welcome Shrilata

```
const HelloComponent = (props) => {
  return (<h3>Hello, welcome {props.name}</h3>)
}
export default HelloComponent;
```



Passing data via 'Props'

```
function App() {  
  return (  
    <div className="App">  
      <h2>Welcome to React!</h2>  
      <ExpenseItem expDate="20-12-2020" expTitle="Myntra shopping" expAmount="2500"/>  
      <ExpenseItem expDate="21-12-2020" expTitle="Microwave" expAmount="8000"/>  
    </div>  
  );  
}  
export default App;
```

you can use as many props as you like

```
const ExpenseItem = (props) => {  
  return (  
    <div className="expense-item">  
      <div>{props.expDate}</div>  
      <div className="expense-item__description">  
        <h2>{props.expTitle}</h2>  
        <p className="expense-item__price">Rs {props.expAmount}</p>  
      </div>  
    </div>  
  )  
}  
export default ExpenseItem;
```

Welcome to React!

20-12-2020 Myntra shopping

Rs 2500

21-12-2020 Microwave

Rs 8000

React uses 'props' to pass attributes from 'parent' component to 'child' component.

Working with props

```
function App() {  
  return (  
    <div className="App">  
      <h1> Hi, welcome to React</h1>  
      <Person name="Shri" age="20"/>  
      <Person name="Soha" age="23">Hobbies : Coding</Person>  
      <Person name="sandeep" age="45"/>  
    </div>  
  );  
}
```

Hi, welcome to React

Hi i am Shri and i am 20 years old

Hi i am Soha and i am 23 years old

Hi i am sandeep and i am 45 years old

```
//Person.js  
import React from 'react';  
  
const person = (props) => {  
  return <p>Hi i am {props.name} and i am {props.age} years old</p>  
}  
export default person;
```

React uses 'props' to pass attributes from 'parent' component to 'child' component.

```
const Person = ({name, age}) => {  
  return <p>Hi i am {name} and i am {age} years old</p>  
}
```

If prop names match parent attributes, you can use object destructuring instead of "props".

Working with props

```
function App() {
  const expenses = [
    {
      title: 'Groceries',
      amount: 900,
      date: new Date(2020, 7, 14),
    },
    { title: 'New TV', amount: 34000, date: new Date(2021, 2, 12) },
    { title: 'SofaSet', amount: 25000, date: new Date(2021, 2, 28),
    }
  ];
  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <ExpenseItem expDate={expenses[0].date} expTitle={expenses[0].title}
                    expAmount={expenses[0].amount}/>
      <ExpenseItem expDate={expenses[1].date} expTitle={expenses[1].title}
                    expAmount={expenses[1].amount}/>
      <ExpenseItem expDate={expenses[2].date} expTitle={expenses[2].title}
                    expAmount={expenses[2].amount}/>
    </div>
  );
}
export default App;
```

Welcome to React!

2020-08-13T18:30:00.000Z Groceries

Rs 900

2021-03-11T18:30:00.000Z New TV

Rs 34000

2021-03-27T18:30:00.000Z SofaSet

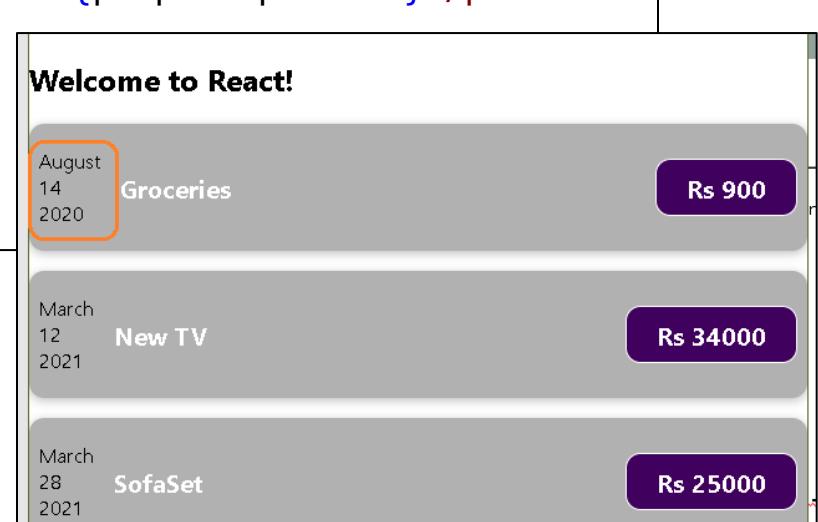
Rs 25000

“Javascript” in components

```
const ExpenseItem = (props) => {
  const month = props.expDate.toLocaleString('en-US', {month: 'long'});
  const day = props.expDate.toLocaleString('en-US', {day: '2-digit'});
  const year = props.expDate.getFullYear();

  return (
    <div className="expense-item">
      <div>
        <div>{month}</div>
        <div>{day}</div>
        <div>{year}</div>
      </div>
      <div className="expense-item__description">
        <h2>{props.expTitle}</h2>
        <p className="expense-item__price">Rs{props.expAmount}</p>
      </div>
    </div>
  )
}

export default ExpenseItem;
```



```

import ExpenseDate from "./ExpenseDate";
const ExpenseItem = (props) => {
  return (
    <div className="expense-item">
      <ExpenseDate date={props.expDate}/>
      <div className="expense-item__description">
        <h2>{props.expTitle}</h2>
        <p className="expense-item__price">Rs {props.expAmount}</p>
      </div>
    </div>
  )
}
export default ExpenseItem;

```

Splitting components further

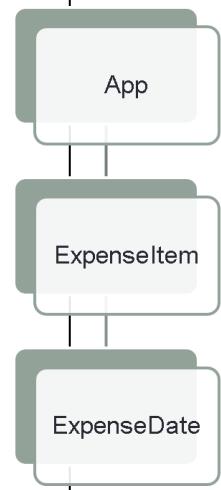
| | | |
|----------------------|-----------|----------|
| August 14 2020 | Groceries | Rs 900 |
| March 12 2021 | New TV | Rs 34000 |
| March 28 2021 | SofaSet | Rs 25000 |

```

import "./ExpenseDate.css"

const ExpenseDate = (props) => {
  const month = props.date.toLocaleString('en-US', {month: 'long'});
  const day = props.date.toLocaleString('en-US', {day: '2-digit'});
  const year = props.date.getFullYear();
  return (
    <div className="expense-date">
      <div className="expense-date__month">{month}</div>
      <div className="expense-date__day">{day}</div>
      <div className="expense-date__year">{year}</div>
    </div>
  );
}
export default ExpenseDate;

```



Splitting components

- As your application grows, it becomes even more crucial to divide it into smaller components.
- Each component should encapsulate a specific part of the application's functionality.
 - These components should have a single responsibility, i.e., they should accomplish one thing only.
 - The benefit is that these components can be easily reused and tested.
 - For instance, if you were to create a blog application, you might divide your app into components like Header, Footer, ArticleList, Article, Comment, and so on.

```
function Header() {  
  // ...Header implementation... }  
  
function ArticleList() {  
  // ...ArticleList  
implementation... }  
  
function Article() {  
  // ...Article implementation... }  
  
function Comment() {  
  // ...Comment implementation... }  
  
function Footer() {  
  // ...Footer implementation... }  
  
function Blog() {  
  return (  
    <div>  
      <Header />  
      <ArticleList>  
        <Article />  
        <Comment />  
      </ArticleList>  
      <Footer />  
    </div>  
  );  
}
```

EVENTS AND EVENT HANDLING

Listening to events and working with event handlers

```
const ExpenseItem = (props) => {  
  
  let btnHandler = () => {  
    console.log("Button clicked!")  
  }  
  
  return (  
    <div className="expense-item">  
      <ExpenseDate date={props.expDate}/>  
      <div className="expense-item__description">  
        <h2>{props.expTitle}</h2>  
        <p className="expense-item__price">Rs {props.expAmount}</p>  
      </div>  
      <button onClick={btnHandler}>Change Title</button>  
    </div>  
  )  
}  
export default ExpenseItem;
```

No parenthesis ()



STATEFUL COMPONENTS

React State

- The state is a built-in React object that is used to contain data or information about the component.
- A component's state can change over time; whenever it changes, the component re-renders.
 - The change in state can happen as a response to user action or system-generated events and these changes determine the behavior of the component and how it will render.
- A component with state is known as stateful component.
- State allows us to create components that are dynamic and interactive.
 - State is private, it must not be manipulated from the outside.
 - Also, it is important to know when to use 'state', it is generally used with data that is bound to change.

Component without state

```
const ExpenseItem = (props) => {

  let title = props.expTitle;

  let btnHandler = () => {
    title = "updated expense"
    console.log("Button clicked!")
  }

  return (
    <div className="expense-item">
      <ExpenseDate date={props.expDate}/>
      <div className="expense-item__description">
        <h2>{title}</h2>
        <p className="expense-item__price">Rs {props.expAmount}</p>
      </div>
      <button onClick={btnHandler}>Change Title</button>
    </div>
  )
}

export default ExpenseItem;
```

React Hooks

- React Hooks are special functions provided by React to handle a specific functionality inside a React functional component.
 - Hooks allow us to "hook" into React features such as state and lifecycle methods
 - React provides *useState()* function to manage state in a functional component.
- You must import Hooks from react
 - `import React, { useState } from "react";`
 - Here - useState is a Hook to keep track of the application state.
- There are some rules for hooks:
 - Hooks can only be called inside React function components.
 - Hooks can only be called at the top level of a component.
 - Hooks cannot be conditional
 - Hooks will not work in React class components.
 - If you have stateful logic that needs to be reused in several components, you can build your own custom Hooks

Working with “state” in functional component

- The React useState Hook allows us to track state in a function component.
- To use the useState Hook, we first need to import it into our component.
 - import { useState } from "react";
 - We initialize our state by calling useState in our function component.

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

const UseStateComponent = () => {
  useState(); //hooks go here
}
```

- useState accepts an initial state and returns two values:
 1. The current state.
 2. A function that updates the state.
- Eg:

```
function FavoriteColor() {
  const [color, setColor] = useState("white");
}
```

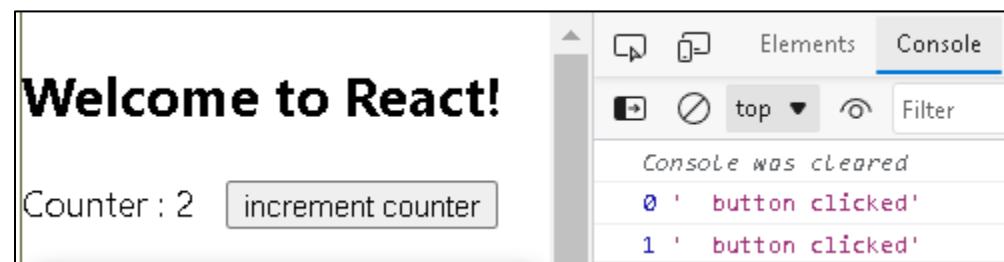
- The first value, color, is our current state.
- The second value, setColor, is the function that is used to update our state.
- Lastly, we set the initial state to a white color: useState("white")

Working with “state” in functional component

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

const UseStateComponent = () => {
  const [counter, setCounter] = useState(0); //hooks go here

  const btnHandler = () => {
    setCounter(counter+1);
    console.log(counter, " button clicked")
  }
  return(
    <div>
      Counter : {counter}&nbsp;&nbsp;
      <button onClick={btnHandler}>increment counter</button>
    </div>
  );
}
export default UseStateComponent;
```



Working with “state” in functional component

```
import React, {useState} from 'react'

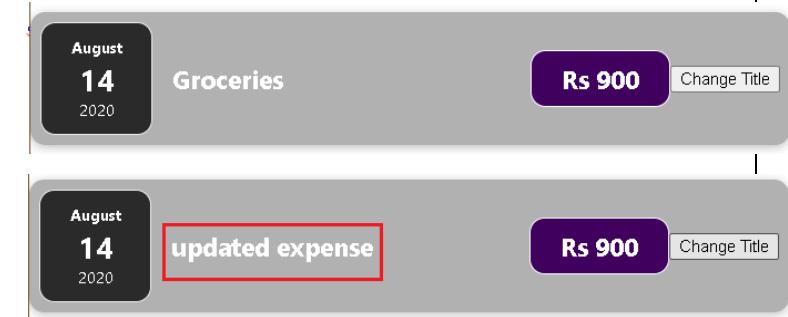
const ExpenseItem = (props) => {

  const [title, setTitle] = useState(props.expTitle);

  let btnHandler = () => {
    setTitle("updated expense")
    console.log("Button clicked!")
  }

  return (
    <div className="expense-item">
      <ExpenseDate date={props.expDate}>/>
      <div className="expense-item__description">
        <h2>{title}</h2>
        <p className="expense-item__price">Rs {props.expAmount}</p>
      </div>
      <button onClick={btnHandler}>Change Title</button>
    </div>
  )
}

export default ExpenseItem;
```



props and state

- props and state are CORE concepts of React.
- Actually, only changes in props and/ or state trigger React to re-render your components and potentially update the DOM in the browser
- **Props** : allow you to pass data from a parent (wrapping) component to a child component.
- **State** : State is used to change the component's state from within.
 - Changes to state also trigger an UI update.

- Props are immutable; once placed, they cannot be altered.
- Props data are read-only, so children of elements cannot change their parents' data
- Data and event handlers are passed to the child components using props.
- They are owned by the component which passes them to some other component.

