



# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act, 1956)



THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

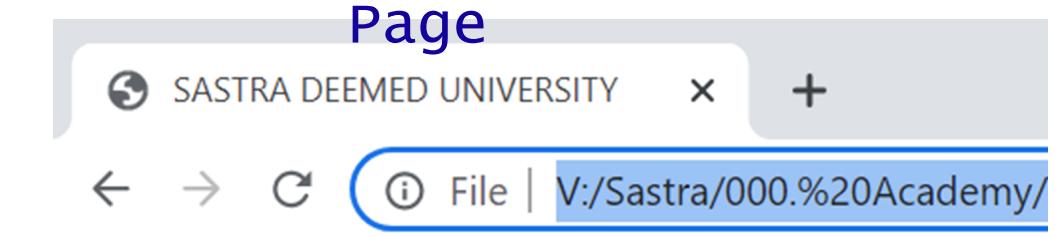
*Topic*  
**HTML - Introduction**

# HTML – HyperText Markup Language

## Sample

```
<html>
<head>
|   <title>SASTRA DEEMED UNIVERSITY</title>
</head>
<body>
|   <H1>Welcome to SASTRA</H1>
</body>
</html>
```

## Sample



# Welcome to SASTRA

# Contents

- Introduction to HTML
- Structure of HTML
- HTML5
- Text formatting
- Links
- Images
- Lists

# Introduction to HTML

- HTML (**HyperText Markup Language**) is the language used to create web pages.
- HTML is the standard markup language for web pages.
- Using HTML, we can create a web page with text, graphics, sound, and video

# Introduction to HTML

- Computers called **web servers** store HTML documents. Clients (such as web browsers running on your local computer or smartphone) request specific resources such as HTML documents from web servers.
- For example, typing [\*\*https://www.sastra.edu/academics/schools.html\*\*](https://www.sastra.edu/academics/schools.html) into a web browser's address field requests the file **schools.html** from the **academics** directory on the web server running at **www.sastra.edu**

# Introduction to HTML

- HTML was created by **Sir Tim Berners-Lee** in late 1991.
- **HTML 1.0** was released in 1993 with the intention of sharing information that can be readable and accessible via web browsers.
- Then comes the **HTML 2.0**, published in 1995, which contains all the features of HTML 1.0 along with that few additional features.
- Then comes the **HTML 3.0**, where Dave Raggett who introduced a fresh paper or draft on HTML. It included improved new features of HTML, giving more powerful characteristics for webmasters in designing web pages.
- But these powerful features of new HTML slowed down the browser
- Then comes **HTML 4.01**, which is widely used and was a successful version of HTML before HTML 5.0, which is currently released and used worldwide.
- **HTML 5** can be said for **an extended version of HTML 4.01**, which was published in the year 2012.

# Introduction to HTML

- In computer text processing, a markup language is a computer language that **uses tags** to define elements within a document. It is human-readable, meaning markup files contain **standard words**, rather than typical programming syntax.
- It is used only to format the text, so that when the document is processed for display, the markup language does not appear.
- Unlike programming languages, mark-up languages are not compiled or interpreted, **just transformed visually**.
- So **even if mistakes are made, no error messages are shown in HTML**, just the expected output will not be seen.

# HTML Editors

- HTML Editor – A word processor that has been specialized to make the **writing of HTML documents more effortless**.
- There are many different programs that you can use to create web documents.
- HTML Editors enable users to **create documents quickly and easily** by pushing a few buttons. Instead of entering all of the HTML codes by hand.
- These programs will generate the HTML Source Code for you.
- Simplest tool to create html documents is **Notepad**.