- The data of the components that must be presented to the view is stored in State
- State is something internal and private to the component.
- State can and will change depending on the interactions with the outer world.
- State should store as simple data as possible, such as whether an input checkbox is checked or not or a CSS class that hides or displays the component

Simple example : props + state

```
import React, {useState} from 'react'
import ChildComponent from './ChildComponent';

const ParentComponent = () => {
  const [uname, setUname] = useState('Shrilata')
  const [email, setEmail] = useState('shrilata@gmail.com')

  return(
    <ChildComponent uname={uname} email={email} />
  );
}
export default ParentComponent;
```

```
function App() {
  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <ParentComponent />
    ...
  )
}
```

```
const ChildComponent = (props) => {
  return(
    <div>
      <div>Name : {props.uname}</div>
      <div>Email : {props.email}</div>
    </div>
  );
}
export default ChildComponent;
```

Welcome to React!

Name : Shrilata
Email : shrilata@gmail.com

USING FORM FOR INPUT

React forms

- Forms are a crucial component of React web applications.
 - They allow users to directly input and submit data in components ranging from a login screen to a checkout page.
 - Since most React applications are single page applications (SPAs), or web applications that load a single page through which new data is displayed dynamically, **you won't submit the information directly from the form to a server.**
 - Instead, you'll capture the form information on the client-side and send or display it using additional JavaScript code.

To add bootstrap to style our form easily:

- `npm install bootstrap –save`
- Import Bootstrap CSS in the beginning of the src/index.js file (main.jsx in Vite)

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import 'bootstrap/dist/css/bootstrap.css';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Simple form demo (without Bootstrap)

```
import { useState } from "react"
let SimpleForm = ()=>
  let [uname, setUname] = useState("")

  let unameHandler = (event)=>{
    setUname(event.target.value)
  }

  let submitHandler = (event) => {
    event.preventDefault();
    console.log("Hello",uname)
    setUname("")
  }

  return(
    <form onSubmit={submitHandler}>
      Username : <input onChange={unameHandler} value={uname} /><br />
      <input type="submit" />
    </form>
  )
}
export default SimpleForm
```



SimpleInput.js (with Bootstrap)

```
const SimpleInput = (props) => {

  const [inputName, setInputName] = useState('')

  const inputNameHandler = (event) => {
    setInputName(event.target.value)
  }

  const formSubmitHandler = event => {
    event.preventDefault();
    console.log(inputName, " submitted")
  }

  return (
    <form onSubmit={formSubmitHandler}>
      <div className='form-group'>
        <label htmlFor='name'>Your Name</label>
        <input type='text' name='uname'
               className='form-control'
               onChange={inputNameHandler}/>
      </div>
      <div className="form-actions">
        <button className="btn btn-primary">Submit</button>
      </div>
    </form>
  );
};
```

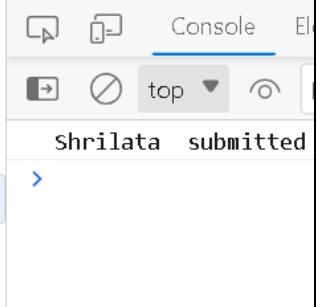
```
function App() {
  return (
    <div className="App">
      <h1>Working with forms</h1>
      <SimpleInput />
    </div>
  );
}
```

Working with forms

Your Name

Shrilata

Submit



SimpleForm with inline styling

```
let SimpleFormWithInBuiltStyle = () => {
...
return (
  <div>
    <h1 style={styles.heading}>Welcome to forms</h1>
    <h3 style={styles.subHeading}>User Input </h3>
    <form onSubmit={submitHandler} style={styles.form}>
      Username : <input onChange={unameHandler}
        style={styles.input}
        value={uname} /><br />
      <input type="submit"
        style={{ ...styles.button}}/>
    </form>
  </div>
)
}
```

Welcome to forms

User Input

Username :

Submit

```
const styles = {
  heading: {
    fontWeight: 'bold',
    fontSize: '25px',
    color: "green",
    textAlign: "center",
  },
  subHeading: {
    fontWeight: 'bold',
    fontSize: '25px',
    color: "black",
    textAlign: "center",
  },
  form: {
    backgroundColor: '#fff',
    padding: '20px',
    borderRadius: '8px',
    boxShadow: '0 2px 4px
rgba(0, 0, 0, 0.1)',
    width: '100%',
    maxWidth: '400px',
    margin: '0 auto',
  },
...
};
```

```

import { useState } from "react"

let SimpleForm1 = ()=>{

  let [uname, setUserName] = useState("")
  let [gender, setGender] = useState("")

  let unameHandler = (event)=>{
    setUserName(event.target.value)
  }

  let genderHandler = (event)=>{
    setGender(event.target.value)
  }

  let submitHandler = (event) => {
    event.preventDefault();
    let user = {
      uname : uname,
      gender : gender
    }
    console.log(user)
    setUserName("")
    setGender("")
  }
}

```

Example-2 : SimpleInput1.js with multiple form elements

```

return(
  <form onSubmit={submitHandler}>
    Username : <input onChange={unameHandler}
      value={uname} /><br />
    Gender:
    <select onChange={genderHandler}
      value={gender}>
      <option value="male">Male</option>
      <option value="female">Female</option>
    </select> <br />
    <input type="submit" />
  </form>
)
}

```

Username :

Gender:

SimpleForm1.js:22
 {uname: 'Shrilata',
 gender: 'female'}

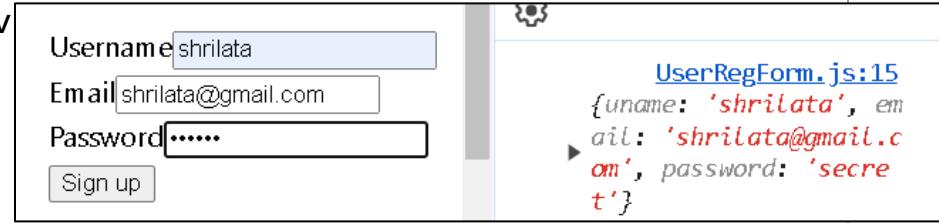
Example-3 : form with multiple form elements

```
const UserRegForm = () => {
  const initialValues = {uname:"", email:"", password:""}
  const [formValues, setFormValues] = useState(initialValues)

  let handleChange = (e) => {
    const {name, value} = e.target
    setFormValues({...formValues,[name]:v
  }

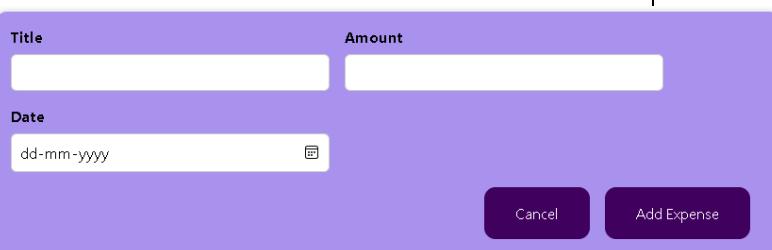
  let submitHandler = (event) => {
    event.preventDefault()
    console.log(formValues)
    setFormValues({uname:"", email:"", password:""})
  }

  return (
    <form onSubmit={submitHandler}>
      <label htmlFor="uname">Username</label>
      <input name="uname" value={formValues.uname} onChange={handleChange}/>
      <label htmlFor="email">Email</label>
      <input type="email" name="email" value={formValues.email}
             onChange={handleChange}/>
      <label htmlFor="password">Password</label>
      <input type="password" name="password" value={formValues.password}
             onChange={handleChange}/>
      <button type="submit">Sign up</button>
    </form>
  );}
```



Adding form inputs

```
const NewExpense = () => {
  return(
    <div className="new-expense">
      <ExpenseForm />
    </div>
  );
}
export default NewExpense;
```



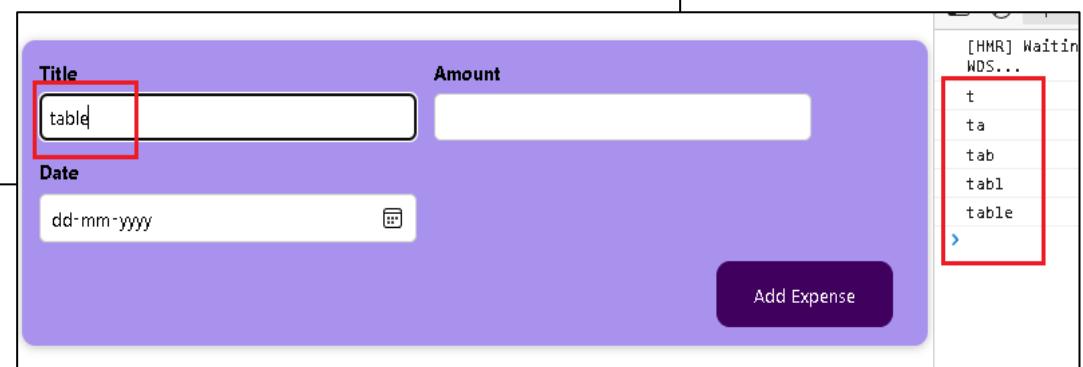
```
import "./ExpenseForm.css"
const ExpenseForm = () => {
  return(
    <form>
      <div className="new-expense__controls">
        <div className="new-expense__control">
          <label>Title</label>
          <input />
        </div>
        <div className="new-expense__control">
          <label>Amount</label>
          <input type="number"/>
        </div>
        <div className="new-expense__control">
          <label>Date</label>
          <input type="date" />
        </div>
      </div>
      <div className="new-expense__actions">
        <button type="submit">Add Expense</button>
      </div>
    </form>
  );
}
export default ExpenseForm;
```

Listening to user input in our tracking application

```
const ExpenseForm = () => {  
  
  const titleChangeHandler = (event) => {  
    console.log(event.target.value)  
  }  
  return(  
    <form>  
      <div className="new-expense__controls">  
        <div className="new-expense__control">  
          <label>Title</label>  
          <input onChange={titleChangeHandler}>/>  
        </div>  
        ...  
      </div>  
      <div className="new-expense__actions">  
        <button type="submit">Add Expense</button>  
      </div>  
    </form>  
  );  
}  
export default ExpenseForm;
```

We want to store the extracted form info somewhere

One way to ensure the value is stored and make sure that the value survives even if the component is re-executed is to use **state**



Storing input into state – working with multiple states

```
import React, {useState} from 'react';
const ExpenseForm = () => {

  const [inputTitle, setInputTitle] = useState('')
  const [inputAmount, setInputAmount] = useState('')
  const [inputDate, setInputDate] = useState('')

  const titleChangeHandler = (event) => {
    setInputTitle(event.target.value)
  }
  const amountChangeHandler = (event) => {
    setInputAmount(event.target.value)
  }
  const dateChangeHandler = (event) => {
    setInputDate(event.target.value)
  }

  return(
    <form>
      <div className="new-expense__controls">
        <div className="new-expense__control">
          <label>Title</label> <input onChange={titleChangeHandler}>
        </div>
        <div className="new-expense__control">
          <label>Amount</label>
          <input type="number" onChange={amountChangeHandler}>
        </div> ...
      </div>
    </form>
  )
}
```

Notice that we are dealing with multiple states which is normal in React.
However, we can also have just one state

Form submission

```
const ExpenseForm = () => {

  const [inputTitle, setInputTitle] = useState('')
  const [inputAmount, setInputAmount] = useState('')
  const [inputDate, setInputDate] = useState('')

  const titleChangeHandler = (event) => { ... }
  const amountChangeHandler = (event) => { ... }
  const dateChangeHandler = (event) => { ... }

  const submitHandler = (event) => {
    event.preventDefault();
  }

  return(
    <form onSubmit={submitHandler}>
      <div className="new-expense__controls"> ...
      </div>
      <div className="new-expense__actions">
        <button type="submit">Add Expense</button>
      </div>
    </form>
  );
}
```

Form submission – extracting data, 2-way binding

```
const ExpenseForm = () => {

  const [inputTitle, setInputTitle] = useState('')
  const [inputAmount, setInputAmount] = useState('')
  const [inputDate, setInputDate] = useState('')

  const titleChangeHandler = (event) => {earlier-code}
  const amountChangeHandler = (event) => {earlier-code}
  const dateChangeHandler = (event) => {...}

  const submitHandler = (event) => {
    event.preventDefault();
    const expenseData = {
      title:inputTitle,
      amount:inputAmount,
      date: new Date(inputDate)
    }
    console.log(expenseData)

    setInputAmount('')
    setInputDate('')
    setInputTitle('')
  }
}
```

The screenshot shows a simple expense form with three input fields: 'Title' (containing 'Mouse'), 'Amount' (containing '900'), and 'Date' (containing '01-12-2021'). Below the inputs is a purple 'Add Expense' button.

The screenshot shows the browser's developer tools element inspector. It highlights the 'Title' input field with a red border. A tooltip message '[HMR] Waiting for update signal from WDS...' is displayed above the form. In the bottom right corner of the browser window, there is a status bar showing the state of the component: > {title: 'Mouse', amount: '900', date: '2021-12-01'}. This indicates that the component is currently in a waiting state for an update signal from Webpack Dev Server (WDS).

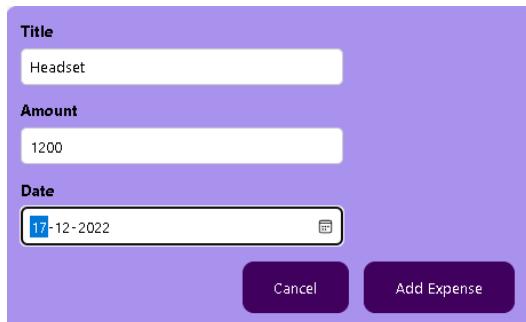
```
return(
  <form onSubmit={submitHandler}>
    <div className="new-expense__controls">
      <div className="new-expense__control">
        <label>Title</label>
        <input value={inputTitle} onChange={titleChangeHandler}/>
      </div>
      ...
    </div>
  </form>
);
```

```

const NewExpense = () => {
  const saveExpenseDataHandler = (inputExpenseData) => {
    const expenseData = {
      ...inputExpenseData,
      id:Math.random().toString()
    }
    console.log("In NewExpense ",expenseData)
  }
  return(
    <div className="new-expense">
      <ExpenseForm onSaveExpenseData={saveExpenseDataHandler}/>
    </div>
  );
}
export default NewExpense;

```

Passing data from
child to parent



```

const ExpenseForm = (props) => {
  ...
  const submitHandler = (event) => {
    event.preventDefault();
    const expenseData = {
      title:inputTitle,
      amount:inputAmount,
      date:inputDate
    }
    //console.log(expenseData)
    props.onSaveExpenseData(expenseData);
    ...
  }
}

```

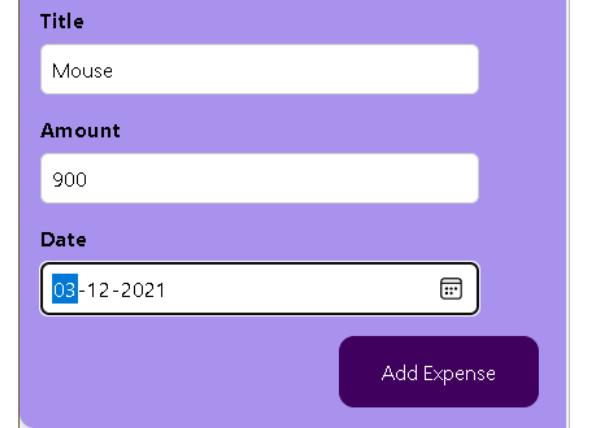
In NewExpense

[NewExpense.js:21](#)

▶ {title: 'Headset', amount: '1200', date: Sat Dec 17 2022 05:30:00 GMT+0530
(India Standard Time), id: '0.4603800892834633'}

Passing data from child to parent

```
function App() {  
  const expenses = [...];  
  
  const addExpenseHandler = expense => {  
    console.log("In App component ", expense)  
  }  
  
  return (  
    <div className="App">  
      <h2>Welcome to React!</h2>  
      <NewExpense onAddExpense={addExpenseHandler} />  
      ...  
    );  
}  
export default App;
```



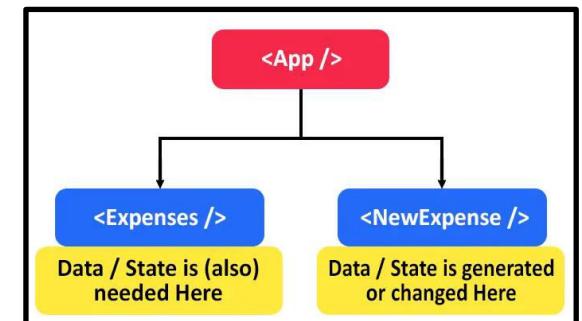
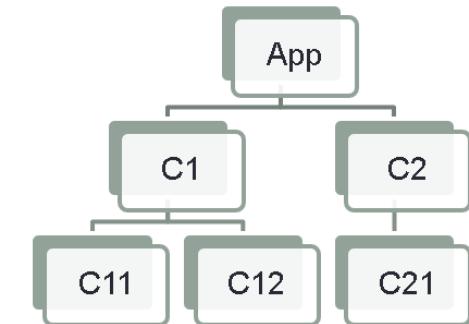
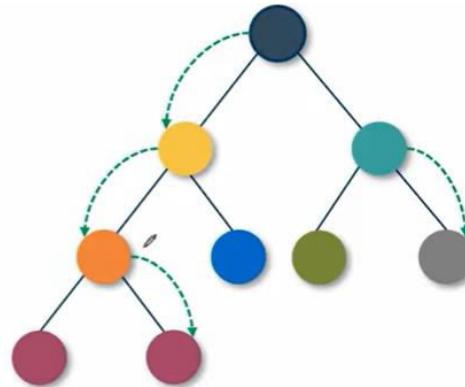
The screenshot shows a user interface for adding a new expense. It features three input fields: 'Title' with the value 'Mouse', 'Amount' with the value '900', and 'Date' with the value '03-12-2021'. Below the inputs is a purple button labeled 'Add Expense'.

In App component [App.js:20](#)
▶ {title: 'Mouse', amount: '900', date: '2021-12-03',
id: '0.25759054649698765'}

```
const NewExpense = (props) => {  
  
  const saveExpenseDataHandler = (inputExpenseData) => {  
    const expenseData = {  
      ...inputExpenseData,  
      id:Math.random().toString()  
    }  
    //console.log("In NewExpense ",expenseData)  
    props.onAddExpense(expenseData)  
  }  
  return(...);  
}  
export default NewExpense;
```

Lifting state up in React.js

- In a typical application, you pass a lot of state down to child components as props.
 - These props are often passed down multiple component levels.
 - That's how state is shared vertically in your application.
- Often there will be a need to **share state between different components**.
 - The common approach to share state between two components is to move the state to common parent of the two components.
 - This approach is called **as lifting state up** in React.js
 - React components can manage their own state
 - Often components need to communicate state to others
 - Siblings do not pass state to each other directly
 - State should pass through a parent, then trickle down



Simple example : Passing state vertically

```
import React, {useState} from 'react'
import ChildComponent from './ChildComponent';

const ParentComponent = () => {
  const [uname, setUserName] = useState('Shrilata')
  const [email, setEmail] = useState('shrilata@gmail.com')

  return(
    <ChildComponent uname={uname} email={email} />
  );
}
export default ParentComponent;
```

```
const ChildComponent = (props) => {
  return(
    <div>
      <div>Name : {props.uname}</div>
      <div>Email : {props.email}</div>
    </div>
  );
}
export default ChildComponent;
```

```
function App() {
  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <ParentComponent />
    ...
  )
}
```

Welcome to React!

Name : Shrilata
Email : shrilata@gmail.com

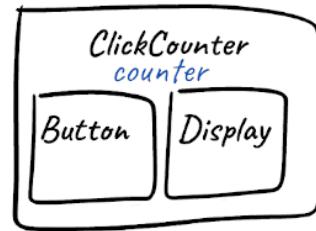
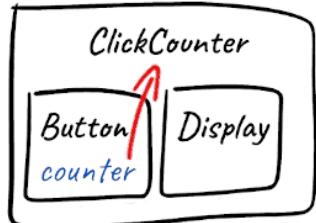
Lifting state up – simple example

```
import React, {useState} from 'react'
import Display from './Display'
import Button from './Button'

const ClickCounter = () => {
  const [counter, setCounter] = useState(0)

  const incrCounter = () => {
    setCounter(counter + 1)
  }
  return(
    <div>
      <Button onClick={incrCounter}>Button</Button>
      <Display message={`Clicked ${counter} times`}>
    </div>
  );
}
export default ClickCounter;
```

```
const Display = (props) => {
  return <p>{props.message}</p>;
}
export default Display;
```



The onClick here is a user defined prop, not an actual event

```
const Button = (props) => {
  return <button onClick={props.onClick}>Click me</button>;
}
export default Button;
```

This is an actual event



WORKING WITH LISTS AND CONDITIONALS

Working with Lists

- Lists are very useful when it comes to developing the UI of any website.
 - Lists are mainly used for displaying menus on a website, for example, the navbar menu.
 - In regular JavaScript, we can use [arrays](#) for creating lists. We can create lists in React in a similar manner as we do in regular JavaScript.

```
const ListComponent = () => {
  let arr = [1,2,3,4,5]

  let updatedArr = arr.map(num => {
    return <li>{num*num}</li>
  })

  return (
    <div>
      Numbers arr : {updatedArr}
    </div>
  )
}
export default ListComponent
```

Numbers arr :

- 1
- 4
- 9
- 16
- 25

```

const SimpleListComponent = () => {
  const nums = [1,2,3,4,5]
  const updatedNums = nums.map((num)=>{
    return <li>{num*num}</li>;
  });

  const items =[ {name:'Item-1'}, {name:'Item-2'} ]
  const updatedItems = items.map(item => (
    <p>{item.name}</p>
  ))

  const students = [{name:"Anita",rollno:101},
                    {name:"Sunita",rollno:102},
                    {name:"Kavita",rollno:103}]
  const updatedStudents = students.map(student => (
    <tr>
      <td>{student.name}</td><td>{student.rollno}</td>
    </tr>
  ))
  return(
    <div>
      <h2> Numbers Arr: </h2> {updatedNums}
      <h2> Items : </h2> {updatedItems}
      <h2> Students Data</h2>
      <table border="1">
        <tr><th>Name</th><th>Roll no</th></tr>
        {updatedStudents}</table>
    </div>
  );
  export default SimpleListComponent

```

Working with lists

Numbers arr :

- 1
- 4
- 9
- 16
- 25

Items :

Item-1

Item-2

Student Data

| Name | Roll no |
|--------|---------|
| Anita | 101 |
| Sunita | 102 |
| Kavita | 103 |

Working with lists in child component

```
import SimpleListChildComponent from './SimpleListChildComponent';

const SimpleListComponent = () => {
  const nums = [1,2,3,4,5]

  const items =[ {name:'Item-1'}, {name:'Item-2'}]

  const students = [{name:"Anita",rollno:101},
                  {name:"Sunita",rollno:102},
                  {name:"Kavita",rollno:103}]

  return(
    <div>
      <SimpleListChildComponent
        nums={nums}
        items={items}
        students={students} />
    </div>
  );
}

export default SimpleListComponent
```

```
const SimpleListChildComponent = (props) => {

  const nums = props.nums;
  const updatedNums = nums.map(...);
  const items = props.items
  const updatedItems = items.map(...)

  const students = props.students;
  const updatedStudents = students.map(...)

  return(...);
}

export default SimpleListChildComponent;
```

Working with the expenses list

```
function App() {
  const expenses = [
    { title: 'Groceries', amount: 900, date: new Date(2020, 7, 14)},
    { title: 'New TV', amount: 34000, date: new Date(2021, 2, 12) },
    { title: 'Sofa Set', amount: 25000, date: new Date(2021, 2, 28)}
  ];
  return (
    <div className="App">
      <h2>Welcome to React!</h2>

      {expenses.map(expense => {
        return <ExpenseItem
          expDate={expense.date}
          expTitle={expense.title}
          expAmount={expense.amount}
        />
      })}
      {/* <ExpenseItem
        expDate={expenses[0].date}
        expTitle={expenses[0].title}
        expAmount={expenses[0].amount}
      /> ... */}
    </div>
  );
}

export default App;
```

```
{expenses.map(expense =>
  (<ExpenseItem
    expDate={expense.date}
    expTitle={expense.title}
    expAmount={expense.amount}
  />))
}
```



Using stateful lists

```
const DUMMY_EXP = [
  { title: 'Groceries', amount: 900, date: new Date(2020, 7, 14) },
  { title: 'New TV', amount: 34000, date: new Date(2021, 2, 12) },
  { title: 'New Sofa Set', amount: 25000, date: new Date(2021, 2, 28) }
];
function App() {
  const [expenses, setExpenses] = useState(DUMMY_EXP)

  const addExpenseHandler = expense => {
    //console.log("In App component ", expense)
    setExpenses(prevArr => {return [expense, ...prevArr]})
  }

  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <NewExpense onAddExpense={addExpenseHandler} />

      {expenses.map(expense => (<ExpenseItem
        expDate={expense.date}
        expTitle={expense.title}
        expAmount={expense.amount}
      />))}
    </div>
  );
}

export default App;
```

| Title | Amount |
|------------------------------|-------------------------------|
| Keyboard | 1500 |
| Date | |
| 01-12-2021 | |
| <button>Add Expense</button> | |
| August 14 2020 | Groceries |
| Rs 900 | <button>Change Title</button> |
| March 12 2021 | New TV |
| Rs 34000 | <button>Change Title</button> |
| March 28 2021 | New Sofa Set |
| Rs 25000 | <button>Change Title</button> |

| | | | |
|------------------------|--------------|----------|-------------------------------|
| December 04 2021 | Keyboard | Rs 1500 | <button>Change Title</button> |
| August 14 2020 | Groceries | Rs 900 | <button>Change Title</button> |
| March 12 2021 | New TV | Rs 34000 | <button>Change Title</button> |
| March 28 2021 | New Sofa Set | Rs 25000 | <button>Change Title</button> |

Lists and keys

- When creating a list in JSX, React may show you an error and ask for a key.
 - Keys are unique identifiers that must be attached to the top-level element inside a map.
 - React uses keys to know how to update a list- whether adding, updating, or deleting items.
 - This is part of how React is so fast with large lists.
 - Keys are a way to help React know how to efficiently update a list.
 - We can add a key using the key prop like so:

```
✖ ▶ Warning: Each child in a list should have a unique "key" prop.      index.js:1 ⓘ  
Check the render method of `App`. See https://reactjs.org/link/warning-keys for more  
information.  
    at ExpenseItem (http://localhost:3000/static/js/main.chunk.js:685:19)  
    at App (http://localhost:3000/static/js/main.chunk.js:184:89)
```

```
<ul>  
  {[1,2,3].map(no => <li key={no}>{no}</li>)}  
</ul>
```

```
<div>  
  {people.map(person => (  
    <p key={person.name}>{person.name}</p>  
  ))}  
</div>
```

```
<li key={`$somevar`} ...>
```