# HTML Editors

- Adobe Dreamweaver
- Bluefish
- Google Web Designer
- Sublime Text
- CoffeeCup
- HTMLKit
- Mobirise
- Notepad
- Notepad++
- VS Code

## HTML Tags

- The essence of HTML programming is tags
- A tag is a keyword enclosed by angle brackets ( Example: <I> )
- There are opening and closing tags for many but not for all tags; The affected text is between the two tags
- For example, the expression <B> Warning </B> would cause the word ‘warning’ to appear in bold face on a Web page

## Document type

- First step in creating Html document is indicating its type.
- Done with the help of <!DOCTYPE> declaration.
- Required at the beginning of every Html document to help the browser how to render the Html document.
- E.g. <!DOCTYPE Html>

# Structural Elements

- Html elements represent a tree like structure with `<html>` at the root.
- The following elements define the backbone of the structure of a Html document.
- `<html>` - Delimits the html code. It may include lang to specify the human language of the document.
- `<head>` - Used to enclose information necessary to render the page, such as title, character encoding and external files required
- `<body>` - This element defines the content of the document.

```
<!DOCTYPE html>
```

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

```
</html>
```

# Information within <Head>

- **<title>** - define title of page
- **<base>** - defines the URL used by Relative URLs to establish the real location. It includes the Href attribute to declare the URL.
- **<meta>** - represents meta data associated with the document. Used to specify character set, page description, keywords, author of the document, and viewport settings.
- **<link>** - this elements specifies the relationship between the document and an external resource. (usually the css file or js file)
- **<style>** - used to declare the css styles inside the document.
- **<script>** - used to load or declare javascript

# Structure of HTML document

```
<!DOCTYPE html>

<!-- Fig. 2.1: main.html -->
<!-- First HTML5 example. -->
<html>
  <head>
    <meta charset = "utf-8">
    <title>Welcome</title>
  </head>

  <body>
    <p>Welcome to HTML5!</p>
  </body>
</html>
```

Tab shows  
contents of  
`title` element



- HTML code inserted between <HTML> and </HTML> divided into two main sections – Head and body.

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

- within head tag we will define title of the document, define encoding and provide general information.

```
<!DOCTYPE html>
▪ <html lang="en">
  <head>
    <b><title>This text is the title of the document</title></b>
  </head>
  <body>
    </body>
  </html>
```

# Organizing Information: <table> & <div>

- Code between <body> and </body> generates the visible part of the document.
- There are different ways of organizing the information on the document.
- The first element used for this purpose is <table>.
- Tables allow authors to organize data, images, text and tools in rows & columns.
- After CSS, authors started using more practical option called division <div>

# Text Formatting

- Headings : (text size)  
`<h1> <h2> <h3> <h4> <h5> <h6>`  
(Big to Small)
- Alignment: `<center></center>`
- Paragraph: `<p></p>`
- Bold: `<b></b>`
- Italic: `<i></i>`
- Line Breaks: `<br />`
- Horizontal rules: `<hr/>`

# Text Formatting

- <p>This text is normal.</p>
- <b>This text is bold.</b>
- <strong> - Important text </strong>
- <i> - Italic text</i>
- <em> - Emphasized text</em>
- <mark> - Marked text</mark>
- <small> - Smaller text</small>
- <del> - Deleted text</del>
- <ins> - Inserted text</ins>
- <sub> - Subscript text</sub>
- <sup> - Superscript text</sup>

This text is normal.

**This text is bold.**

- **Important text**

- *Italic text*

- *Emphasized text*

- **Marked text**

- **Smaller text**

- **Deleted text**

- **Inserted text**

- **Subscript text**

- **Superscript text**

# Links

- The `<a>` tag defines a hyperlink, which is used to link from one page to another.
- When a user clicks a hyperlink, the browser tries to execute an action associated with it.
- The most important attribute of the `<a>` element is the **href attribute**, which indicates the link's destination.
- By default, links will appear as follows in all browsers:
  - An unvisited link is underlined and blue
  - A visited link is underlined and purple
  - An active link is underlined and red

# Links

- A link lets you move from one page to another, play movies and sound, send email, download files, and more....
- A link has three parts: a destination, a label, and a target
- To create a link type  
`<A HREF="page.html"> label </A>`

# Anatomy of a Link

```
<A HREF="page.html"> label </A>
```

- In the above link, “page.html” is the destination. The destination specifies the address of the web page or file the user will access when he/she clicks on the link.
- The label is the text that will appear underlined or highlighted on the page

## Example: Links

- To create a link to SASTRA, I would type:

```
<A HREF="http://www.sastra.edu">SASTRA</A>
```