Lists and keys

```
✖ ▶ Warning: Each child in a list should have a unique "key" prop.      index.js:1 ⓘ
Check the render method of `App`. See https://reactjs.org/link/warning-keys for more
information.
    at ExpenseItem (http://localhost:3000/static/js/main.chunk.js:685:19)
    at App (http://localhost:3000/static/js/main.chunk.js:184:89)
```

```
var expenses = [
  {id:1, title : "Myntra shopping", date:new Date(2024, 1, 15, 12, 0, 0)},
  {id:2, title : "Amazon shopping", date:new Date(2024, 1, 15, 12, 0, 0)},
  {id:3, title : "Renovation", date:new Date(2024, 1, 15, 12, 0, 0)}
]

return (
  <div className="App">
    <h1>Learn React</h1>
    <NewExpense onSave={inputHandler}>{/}
    {
      expenses.map(expense => (
        <ExpenseItem
          key={expense.id}
          expDate={expense.date}
          expTitle={expense.title}
          expAmount={expense.amount} />
      ))
    }
  </div>
)
```

- Each list item in a list should have a unique key.
- A “key” is a special string attribute you need to include when creating lists of elements in React.
- Keys allow React to keep track of elements. This way, if an item is updated or removed, only that item will be re-rendered instead of the entire list.
- key property to enable React to keep track of elements, so that it has a clear property that it can compare between the elements - to find out which elements changed and which didn't - so that it only renders the elements that changed and not the whole list.

```

let ListComponent = () => {
  var personarr = [
    { pid: 101, pname: "Savita" },
    { pid: 102, pname: "Anita" },
    { pid: 103, pname: "Kavita" }
  ]
  let [persons, setPersons] = useState(personarr)

  let deleteHandler = pid => {
    setPersons(oldarr => oldarr.filter(person => person.pid !== pid))
  }
  return (
    <div>
      <table border="1">
        {persons.map(p => (
          <tr key={p.pid}>
            <td>{p.pid}</td><td>{p.pname}</td>
            <td><button onClick={() =>
              deleteHandler(p.pid)}>delete</button></td>
          </tr>
        )))
      </table>
    </div>
  )
}

export default ListComponent

```

Deleting ExpenseItem from list

| | | |
|-----|--------|-------------------------|
| 101 | Savita | <button>delete</button> |
| 102 | Anita | <button>delete</button> |
| 103 | Kavita | <button>delete</button> |

```

function App() {
...
let deleteHandler = (id) => {
  console.log("item to be deleted", id)
  setExpenses(oldValues => {
    return oldValues.filter(item => item.expTitle !== id)
  })
}
return (
...
{ expenses.map(expense => (
  <ExpenseItem key={expense.id}>
    ...
    onDelete={deleteHandler} />
))})

```

Deleting ExpenseItem from list

```

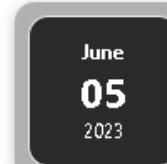
//expenseItem.css
.delete-button {
  display: flex;
  flex-direction: column;
  width: 5.5rem;
  height: 2.5rem;
  border: 1px solid #ececfc;
  background-color: red;
  color: white;
  border-radius: 12px;
  align-items: center;
  justify-content: center;
}

```

```

let ExpenseItem = (props)=>{
  return (
    <div className="expense-item">
      ...
      <button className="delete-button"
        onClick={()=>props.onDelete(props.expTitle)}>Delete Item</button>
    </div>)
}

```



Amazon Shopping

3000

Delete Item

CONDITIONAL RENDERING

Rendering content conditionally

- Conditional rendering means to render a specific HTML element or React component depending on a prop or state value.
 - In a conditional render, a React component decides based on one or several conditions which DOM elements it will return.
 - For instance, based on some logic it can either return a list of items or a text that says "Sorry, the list is empty".

```
if(condition_is_met) {  
    renderSectionOfUI();  
}
```

- Consider these two components:

```
function UserGreeting(props) {  
    return <h1>Welcome back!</h1>;  
}
```

```
function GuestGreeting(props) {  
    return <h1>Please sign up.</h1>;  
}
```

- Lets create a Greeting component that displays either of these components depending on whether a user is logged in:

```
function Greeting(props) {  
    const isLoggedIn = props.isLoggedIn;  
    if (isLoggedIn) {  
        return <UserGreeting />;  
    }  
    return <GuestGreeting />;  
}
```

```

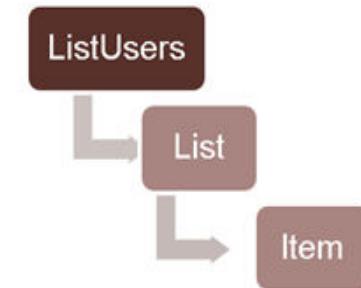
/*const users = [
  { id: '1', firstName: 'Shrilata', lastName: 'T' },
  { id: '2', firstName: 'Anita', lastName: 'Patil' },
];
*/
const users = []
var users;

function ListUsers() {
  return (
    <div>
      <List list={users} />
    </div>
  )
}

function List({ list }) {
  if (!list) {
    return null;
  }
  if (!list.length) {
    return <p>Sorry, the list is empty.</p>;
  }
  return (
    <ul>
      {list.map(item => (
        <Item key={item.id} item={item} />
      ))}
    </ul>
  )
}

```

Rendering content conditionally : example-1



```

function Item({ item }) {
  return (
    <li>
      {item.firstName} {item.lastName}
    </li>
  );
}
export default ListUsers;

```

Hello Com

- Shrilata T
- Anita Patil

Rendering content conditionally : example-2

```
import React, { useState } from 'react'
function ConditionalRenderingDemo() {

  const [uname, setuname] = useState('')
  const [submitted, setSubmitted] = useState(false)

  let handleChange = (event) => {
    setuname(event.target.value)
  }

  let handleClick = () => {
    setSubmitted(true)
  }

  return (
    <div>
      <h2>Welcome {submitted? uname:'unknown entity'}</h2>
      <input onChange={handleChange} value={uname} /><br />
      <input type="button" value="submit" onClick={handleClick} />
    </div>
  )
}
export default ConditionalRenderingDemo
```

Welcome unknown entity

Welcome Shrilata

Using ternary operator

arrow pointing to the line: Welcome {submitted? uname:'unknown entity'}

Rendering content conditionally : example-3

- Using conditions with logical && operator

```
import ListBooks from "./ListBooks";

const BookInfo = () => {
  const books = [
    {id:101, name:"Core Java", author:"author1", price:400},
    {id:102, name:"Core Servlets", author:"author2", price:450},
    {id:103, name:"Core JSP", author:"author2", price:300},
    {id:104, name:"Spring Basics", author:"author3", price:500},
  ]
  return(
    <div>
      <ListBooks books={books} />
    </div>
  );
}
export default BookInfo;
```

```

import { useState } from "react";
function ListBooks(props) {

  const [show, setShow] = useState(false)
  const [books, setBooks] = useState(props.books)

  const btnHandler = () => {
    setShow(!show);
  }
  return (
    <div>
      <h1>Book List</h1>
      <table border="1">
        {show && books.map(book =>
          (<tr>
            <td>{book.id}</td><td>{book.name}</td>
            <td>{book.author}</td><td>{book.price}</td>
          </tr>))
        )
      </table>
      <button onClick={btnHandler}>Toggle book list</button>
    </div>
  );
}
export default ListBooks;

```

Using Inline Boolean Condition

| Book List | | | |
|-------------------------------------------------|--|--|--|
| <input type="button" value="Toggle book list"/> | | | |
| | | | |

| Book List | | | |
|-------------------------------------------------|---------------|---------|-------|
| Id | Name | Author | Price |
| 101 | Core Java | author1 | 400 |
| 102 | Core Servlets | author2 | 450 |
| 103 | Core JSP | author2 | 300 |
| 104 | Spring Basics | author3 | 500 |
| <input type="button" value="Toggle book list"/> | | | |

We can use the logical `&&` operator along with some condition to decide what will appear in output based on whether the condition evaluates to true or false.

If the *condition* provided in the above syntax evaluates to True then the elements right after the `&&` operator will be a part of the output and if the condition evaluates to false then the code within the curly braces will not appear in the output.

Rendering content conditionally : example-3 alternative

```
function ListBooks(props) {  
  
  const [show, setShow] = useState(false)  
  const [books, setBooks] = useState(props.books)  
  
  const btnHandler = () => {  
    setShow(!show);  
  }  
  return (  
    <div>  
      <h1>Book List</h1>  
      <table border="1">  
        { show ? books.map(book =>  
          (<tr>  
            <td>{book.id}</td><td>{book.name}</td>  
            <td>{book.author}</td><td>{book.price}</td>  
          </tr>) ) : null }  
      </table>  
      <button onClick={btnHandler}>Toggle book list</button>  
    </div>  
  );  
}  
export default ListBooks;
```

Using ternary operator

Rendering content conditionally : example-3 alternative

```
import { useState } from "react/cjs/react.development";

function ListBooks(props) {

  const [show, setShow] = useState(false)
  const [books, setBooks] = useState(props.books)

  const btnHandler = () => {
    setShow(!show);
  }
  const showBooks = () => {
    const bookdata = books.map(book => (<tr>
      <td>{book.id}</td><td>{book.name}</td>
      <td>{book.author}</td><td>{book.price}</td>
    </tr>))
    if (show)
      return (bookdata);
    else
      return null
  }

  return (
    <div>
      <h1>Book List</h1>
      <button onClick={btnHandler}>Toggle book list
      </button>
      {showBooks()}
    </div>
  );
}

export default ListBooks;
```

Using a function

React Fragments

```
function App() {  
  return (  
    <div>  
      <h1> Hi, welcome to React</h1>  
      <p>This is a simple para </p>  
    </div>  
  );  
}
```

```
import { Fragment } from 'react'  
function App() {  
  return (  
    <Fragment>  
      <h1>Lets get started</h1>  
      <p>This is a simple para </p>  
    </Fragment>  
  )  
}
```

```
<body>  
  <div id="root">  
    <div> == $0  
      <h1>Lets get started</h1>  
      <p>This is a simple para </p>
```

```
<body>  
  <div id="root">  
    <h1>Lets get started</h1> == $0  
    <p>This is a simple para </p>
```

```
function App() {  
  return (  
    <>  
      <h1>Lets get started</h1>  
      <p>This is a simple para </p>  
    </>  
  )  
}
```

CLASS-BASED COMPONENTS

Class-based components : Examples

```
const HelloComponent = () => {
  return <h3>Hello, welcome user!!</h3>
}
export default HelloComponent;
```

```
class HelloComponent extends Component{
  render(){
    return (<h3>Hello, welcome user!!</h3>)
  }
}
export default HelloComponent;
```

Functional components are regular javascript functions which return renderable results (typically JSX)

Class based components are defined using Javascript classes where a render method defines the to-be-rendered output

Class-based components : passing into props

```
import React, {Component} from 'react';
import User from './User'

const users= [
  { id: 'u1', name: 'Shrilata' },
  { id: 'u2', name: 'Soha' },
  { id: 'u3', name: 'Sia' },
]
```

```
class Users extends Component{
  render(){
    return(
      <div>
        <User name={users[0].name} />
        <User name={users[1].name} />
        <User name={users[2].name} />
      </div>
    );
  }
}
export default Users;
```

```
function App() {
  return (
    <div className="App">
      <h1> Welcome to React</h1>
      <Users />
    </div>
  );
}
```

- Hello Shrilata
- Hello Soha
- Hello Sia

```
import './User.css';
import React, {Component} from 'react';

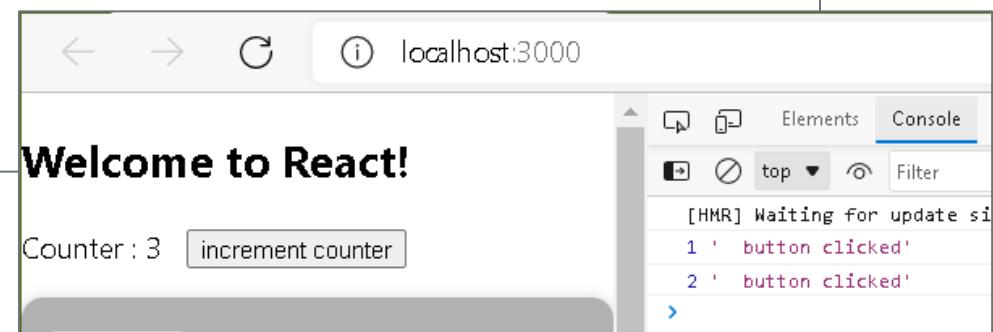
class User extends Component{
  render(){
    return (<li>Hello {this.props.name}</li>);
  }
}
export default User;
```

React State : Class-based component

```
import React, {Component} from "react";
class StatefulComponent extends Component{
  state = {counter:1 }

  btnHandler = () => {
    this.setState({counter:this.state.counter+1})
    console.log(this.state.counter, " button clicked")
  }
  render(){
    return(
      <div>
        Counter : {this.state.counter} &nbsp;
        <button onClick={this.btnHandler}>increment counter</button>
      </div>
    );
  }
  export default StatefulComponent;
```

```
constructor(){
  super();
  this.state = {counter:1}
}
```



React State : Example

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

class statefulComponent extends Component{
  state = {
    name: "Shrilata",
    email: "shrilata@gmail.com",
    address:"Pune"
  }
  render(){
    return(
      <div>
        <h3>Name : {this.state.name}</h3>
        <h3>Email : {this.state.email}</h3>
        <h3>Address : {this.state.address}</h3>
      </div>
    );
  }
  export default statefulComponent;
```

Unlike useState, its so easy to group state together in class component

Name : Shrilata

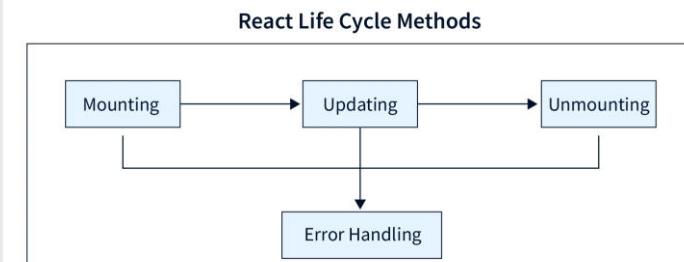
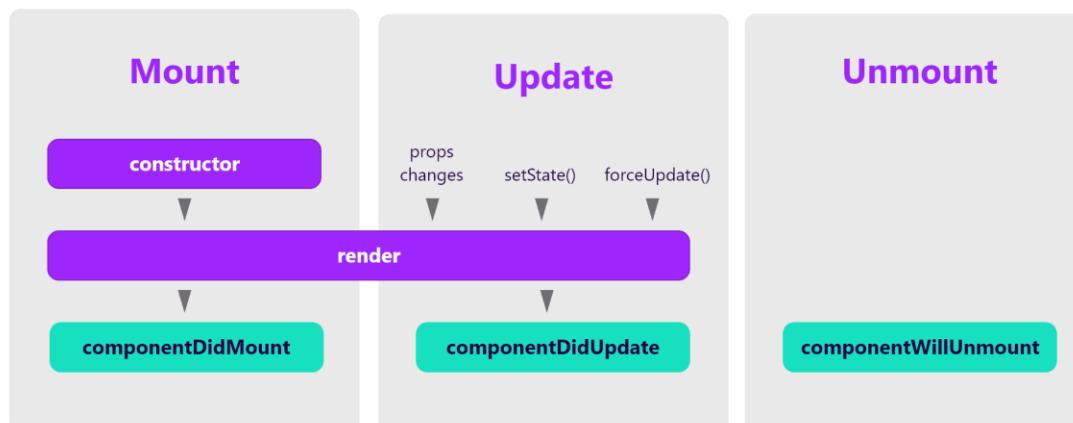
Email : shrilata@gmail.com

Address : Pune

LIFECYCLE EVENTS

Lifecycle methods

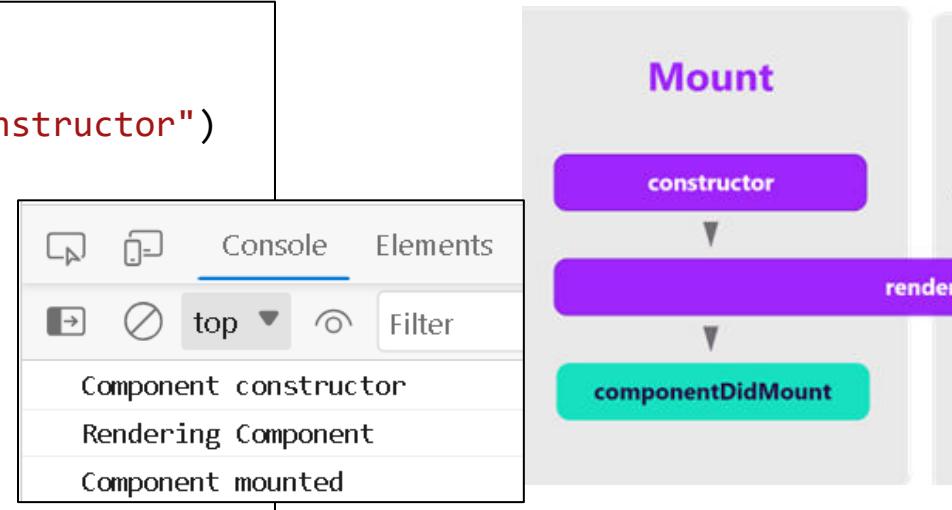
- Lifecycle methods are pre-defined functions in React defined for components
- A [React](#) component undergoes 3 phases in its lifecycle - mounting, updating, and unmounting.
 - Each React lifecycle phase has a number of lifecycle methods that you can override to run code at specified times during the process.
 - These are popularly known as component lifecycle methods.
 - Each phase has specific methods responsible for a stage in a component's lifecycle.
 - These methods are technically particular to class-based components and not intended for functional components.
 - However, since the concept of Hooks was released in React, you can now use abstracted versions of these lifecycle methods when you're working with functional component state.



Lifecycle methods : mounting phase

- The mounting phase refers to the phase during which a component is created and inserted to the DOM.
- The following methods are called in order.
 - `constructor()` : This lifecycle method is the first one that is called and is usually where you will set the state of your component.
 - `render()`: You have this method for all the components created; it is the only required method within a class component in React. This method is in charge of displaying HTML markup to the user.
 - `componentDidMount()`: This method is called right after the react component is mounted on DOM; ie it is called right after render method executed for the first time
 - Here we can make API call, for eg, interact with dom or perform any ajax call to load data.

```
class HelloComponent extends Component{
  constructor(){
    super(); console.log("Component constructor")
  }
  componentDidMount(){
    console.log("Component mounted")
  }
  render(){
    console.log("Rendering Component")
    return <h3>Hello</h3>
  }
}
```

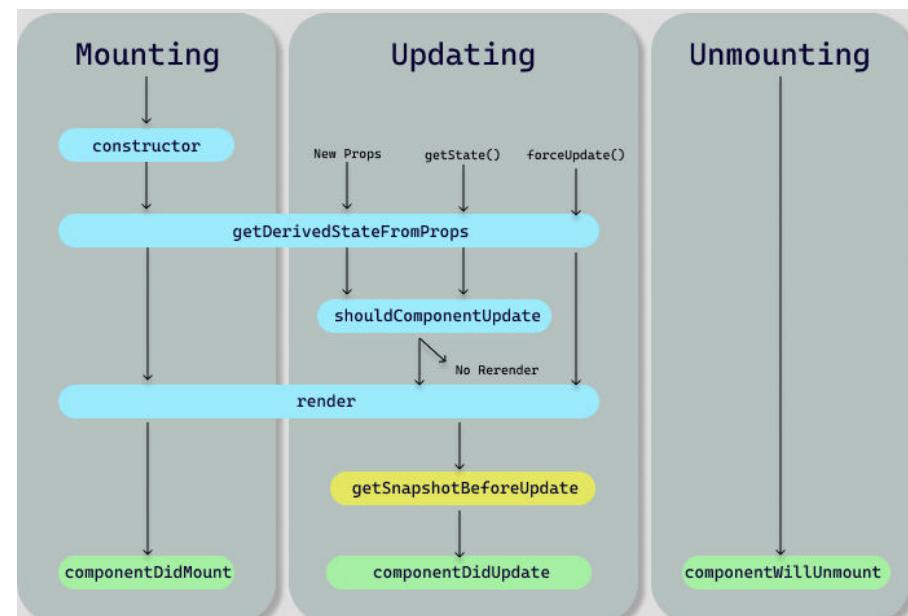


Lifecycle methods : updating phase

- An update is caused by props or state changes.
 - The component is re-rendered whenever a change is made to react component's state or props ; ie the component is updated
 - For example, you enter something in the textbox, so the state properties are updated.
- Some methods and order in which they are called:
 - `shouldComponentUpdate()`: executed before rendering data if any changes are updated in state & props. Returns true by default, if false, component will not re-render on updating the existing value of state & props
 - `render()` : already seen
 - `componentDidUpdate()` : executes after the real DOM is updated

`getDerivedStateFromProps()` :available in React 16.3 and later versions. Is invoked during the mounting and updating phase of a component.

During the mounting phase, it is called after the constructor and before render(). This method is called for every render cycle and provides an opportunity to update the component's state based on changes in props before the initial render.



Lifecycle methods : unmounting phase

- UnMounting: this state comes into the picture when the Component is not required or removed.
- Following are the methods available in unmount state:
 - `ComponentWillUnmount()`: is called just before the component is unmounted or destroyed.
 - Unmounting is simply removing the component from the DOM so that we cannot modify its state or rerender it.

```
class App extends React.Component {  
  componentDidMount() {  
    //the event listener is added  
    element.addEventListener()  
  }  
  
  componentWillUnmount() {  
    //the event listener is removed  
    element.removeEventListener()  
  }  
}
```

example : we put an event listener inside the `componentDidMount()` method and then remove that event listener inside the `componentWillUnmount()` method.

ERROR BOUNDARIES

Using Error Boundaries

```
class HelloComponent extends Component{  
  render(){  
    let rnd = Math.random();  
    if(rnd > 0.7){  
      throw new Error("Something went wrong");  
    }  
    return <h3>Hello {rnd}</h3>  
  }  
}
```

```
class Users extends Component{  
  componentDidUpdate(){  
    if(this.props.users.length == 0)  
      throw new Error("No users provided")  
  }  
  ...  
}
```

Hello 0.003978319817133036

Error: No users provided!

Users.componentDidUpdate
src/components/Users.js:17

```
14 |  
15 |   componentDidUpdate() {  
16 |     if (this.props.users.length === 0) {  
> 17 |       throw new Error('No users provided!');  
18 |     }  
19 |   }  
20 |
```

View compiled

The screenshot shows a browser window with an error message: "Uncaught runtime errors:" and "Something went wrong". Below the browser is the React DevTools console. A red box highlights the error message in the console:

```
Uncaught Error: Something went wrong  
at HelloComponent.render (HelloComponent.js:7:1)
```

- By default, if your application throws an error during rendering, React will remove its UI from the screen.
- A JavaScript error in a part of the UI shouldn't break the whole app.
- To solve this problem for React users, React 16 introduces a new concept of an “error boundary”.

Error Boundaries

Error boundaries are like a JavaScript catch {} block, but for components.

- Error boundaries were introduced in React v16
 - To use them, you need to define a class component with either or both of the following lifecycle methods: `getDerivedStateFromError()` or `componentDidCatch()`.
 - `componentDidCatch()`: This method is used for logging error information.

```
import React, {Component} from 'react';
class ErrorBoundary extends Component{
  state = {
    hasError:false,
    errorMessage:''
  }
  componentDidCatch = (error, info) => {
    this.setState({hasError:true, errorMessage:error});
  }
  render(){
    if(this.state.hasError)
      return <h1>{this.state.errorMessage}</h1>
    else
      return this.props.children
  }
}
export default ErrorBoundary;
```

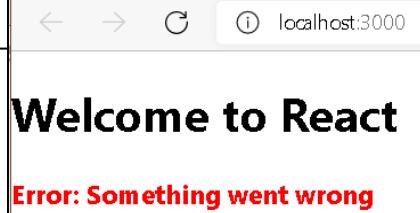
Use it like this:
<ErrorBoundary>
 <Artists />
</ErrorBoundary>

```
import React from 'react'
function Artists({artistName}) {
  if (artistName === 'peruzzi') {
    throw new Error ('not performing tonight!')
  }
  return (
    <div>
      {artistName}
    </div>
  )
}
export default Artists
```

Using Error Boundaries

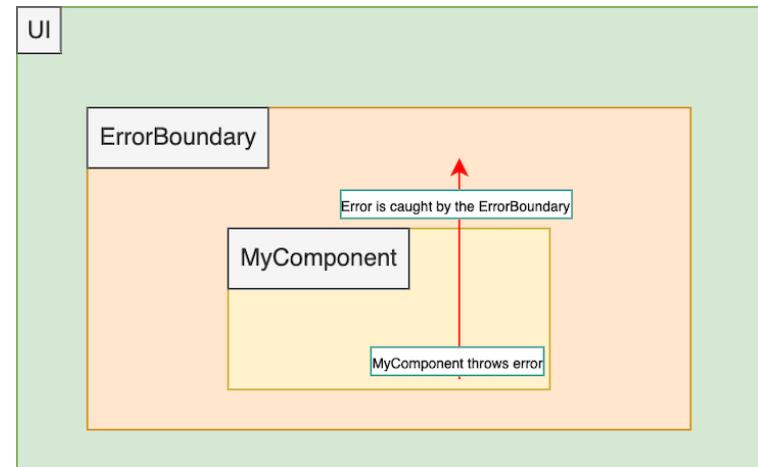
- Where to place these error boundaries?
 - They can be set up around the entire app or individual components for more granular control.