- To create a link to CNN, I would type:

```
<A HREF="http://www.cnn.com">CNN</A>
```

- To create a link to MIT, I would type:

```
<A HREF="http://www.mit.edu">MIT</A>
```

# Changing the Color of Links

- The LINK, VLINK, and ALINK attributes can be inserted in the <BODY> tag to define the color of a link
  - VLINK defines the color of links that have not been visited
  - LINK defines the color of links that have already been visited
  - ALINK defines the color of a link when a user clicks on it
  - Example: <body vlink="red">

# Using Links to Send Email

- To create a link to an email address, type <A HREF="mailto:email\_address"> Label</A>
- For example, to create a link to send email to myself, I would type:

```
<A HREF="mailto: tyrvenkat@mca.sastra.edu"> Mail to Me </A>
```

# Inserting Images

- <img> - This is used to add an image into a page.
- Important Attributes:
  - **src** - the path to image file.
  - **alt** - to give the alternate text, if specified image is not available.
  - **height** - pixels/percentage (default is pixels)
  - **width** - pixels/percentage (default is pixels)
  - **border** - border thickness: 1,2,3 and so on.
- Example: 
- Result:



# Alternate Text

- Some browsers don't support images. In this case, the ALT attribute can be used to create text that appears instead of the image.
- Example:

```
<IMG SRC="satellite.jpg" ALT = "Picture of satellite">
```

# Anchors

- Anchors enable a user to jump to a specific place on a web site
- Two steps are necessary to create an anchor. First you must create the anchor itself. Then you must create a link to the anchor from another point in the document.

# Anchors

- To create the anchor itself, type `<A NAME="anchor name">label</A>` at the point in the web page where you want the user to jump to
- To create the link, type `<A HREF="#anchor name">label</A>` at the point in the text where you want the link to appear

## Example: Anchor

```
<A HREF="#chap2">Chapter Two</A><BR>
```



```
<A NAME="chap2">Chapter 2 Anch →  
or
```

## HTML Basic List Elements:

- In a webpage there are so many occasions where we need to use lists. HTML provides us with **three different types**.
  1. Ordered Lists,
  2. Unordered Lists
  3. Definition Lists

# Ordered Lists

- Ordered Lists are lists where each item in the list is numbered
- The ordered list is created with the `<ol>` element.
- Each item in the list is placed between an opening `<li>` tag and a closing `</li>` tag.

```
<ol>
  <li>JAVA</li>
  <li>MS.Net</li>
  <li>C++</li>
  <li>SAP</li>
  <li>Mainframe</li>
</ol>
```

1. JAVA
2. MS.Net
3. C++
4. SAP
5. Mainframe

**Result:**

## More Ordered Lists...

- The TYPE=x attribute allows you to change the kind of symbol that appears in the list.
  - A is for capital letters
  - a is for lowercase letters
  - I is for capital roman numerals
  - i is for lowercase roman numerals

```
<ol type="I">
  <li>One</li>
  <li>Two</li>
  <li>Three</li>
</ol>
```

I. One  
II. Two  
III. Three

# Unordered Lists

- Unordered Lists are lists that begin with a bullet point
- The ordered list is created with the `<ul>` element.
- Each item in the list is placed between an opening `<li>` tag and a closing `</li>` tag.

- ```
<ul>
    <li>JAVA</li>
    <li>MS.Net</li>
    <li>C++</li>
    <li>SAP</li>
    <li>Mainframe</li>
</ul>
```
- **Output:**
  - JAVA
  - MS.Net
  - C++
  - SAP
  - Mainframe

## More Unordered Lists...

- The TYPE=shape attribute allows you to change the type of bullet that appears
  - *circle* corresponds to an empty round bullet
  - *square* corresponds to a square bullet
  - *disc* corresponds to a solid round bullet; this is the default value

# Definition Lists

- Definition Lists are made up of a set of terms along with the definitions for each of those terms
- Created with the `<dl>` element and usually consists of a series of terms and their definitions.
- `<dt>` is used to contain the term being defined (the definition term). `<dd>` is used to contain the definition..
- **Example:**