```
function App() {  
  return (  
    <div className="App">  
      <h1> Welcome to React</h1>  
      <ErrorBoundary>  
        <Person />  
      </ErrorBoundary>  
    </div>  
  );  
}
```



Welcome to React

From Error throwing Person component - all is well!



A screenshot of a browser developer tools console. The title bar says 'localhost:3000'. The console output shows a stack trace starting with 'Array.WrappedJsonpCallback [as push]'. The code snippets are as follows:

```
> 45 |     result = __webpack_require__(__webpack_require__.s = deferredModule[0]);  
46 | }  
47 | }  
48 |  
View compiled  
  
Array.WrappedJsonpCallback [as push]  
E:/FreelanceTrg/ReactJS/Demo-Nov2021/second-app/webpack/bootstrap:32  
  
> 29 |     deferredModules.push.apply(deferredModules, executeModules || []);  
30 |  
31 | // run deferred modules when all chunks ready  
> 32 |     return checkDeferredModules();  
33 | };  
34 | function checkDeferredModules() {  
35 |     var result;  
  
View compiled  
  
(anonymous function)  
http://localhost:3000/static/js/main.chunk.js:1:73
```

At the bottom of the console, there is a message in an orange box: 'This screen is visible only in development. It will not appear if the app crashes in production. Open your browser's developer console to further inspect this error. Click the X or hit ESC to dismiss this message.'

USE EFFECTS IN FUNCTIONS

Understanding useEffect() hook

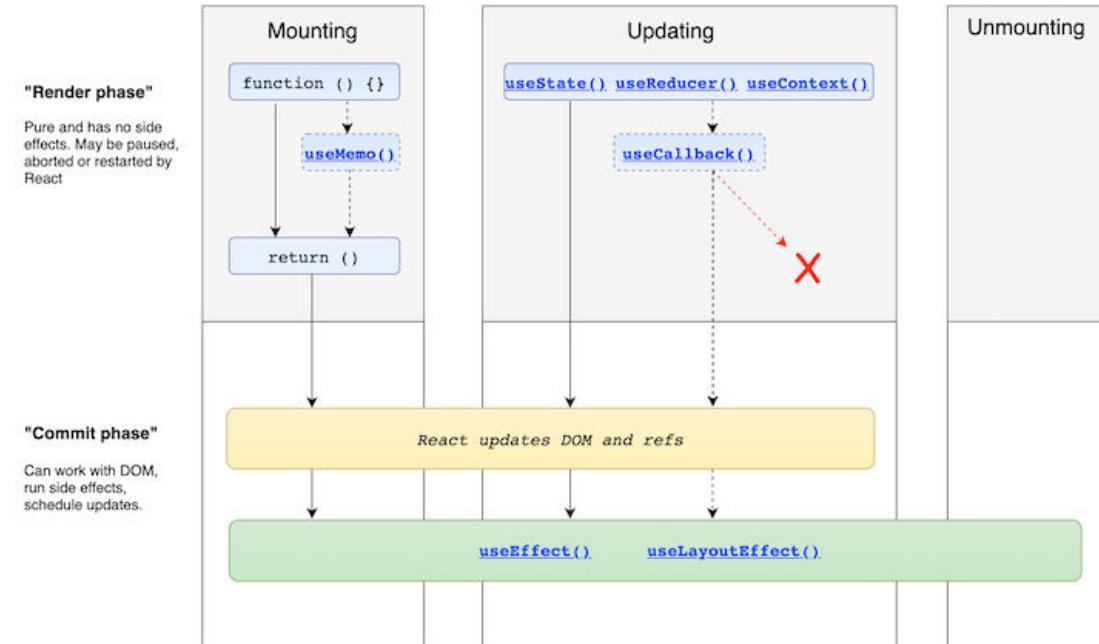
- In React.js, managing side effects, such as fetching data, subscriptions, and manipulating the DOM, is a crucial aspect of building modern and dynamic web applications.
- The `useEffect()` hook plays a central role in handling **side effects and lifecycle events** in functional components.
- How is working with `useEffect()` different from working with the lifecycle methods of class-based components?
 - lifecycle methods give components a predictable structure.
 - The code is more explicit (in contrast to effects), so developers can directly spot the relevant parts (e.g., `componentDidMount`) in terms of performing tasks at particular lifecycle phases (e.g., on component unmount).
- `useEffect()` allows functional components to perform side effects, such as data fetching, subscriptions, or DOM manipulations, **after rendering**.
 - With `useEffect`, developers can perform side effects without writing class components and managing lifecycle methods like `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.

`useEffect()` is a React hook that's lets you run side effects



React Hooks Lifecycle

- Effects defined with `useEffect` are invoked after render.
- To be more specific, it runs both after the first render and after every update.
- In contrast to lifecycle methods, effects don't block the UI because they run asynchronously.
- Unlike class components, you do not do something after the component is mounted; you do something after the component is first presented to the user.



- Hooks force you to think more from the user's perspective.
- Syntax:
 - `import { useEffect } from 'react';`
 - `useEffect(()=>{} , [dependencies])`
- The `useEffect` hook takes two arguments: a callback function and an optional dependency array.
- The callback function represents the side effect logic, while the dependency array controls when the side effect should run or update.

```
useEffect(() => {
  //Runs on every render
});
```

Understanding useEffect() hook

- The following steps are carried out when executing an effect:
 - **Initial render/mounting:** When a functional component that contains a useEffect Hook is initially rendered, the code inside the useEffect block runs after the initial render of the component.
 - This is similar to componentDidMount class components
- **Subsequent renders:** You can pass a dependency array as the second argument to the useEffect Hook.
- This array contains variables or values that the effect depends on. Any change in these variables will re-render the component.
- *If no dependency array is given, the effect will run on every render*
- **Cleanup function:** You can also run an optional cleanup function inside the effect.
- It is used to clean up any resources or subscriptions created by the effect when the component is unmounted or when the dependencies change
- **Unmounting:** If a cleanup function is present, it is run and then the component is unmounted from the DOM

Understanding useEffect() hook : Example-1

```
import { useState, useEffect } from "react";

function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    setTimeout(() => {
      setCount((count) => count + 1);
    }, 1000);
  });

  return <h1>I have rendered {count} times!</h1>;
}
export default Timer
```

I have rendered 15 times!

//No dependency passed:

```
useEffect(() => {
  //Runs on every render
});
```

- Using setTimeout() to count 1 second after initial render
- But it keeps counting even though it should only count once!
- Why? useEffect runs on every render. That means that when the count changes, a render happens, which then triggers another effect.

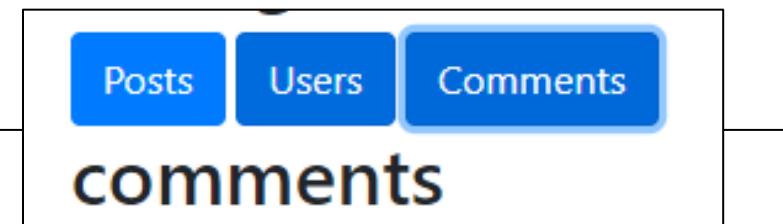
Understanding useEffect() hook

- How to control when side effects run?
 - Always include the second parameter which accepts an array. We can pass dependencies to useEffect in this array

```
//An empty array:  
  
useEffect(() => {  
  //Runs only on the first render  
}, []);
```

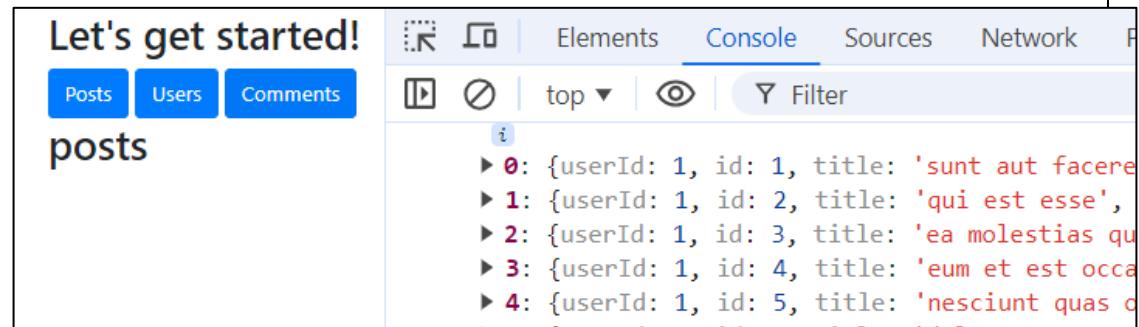
Understanding useEffect() hook : Example-3

```
const EffectDemo = () => {  
  
  const [choice, setChoice] = useState('posts')  
  
  return (  
    <div>  
      <div>  
        <button className="btn btn-primary"  
          onClick={() => setChoice('posts')}>Posts</button>  
        <button className="btn btn-primary"  
          onClick={() => setChoice('users')}>Users</button>  
        <button className="btn btn-primary"  
          onClick={() => setChoice('comments')}>Comments</button>  
      </div>  
      <h2>{choice}</h2>  
    </div>  
  )  
}  
export default EffectDemo
```



Understanding useEffect() hook : Example-3 (contd)

```
const EffectDemo = () => {  
  
  const [choice, setChoice] = useState('posts')  
  
  useEffect(()=>{  
    fetch(`https://jsonplaceholder.typicode.com/${choice}`)  
    .then(response => response.json())  
    .then(json => console.log(json))  
  }, [choice])  
  
  return (  
    <div>  
      <div>  
        <button ...>  
      </div>  
      <h2>{choice}</h2>  
    </div>  
  )  
}  
export default EffectDemo
```



Understanding useEffect() hook : Example-3 (contd)

```
const EffectDemo = () => {

  const [choice, setChoice] = useState('posts')
  const [items, setItems] = useState([])

  useEffect(()=>{
    fetch(`https://jsonplaceholder.typicode.com/${choice}`)
      .then(response => response.json())
      .then(json => setItems(json))
  }, [choice])

  return (
    <div>
      <div>
        <button ...> ...
      </div>
      <h2>{choice}</h2>
      {
        items.map(item => {
          return <pre>{JSON.stringify(item)}</pre>
        })
      }
    </div>
  )
}
```



Understanding useEffect() hook : Example-3 (contd)

- Same code as previous, but with async and await

```
import { useEffect, useState } from "react"

const EffectDemo = () => {

  const [choice, setChoice] = useState('posts')
  const [items, setItems] = useState([])

  useEffect(() => {
    async function fetchData() {
      const response =
        await fetch(`https://jsonplaceholder.typicode.com/${choice}`)
      const data = await response.json()
      setItems(data)
    }

    fetchData()
  }, [choice])

  ...
}
```

Controlled Components and Uncontrolled components

- Uncontrolled component and Controlled component are terms used to describe React components that render HTML form elements.
 - Every time you create a React component that renders an HTML form element, you are creating one of those two.
 - **Controlled component** are those whose form input fields are handled via state by React
 - **Uncontrolled component** are those that are handled directly via DOM.
- A controlled component is bound to a value, and its adjustments will be handled in code by using event-based callbacks.
 - Here, the input form variable is handled by the react itself rather than the DOM.
 - In this case, the mutable state is maintained in the state property and modified using `setState()`.
- **Controlled components have functions which regulate the data** that occurs at each on Change event.
 - **This data is subsequently saved in the `setState()` method and updated.** It helps components manage the elements and data of the form easier.

Uncontrolled Inputs

- “uncontrolled” form inputs: React doesn’t track the input’s state.
 - The DOM handles the data
 - Uncontrolled components are inputs that do not have a value property.
 - In opposite to controlled components, it is the application’s responsibility to keep the component state and the input value in sync.
 - To create a uncontrolled component, we can use JavaScript. But React allows us to create a “**ref**” (reference) to associate with an element, giving access to the underlying DOM node
- “**ref**” is used to receive the form value from DOM.
 - To enable this, React allows us to create a “ref” (reference) to associate with an element, giving access to the underlying DOM node.
 - Refs provide a way to access DOM nodes or React elements created in the render method.
 - In class component, Refs are created using `React.createRef()` and attached to React elements via the **ref** attribute.
 - Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.
 - In function components, refs are created using `useRef()` hook

Uncontrolled Inputs

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
    return <div ref={this.myRef} />;
  }
}
```

```
let SimpleUncontrolledForm = () => {

  let unameRef = useRef("")

  return (
    <form onSubmit={submitHandler}>
      Username : <input ref={unameRef} />
      ...
  )
}
```

Uncontrolled Inputs with function component

```
import React, { useRef, useState } from 'react'

function RefDemo() {
  const uname = useRef()
  const [username, setUsername] = useState('')

  let handleClick = () => {
    setUsername(uname.current.value)
  }

  return (
    <div>
      <h2>Welcome {username? username:'unknown entity'}</h2>
      <input ref={uname}/><br />
      <input type="button" value="submit" onClick={handleClick} />
    </div>
  )
}

export default RefDemo
```

Welcome unknown entity

Welcome Shrilata

Uncontrolled Inputs : class component

```
import React, {Component} from 'react';
class SimpleForm extends Component {
  constructor(props) {
    super(props);
    // create a ref to store the DOM element
    this.nameEl = React.createRef(),
  }

  handleSubmit = (e) => {
    e.preventDefault();
    alert(this.nameEl.current.value);
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>Name:
          <input type="text" ref={this.nameEl} />
        </label>
        <input type="submit" name="Submit" />
      </form>
    )
  }
}

export default SimpleForm;
```

- You initialize a new ref in the constructor by calling `React.createRef`, assigning it to an instance property so it's available for the lifetime of the component.
- In order to associate the ref with an input, it's passed to the element as the special `ref` attribute.
- Once this is done, the input's underlying DOM node can be accessed via `this.nameEl.current`.



```

class LoginForm extends Component {
  constructor(props) {
    super(props);
    this.nameEl = React.createRef();
    this.passwordEl = React.createRef();
    this.rememberMeEl = React.createRef();
  }
  handleSubmit = (e) => {
    e.preventDefault();
    const data = {
      username: this.nameEl.current.value,
      password: this.passwordEl.current.value,
      rememberMe: this.rememberMeEl.current.checked,
    }
    console.log(data)
  }
  render(){
    return (
      <form onSubmit={this.handleSubmit}>
        <fieldset><legend>Login Form</legend>
        <input type="text" placeholder="username" ref={this.nameEl} /><br><br>
        <input type="password" placeholder="password" ref={this.passwordEl} /><br><br>
        <label><input type="checkbox" ref={this.rememberMeEl} />Remember me
        </label><br><br>
        <button type="submit" className="myButton">Login</button>
      </fieldset>
    </form>
  );
}

```

Another example : Login form

▶ {
 username: 'aaa', password: 'bbb', rememberMe: true}
 }

COMPOSITION VS. INHERITANCE

Composition over Inheritance

- Composition and inheritance are the approaches to use multiple components together in React.js .
- This helps in code reuse.
- React recommends using composition instead of inheritance as much as possible and inheritance should be used in very specific cases only.
- Composition works with functions as well as classes both.

- Composition is a code reuse technique where a larger object is created by combining multiple smaller objects

```

class UserNameForm extends Component {
  render() {
    return (
      <div><input type="text" /></div>
    )
  }
}

export class CreateUserName extends UserNameForm {
  render() {
    const parent = super.render();
    return (
      <div>{parent}
        <button>Create</button>
      </div>
    )
  }
}

export class UpdateUserName extends UserNameForm {
  render() {
    const parent = super.render();
    return (
      <div>{parent}
        <button>Update</button>
      </div>
    )
  }
}

```

Inheritance

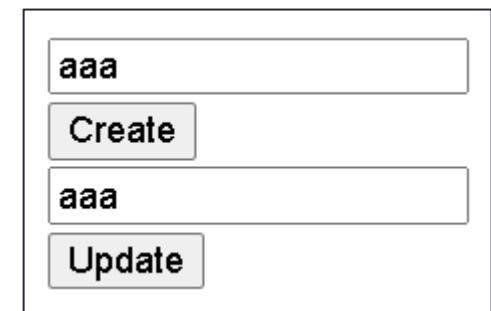
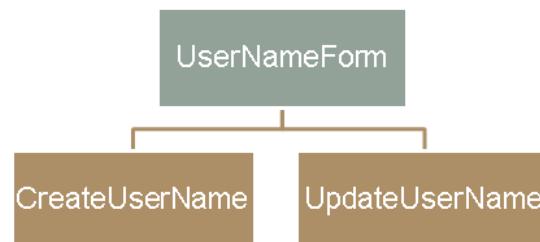
```

import { CreateUserName,
  UpdateUserName } from ...

function App() {
  return (
    <div className="App">
      <CreateUserName />
      <UpdateUserName />
    </div>
  )
}

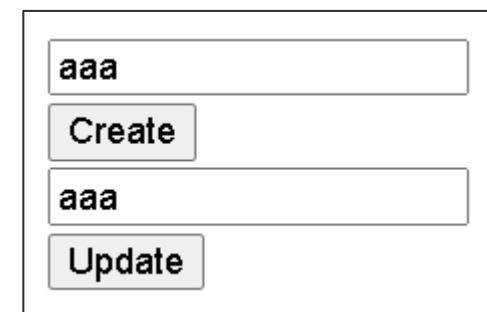
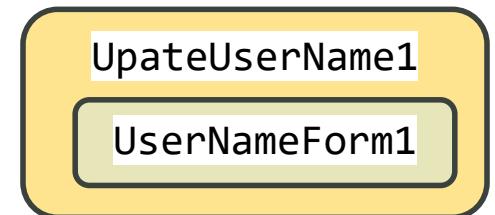
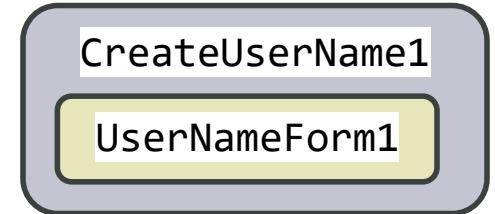
export default App;

```



Composition

```
class UserNameForm1 extends Component {  
    render() {  
        return (  
            <div><input type="text" /></div>  
        )  
    }  
}  
  
export class CreateUserName1 extends Component {  
    render() {  
        return (  
            <div>< UserNameForm1 />  
                <button>Create</button>  
            </div>  
        )  
    }  
}  
  
export class UpdateUserName1 extends Component {  
    render() {  
        return (  
            <div><UserNameForm1 />  
                <button>Update</button>  
            </div>  
        )  
    }  
}
```



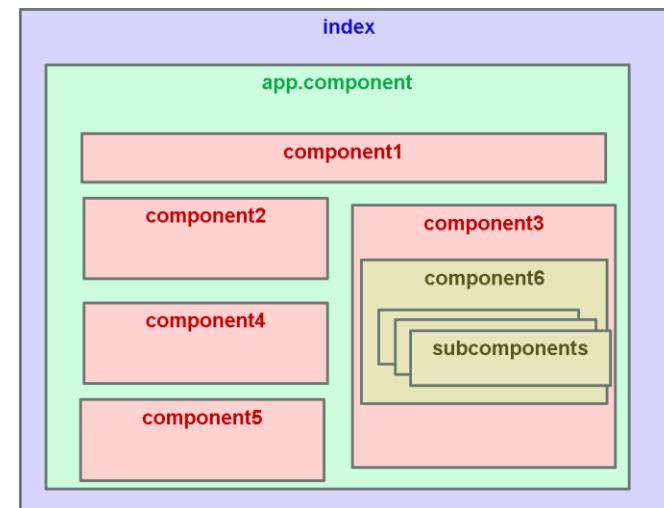
Composition versus inheritance

- In Inheritance, the components are tightly coupled, and therefore any change in the parent component can bring unexpected consequences to its descendant child components.

- Moreover, it causes the complete restructuring of the component tree.

- Component composition in React involves combining smaller, independent components to create complex UIs.
- Each sub-component can contain further components, enhancing code readability.
- This process breaks down the UI into manageable pieces, aiding traceability, scalability, and maintenance.

- Compositions integrate components and keep them loosely coupled.
- That helps to manage future requirements in the application.



```

class App extends Component {
state = {
  date:new Date()
...
}
...
return (
  <div className="container">
    <NewsHeader className="news" subject="Sports"
      date={this.state.date.toString()} />

```

```

const newsHeader = (props) => {
return(
<div>
  <h1>News for {props.date}</h1>
  <h2>News Heading : {props.subject}</h2>
  <NewsContent title="Content Title-1" content="Lots of Content-1" />
  <NewsContent title="Content Title-2" content="Lots of Content-2" />
  <NewsContent title="Content Title-3" content="Lots of Content-3" />

```

```

const newsContent = (props) => {
return(
<div>
  {/*complex code that filters out news based on subject*/}
  <h4><b><i>News Title {props.title}</i></b></h4>
  <h4>News Content : {props.content}</h4>
  <Author title={props.title} name="Shrilata" />

```

News for Sun Jun 1

News Heading : Sports

News Title Content Title-1

News Content : Lots of Content-1

Author for Content Title-1 - Shrilata

News Title Content Title-2

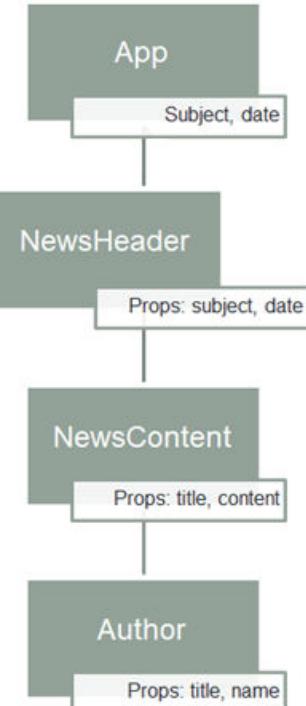
News Content : Lots of Content-2

Author for Content Title-2 - Shrilata

News Title Content Title-3

News Content : Lots of Content-3

Author for Content Title-3 - Shrilata



```

const author = (props) => {
return(
  <h6>Author for {props.title} - {props.name}</h6>

```

CONTEXT API

```
function Component1() {  
  const [user, setUser] = useState("Shrilata T");  
  
  return (  
    <>  
      <h1>`Hello ${user}!`</h1>  
      <Component2 user={user} />  
    </>  
  );  
}  
  
function Component2({ user }) {  
  return (  
    <>  
      <h1>Component 2</h1>  
      <Component3 user={user} />  
    </>  
  );  
}  
  
function Component3({ user }) {  
  return (  
    <>  
      <h1>Component 3</h1>  
      <Component4 user={user} />  
    </>  
  );  
}
```

```
Hello Shrilata T!  
Component 2  
Component 3  
Component 4  
Component 5  
Hello Shrilata T again!
```

```
function Component4({ user }) {  
  return (  
    <>  
      <h1>Component 4</h1>  
      <Component5 user={user} />  
    </>  
  );  
}
```

```
function Component5({ user }) {  
  return (  
    <>  
      <h1>Component 5</h1>  
      <h2>`Hello ${user} again!`</h2>  
    </>  
  );  
}
```

Component1

Component2

Component3

Component4

Component5

Even though components 2-4 did not need the state, they had to pass the state along so that it could reach component 5.

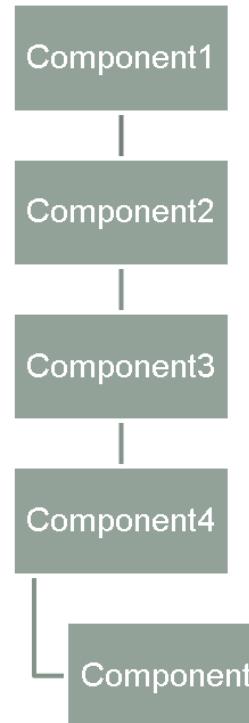
Context API

- The main purpose of Context API is to avoid props drilling.
- Context is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language
- Context API consists of two main components: the context provider and the context consumer.
 - The provider is responsible for creating and managing the context, which holds the data to be shared between components.
 - The consumer is used to access the context and its data from within a component.
 - This avoids the need to pass the information down through props, making your code more efficient and easier to manage.
- Store the state in a Context value in the common ancestor component (called the Provider Component), and access it from as many components as needed (called Consumer Components), which can be nested at any depth under this ancestor.
 - React.js takes care of all the magic behind the scenes to make this work.
- Primary situations where the React Context API really shines are:
 - When your state needs to be accessed or set from deeply nested components.
 - When your state needs to be accessed or set from many child components

Demo

Hello Soha T!
Component 5
Hello again!

SimpleContextAPI
JS CbxComponent1.js
JS CbxComponent5.js
JS UserContext.js



- Step 1 : Create context obj

```
// UserContext.js
import React from 'react';
export default React.createContext();
```

When React renders a component that subscribes to this Context object it will read the current context value from the closest matching Provider above it in the tree

- There are two ways to assign a state to our context.
 - a) By assigning default values when our context object is created.
 - b) Use Provider component in parent component by using value to set state in our context object.
 - For our example, we will assign a default value (" ") while creating our context

```

import { useState } from "react";
import UserContext from "./UserContext";

function CtxComponent1() {
  const [user, setUser] = useState("Soha T");

  return (
    <UserContext.Provider value={user}>
      <h1>`Hello ${user}!`</h1>
      <Component2 user={user} />
    </UserContext.Provider>
  );
}

export default CtxComponent1;

```

```

function CtxComponent5() {
  const user = useContext(UserContext);

  return (
    <>
      <h1>Component 5</h1>
      <h2>`Hello ${user} again!`</h2>
    </>
  );
}

```

- Step-2: wrap the components that need access to the shared data with a Provider component.
 - The Provider component accepts a "value" prop that holds the shared data, and any component that is a child of the Provider component can access that shared data.
 - It's important to note that the Provider component should be wrapped around the top-level component in an application to ensure that all child components have access to the shared data.

- Step-3 :In order to use the Context in a child component, we need to access it using the `useContext` Hook.
 - First, include the `useContext` in the import statement
 - Then you can access the user Context in all components:

Hello Soha T!
Component 5
Hello again!

REDUX

Because state management can be hard

State

- When we discuss state, there are 3 types of state:
 - Local state** : state that belongs to a single component, eg listening to user input in a input field, toggling a “show more” details field, state for our error boundary class etc
 - Cross component state** : state that affects multiple components, eg open/closed state of a modal overlay, ie code to open is model is not inside the modal, though code to close and dispose modal may lie within. For this we might have to do “prop-drilling”.
 - App-wide state** : state that affect the entire app, or state that affects most of the components of app. Eg user authentication status. Eg based on user authentication, some of the components will change their navigation bar etc.
- React Context and Redux – both are meant to resolve cross component or app-wide state.
- As an application grows, it becomes difficult to keep it organized and maintain data flow. That is where Redux comes to the rescue

```
state = {  
  hasError:false,  
  errorMessage:''  
}
```

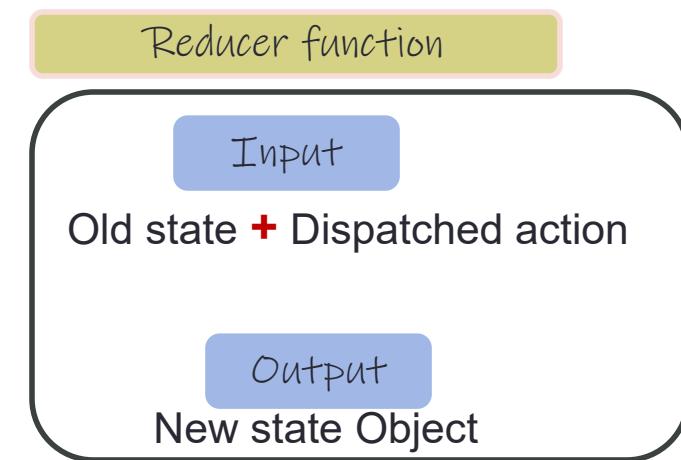
```
class LoggingComponent extends Component {  
  constructor() {  
    this.state = { isLoggedIn: true }  
  }  
}
```

Terms

- **Store**
- **Reducer** : simply a kind of function that will change state of store

- A reducer receives the **current state and an action** object, decides how to update the state if necessary, and returns the new state:

```
const myreducer = (state, action) => newState
```



- **useSelector()** : a react hook that will allow us to fetch from the store
- **useDispatch()** : To update something in store, I need to call reducer. But I cannot call reducer directly, it would break the flow.
 - I will use the useDispatch() hook to send an action to reducer to update state. The useDispatch is used to call a very specific reducer
- An **action** is a plain JavaScript object that has a type field. You can think of an action as an event that describes something that happened in the application.