## □ Result:

```
<dl>
  <dt>Sashimi</dt>
  <dd>Sliced raw fish that is served with
  condiments such as shredded daikon radish or
  ginger root, wasabi and soy sauce</dd>
</dl>
```

Sashimi

Sliced raw fish that is served with condiments such as shredded daikon radish or ginger root, wasabi and soy sauce

# Forms

- An HTML form is an area of the document that allows users to enter information into fields.
- A form may be used to collect personal information, opinions in polls, user preferences and other kinds of information.
- E.g: You will see forms when **registering as a member of a website** and when signing up for newsletters.
- There are **different types for form controls** which can be used to collect information users of website.

## Basic Input/Output GUI elements:

Textbox, Button, Radio button, checkboxes, Dropdown list, List, Textbox with multiple lines, Submit button etc.,

# Thank You



# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY  
(U/S 3 of the UGC Act, 1956)



THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

*Topic*  
**HTML – Form Elements**

# Forms

- Any **input** needed for the web application can be collected from the user through **FORM elements**.
- An HTML form is an area of the document that allows users to enter information into fields.
- A form may be used to collect personal information, opinions in polls, user preferences and other kinds of information.
- E.g: You will see forms when **registering as a member of a website** and when signing up for newsletters.
- There are **different types for form controls** which can be used to collect information users of website.

## Basic Input/Output GUI elements:

Textbox, Button, Radio button, Checkboxes, Dropdown list, List, Textbox with multiple lines, Submit button etc.,

# Create a Form with input controls:

- **<form>** - Form controls live inside a `<form>` element.
  - Attributes: *action*, *method*, *id* and *onsubmit*
- **Action** - Its value is the URL for the page on the server that will receive the information in the form when it is submitted.
- **Onsubmit** - Specifies *javascript function* to validate the inputs before submitting the form. The function should return true or false as a decision for submitting the form.
- **Method** - *'get'* or *'post'*
  - With the get method, the values from the form are added to the end of the URL specified in the action attribute.

# Form Creation Example

```
<html>
<body>
<form
    action="Success.html"
    id="myfrm"
    onsubmit="return myValidation()"
    method="post">

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

# Input Types

- <input> - Used to create several different form controls.
- **Attributes:**
  - ***Id*** - The value of Id is used to identify the form distinctly from other elements on the page.
  - ***Name*** - when users enter information into a form, the server needs to know in which form control each piece of data was entered into. Name will uniquely identify the control.
  - ***MaxLength*** - To limit the number of characters a user may enter into the text field.

## *Some Important Types:*

- <input type="text"> - To Read a Text Input
- <input type="password"> - To Read a Password Input
- <input type="radio"> - Radio Buttons
- <input type="checkbox"> - Check Boxes

```
<input type="text">
```

- Attributes: *Id, Name, value and type*

- *Example*

```
<form id="myform1" action="www.sastra.edu" method="get">
    First name:
    <input type="text" name="firstname" id="firstname">
    <br><br>
    Last name:
    <input type="text" name="lastname" id="lastname">
</form>
```

- *Result*

First name:

Last name:

```
<input type="password">
```

- Attributes: *Id, Name, value and type*

- *Example*

```
<form id="myform2" action="www.sastra.edu" method="get">
  User Name:
  <input type="text" name="uname" id="uname">
  <br><br>
  Password:
  <input type="password" name="pwd" id="pwd">
</form>
```

- *Result*

User Name:

Password:

# <input type="radio">

- Attributes: *Id, Name, value, type and checked*
- *Example*

```
<form id="myform3" action="www.sastra.edu" method="get">
    <input type="radio" id="rad1" name="gender" value="male" checked> Male<br>
    <input type="radio" id="rad2" name="gender" value="female"> Female<br>
    <input type="radio" id="rad3" name="gender" value="other"> Other<br><br>
</form>
```

- *Result*

- Male
- Female
- Other

# <input type="checkbox">

- Attributes: *Id, Name, value, type and checked*

- *Example*