Simple Node Redux project to understand the various terms

- Create new folder
- npm init -y
- npm install redux

```
//step-1 : create reducer
import { createStore } from "redux";

const reducer = (state = {count:0}, action) => {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + 1 };
    case "DECREMENT":
      return { count: state.count - 1 };
    default:
      return state;
  }
};

//step-2 : create store
const store = createStore(reducer);

//step-3 : create subscriber
const counterSubscriber = () => {
  console.log("State changed:", store.getState());
};
```

```
//step-4 : subscribe to the store
store.subscribe(counterSubscriber);

//step-5 : create actions
const increment =
  () => ({ type: "INCREMENT" });
const decrement =
  () => ({ type: "DECREMENT" });

//step-6 : dispatch actions
store.dispatch(increment());
store.dispatch(decrement());
```

```
PS E:\FreelanceTraining\ReactJ
State changed: { count: 1 }
State changed: { count: 0 }
```

Creating an Vite Redux application

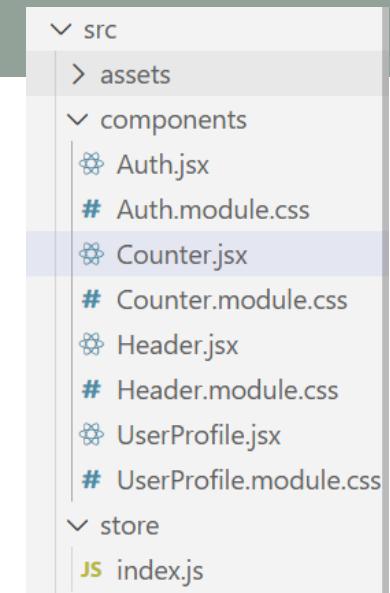
- Step-1 : Create a Redux app using Vite
- Step-2: Install : npm install redux react-redux
 - Copy all contents of component.zip into components folder
- Step-3 : Create a folder called “store” and a index.jsx in it.

```
//src/store/index.js
import { createStore } from "redux";

const initialState = { count: 0 };

const reducer = (state = initialState, action) => {
  switch (action.type) {
    case "INCREMENT":
      return { ...state, count: state.count + 1 };
    case "DECREMENT":
      return { ...state, count: state.count - 1 };
    default:
      return state;
  }
};

const store = createStore(reducer);
export default store;
```



```
//main.jsx
import {Provider} from 'react-redux'
import store from './store/index.js'

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </StrictMode>
)
```

Once the store is created, we can make it available to our React components by putting a React Redux `<Provider>` around our application in `src/index.js`. Import the Redux store we just created, put a `<Provider>` around your `<App>`, and pass the store as a prop

Creating an Vite Redux application

```
import classes from './Counter.module.css';
import { useSelector } from 'react-redux';

const Counter = () => {
  const counter = useSelector((state) => state.counter);

  const toggleCounterHandler = () => {};

  return (
    <main className={classes.counter}>
      <h1>Redux Counter</h1>
      <div className={classes.value}>{counter}</div>
      <button onClick={toggleCounterHandler}>Toggle Counter</button>
    </main>
  );
};

export default Counter;
```



Http Requests

- There are many ways to fetch data from an external API in React:
 - Using `fetch()` API
 - Using `axios` library
 - Using `async / await` syntax
 - Using a Custom React Hook (`useFetch`)
 - Using the React Query Library
- Egs of ready-to-use api's for json data
 - <https://jsonplaceholder.typicode.com/users>
 - <https://randomuser.me/api/>
 - <https://www.anapioficeandfire.com/api/books>

Fetch data from an api in react

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

class FetchData extends Component {

  constructor(props) {
    super(props);
    this.state = {
      users: [],
      dataLoaded: false
    };
  }
  componentDidMount() {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then((res) => res.json())
      .then((json) => {
        this.setState({
          users: json,
          dataLoaded: true
        });
      })
  }
}
```

Fetch data from an api in react : continued

```
render() {
    const { dataLoaded, users } = this.state;
    if (!dataLoaded) return <div>
        <h1> Loading...please wait some time.... </h1> </div>

    return (
        <div className="App">
            <h2> Fetch data from an api in react </h2>
            <table border="1">
                <tr><th>Username</th><th>Fullname</th><th>email</th></tr>
                {users.map(user => (
                    <tr key={user.id} >
                        <td>{user.username}</td>
                        <td>{user.name}</td>
                        <td>{user.email}</td>
                    </tr>
                ))
            }
        </table>
    </div>
);
}

export default FetchData
```

Fetch data from an api in react

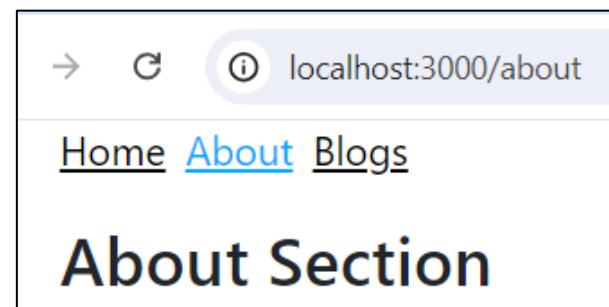
| Username | Fullname | email |
|------------------|--------------------------|---------------------------|
| Bret | Leanne Graham | Sincere@april.biz |
| Antonette | Ervin Howell | Shanna@melissa.tv |
| Samantha | Clementine Bauch | Nathan@yesenia.net |
| Karianne | Patricia Lebsack | Julianne.OConner@kory.org |
| Kamren | Chelsey Dietrich | Lucio_Hettinger@annie.ca |
| Leopoldo_Corkery | Mrs. Dennis Schulist | Karley_Dach@jasper.info |
| Elwyn.Skiles | Kurtis Weissnat | Telly.Hoeger@billy.biz |
| Maxime_Nienow | Nicholas Runolfsdottir V | Sherwood@rosamond.me |
| Delphine | Glenna Reichert | Chaim_McDermott@dana.io |
| Moriah.Stanton | Clementina DuBuque | Rey.Padberg@karina.biz |

REACT ROUTER

React Router v6

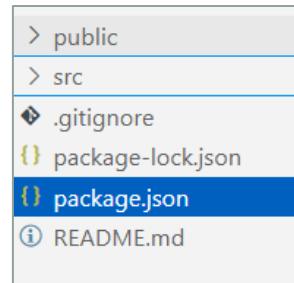
What is React Router

- React Router enables "client side routing".
 - In traditional websites, the browser requests a document from a web server, downloads and evaluates CSS and JavaScript assets, and renders the HTML sent from the server. When the user clicks a link, it starts the process all over again for a new page.
 - Client side routing allows your app to update the URL from a link click without making another request for another document from the server. Instead, your app can immediately render some new UI and make data requests with fetch to update the page with new information.
 - This enables faster user experiences because the browser doesn't need to request an entirely new document or re-evaluate CSS and JavaScript assets for the next page. It also enables more dynamic user experiences with things like animation.



Setting up app

- **Step-1:** Create React App doesn't include page routing. Hence we need to install React Router in our application:
- **npm install react-router-dom**



```
4  private: true,
5  "dependencies": {
6    "@testing-library/jest-dom": "^5.17.0",
7    "@testing-library/react": "^13.4.0",
8    "@testing-library/user-event": "^13.5.0",
9    "react": "^18.3.1",
10   "react-dom": "^18.3.1",
11   "react-router-dom": "^6.26.1",
12   "react-scripts": "5.0.1"
```

- **Step-2 :** To configure React router, navigate to the index.js file, import **BrowserRouter** from the react-router-dom package and wrap it around App component as follows:

```
import { BrowserRouter } from 'react-router-dom';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>
);
```

Defining components of our application

- Step-3 : Create the components we want to navigate:

```
const Home = () => {
  return <h2> Home Section</h2>
}
export default Home
```

```
const About = () => {
  return <h2> About Section</h2>
}
export default About
```

```
const Blogs = () => {
  return <h2> Blogs Section</h2>
}
export default Blogs
```

```
src
└── components
    ├── About.js
    ├── Blogs.js
    ├── Home.js
    └── NavBar.js
```

Defining components of our application

- **Step-4** : Create a standard Navigation bar component that can be used to navigate inside our application.

```
import { NavLink } from "react-router-dom";

const NavBar = () => {
  return (
    <nav>
      <ul>
        <li>
          <NavLink to="/">Home</NavLink>
        </li>
        <li>
          <NavLink to="/about">About</NavLink>
        </li>
        <li>
          <NavLink to="/blogs">Blogs</NavLink>
        </li>
      </ul>
    </nav>
  );
};
```

NavLink component from react-router-dom is a special component that helps you navigate different routes using the **to** prop.

The NavLink component also knows whether the route is currently "active" and adds a default active class to the link. We can use this class in our CSS to define some styling for active links

```
/*index.css  */
ul li a {
  color: #000;
}
ul li a:hover {
  color: #00a8ff;
}
ul li a.active {
  color: #00a8ff;
}
```

Setting up the App.js

- Step-5 : use the Navbar component inside our App
- Step-6: Define/configure our routes
 - An application can have any number of <Routes>

```
import { Route, Routes } from 'react-router-dom';
function App() {
  return (
    <div className="App">
      <NavBar />
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/blogs" element={<Blogs />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </div>
  );
}
```

→ ⌂ ⓘ localhost:3000/about

[Home](#) [About](#) [Blogs](#)

About Section

```
function App() {
  return (
    <div className="App">
      <NavBar />
    ...
  );
}
```

We imported Routes and Route components from `react-router-dom`. Used them to declare the routes we want.

All Routes are wrapped in the `Routes` tag, and these Routes have two major properties:

path: identifies the path we want users to take to reach the set component. When we set the path to `/about`, for example, when the user adds `/about` to the URL link, it navigates to that page.

element: contains the component that we want the set path to load.

REACT AXIOS

Axios

- Many projects on the web need to interface with a REST API at some stage
- Axios is a lightweight HTTP client based on the \$http service within Angular.js v1.x and is similar to the native JavaScript Fetch API.

```
import axios from 'axios';
const baseUrl = 'http://localhost:8080/course/courses'

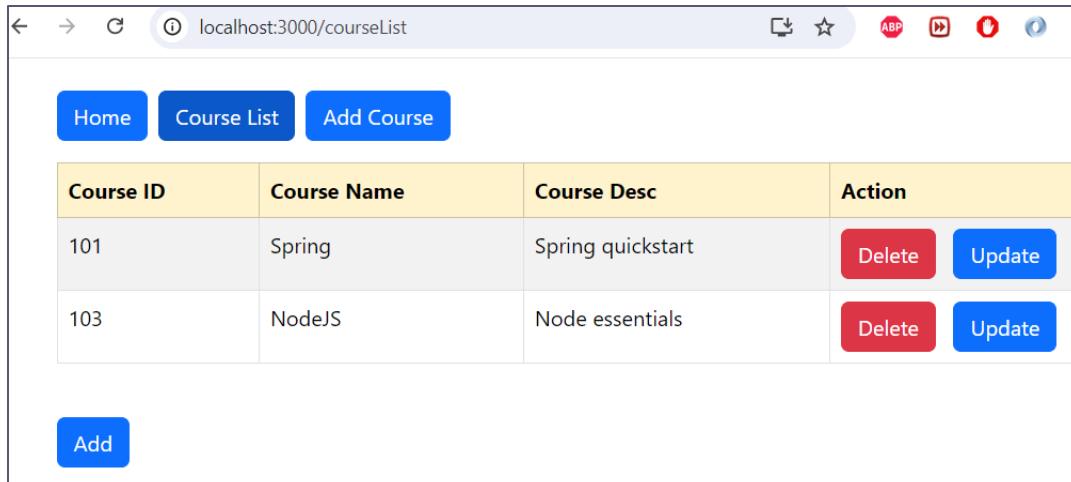
class CourseService{
    getCourses(){
        return axios.get(baseUrl)
    }
    getById(id){
        return axios.get(baseUrl+"/"+id)
    }
    updateCourse(course){
        return axios.put(baseUrl+"/"+course.id, course)
    }
    addCourse(course){
        return axios.post(baseUrl, course)
    }
    deleteById(id){
        return axios.delete(baseUrl+"/"+id)
    }
}
export default new CourseService();
```

- Axios is promise-based, which gives you the ability to take advantage of JavaScript's `async` and `await` for more readable asynchronous code.
- There's built-in client-side protection against cross-site request forgery.

Setting up the app

- Create a new React application
 - `npx create-react-app course-axios-app`
- Install axios :
 - `npm i axios`
- install React Router in our application:
 - `npm install react-router-dom`
- For React router, go to the index.js file, import BrowserRouter from the react-router-dom package and wrap it around App component

```
<BrowserRouter>
  <App />
</BrowserRouter>
```



A screenshot of a web browser window displaying a course list. The URL in the address bar is `localhost:3000/courseList`. The page has a header with three buttons: "Home", "Course List" (which is highlighted in blue), and "Add Course". Below the header is a table with four columns: "Course ID", "Course Name", "Course Desc", and "Action". The table contains two rows of data:

| Course ID | Course Name | Course Desc | Action |
|-----------|-------------|-------------------|-------------------------------------------------|
| 101 | Spring | Spring quickstart | <button>Delete</button> <button>Update</button> |
| 103 | NodeJS | Node essentials | <button>Delete</button> <button>Update</button> |

At the bottom left of the table is a blue "Add" button.