```
<form id="myform4" action="www.sastra.edu" method="get">
    <input type="checkbox" id="ch1" name="v1" value="Bike" checked="true">
    I have a bike
    <br>
    <input type="checkbox" id="ch2" name="v2" value="Car" checked="true">
    I have a car
</form>
```

- *Result*

- I have a bike
- I have a car

# <input type="submit ">

- defines a button for **submitting** form data to a **form-handler**.
- The form-handler is typically a server page with a script for processing input data.
- Attributes: *Id, type and value*

## □ *Example*

```
<form id="myform5" action="www.sastra.edu" method="get" onsubmit="return fun1()">
<input type="submit" value="Submit">
</form>
```

## □ *Result*



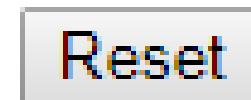
# <input type="reset">

- defines a **reset button** that will reset all form values to their default values:
- Attributes: *Id, type and value*

## □ *Example*

```
<form id="myform5" action="www.sastra.edu" method="get" onsubmit="return fun1()">
<input type="reset" value="Reset">
</form>
```

## □ *Result*



# <input type="button">

- defines a **button** that can perform some actions by calling javascript function
- Attributes: *Id, type, value and onclick*

## □ *Example*

```
<form id="myform7" action="www.sastra.edu" method="get" onsubmit="return fun1()">
<input type="button" value="Click Here" onclick="fun2()">
</form>
```

## □ *Result*

Click Here

# Input Attributes

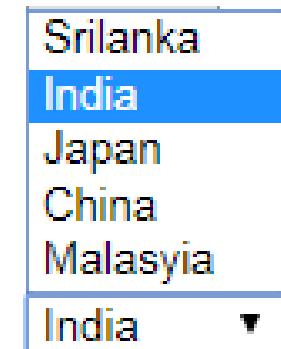
- The ***value*** Attribute - specifies the initial value for an input field
  - <input type="text" name="firstname" value="John">
- The ***readonly*** Attribute - specifies that the input field is read only (cannot be changed)
  - <input type="text" name="firstname" value="John" readonly>
- The ***disabled*** Attribute - specifies that the input field is disabled
  - <input type="text" name="firstname" value="John" disabled>
- The ***size*** Attribute - specifies the size (in characters) for the input field

# Drop-down List in HTML

- Using <select> and <option> tags
- Attributes: Id, onchange for select, value, selected for option
- *Example*

```
<form id="myform6" action="www.sastra.edu" method="get" onsubmit="return fun1()">
<select id="country" onchange="selState()">
    <option value="Srilanka">Srilanka</option>
    <option value="India" selected="true">India</option>
    <option value="Japan">Japan</option>
    <option value="China">China</option>
    <option value="Malasyia">Malasyia</option>
</select>
</form>
```

- *Result*



# Thank You



# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY  
(U/S 3 of the UGC Act, 1956)



THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

*Topic*  
**HTML5 – New Input Types**

# OBJECTIVES

In this chapter you'll:

- Build a form using the new HTML5 **input** types.
- Specify an **input** element in a form as the one that should receive the focus by default.
- Use self-validating **input** elements.
- Specify temporary **placeholder** text in various **input** elements
- Use **autocomplete** **input** elements that help users re-enter text that they've previously entered in a form.
- Use a **datalist** to specify a list of values that can be entered in an **input** element and to autocomplete entries as the user types.
- Use HTML5's new page-structure elements to delineate parts of a page, including headers, sections, figures, articles, footers and more.

### **3.1** Introduction

### **3.2** New HTML5 Form input Types

- 3.2.1 `input Type color`
- 3.2.2 `input Type date`
- 3.2.3 `input Type datetime`
- 3.2.4 `input Type datetime-local`
- 3.2.5 `input Type email`
- 3.2.6 `input Type month`
- 3.2.7 `input Type number`
- 3.2.8 `input Type range`
- 3.2.9 `input Type search`
- 3.2.10 `input Type tel`
- 3.2.11 `input Type time`
- 3.2.12 `input Type url`
- 3.2.13 `input Type week`

## 3.2 New HTML5 Form input Types

- Figure 3.2 demonstrates HTML5's new form input types.
- These are not yet universally supported by all browsers.

```
1 <!DOCTYPE html>
2
3 <!-- Fig. 3.1: newforminputtypes.html -->
4 <!-- New HTML5 form input types and attributes. -->
5 <html>
6   <head>
7     <meta charset="utf-8">
8     <title>New HTML5 Input Types</title>
9   </head>
10
11  <body>
12    <h1>New HTML5 Input Types Demo</h1>
13    <p>This form demonstrates the new HTML5 input types
14      and the placeholder, required and autofocus attributes.
15    </p>
16
17    <form method = "post" action = "http://www.deitel.com">
18      <p>
19        <label>Color:
20          <input type = "color" autofocus />
21          (Hexadecimal code such as #ADD8E6)
22        </label>
23      </p>
```

**Fig. 3.1** | New HTML5 form input types and attributes. (Part 1 of 5.)

```
24      <p>
25          <label>Date:
26              <input type = "date" />
27                  (yyyy-mm-dd)
28          </label>
29      </p>
30      <p>
31          <label>Datetime:
32              <input type = "datetime" />
33                  (yyyy-mm-ddThh:mm+ff:gg, such as 2012-01-27T03:15)
34          </label>
35      </p>
36      <p>
37          <label>Datetime-local:
38              <input type = "datetime-local" />
39                  (yyyy-mm-ddThh:mm, such as 2012-01-27T03:15)
40          </label>
41      </p>
42      <p>
43          <label>Email:
44              <input type = "email" placeholder = "name@domain.com"
45                  required /> (name@domain.com)
46          </label>
47      </p>
```

**Fig. 3.1** | New HTML5 form input types and attributes. (Part 2 of 5.)

```
48      <p>
49          <label>Month:
50              <input type = "month" /> (yyyy-mm)
51          </label>
52      </p>
53      <p>
54          <label>Number:
55              <input type = "number"
56                  min = "0"
57                  max = "7"
58                  step = "1"
59                  value = "4" />
60          </label> (Enter a number between 0 and 7)
61      </p>
62      <p>
63          <label>Range:
64              0 <input type = "range"
65                  min = "0"
66                  max = "20"
67                  value = "10" /> 20
68          </label>
69      </p>
```

**Fig. 3.1** | New HTML5 form input types and attributes. (Part 3 of 5.)

```
70      <p>
71          <label>Search:
72              <input type = "search" placeholder = "search query" />
73          </label> (Enter your search query here.)
74      </p>
75      <p>
76          <label>Tel:
77              <input type = "tel" placeholder = "(###) ###-####"
78                  pattern = "\(\d{3}\) +\d{3}-\d{4}" required />
79                  (###) ###-####
80          </label>
81      </p>
82      <p>
83          <label>Time:
84              <input type = "time" /> (hh:mm:ss.ff)
85          </label>
86      </p>
87      <p>
88          <label>URL:
89              <input type = "url"
90                  placeholder = "http://www.domainname.com" />
91                  (http://www.domainname.com)
92          </label>
93      </p>
```

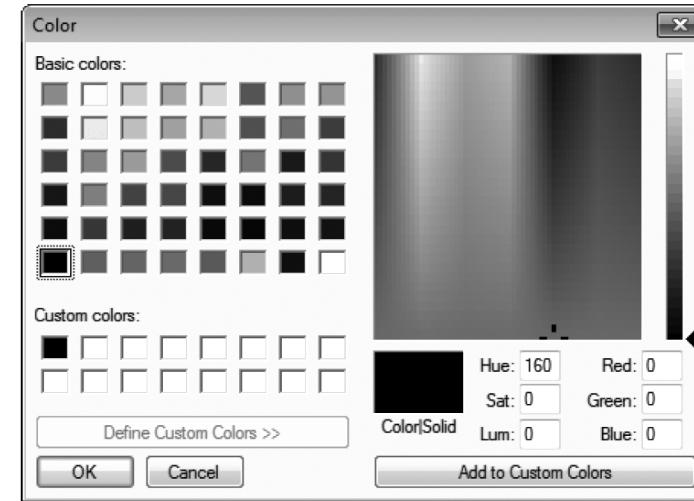
**Fig. 3.1** | New HTML5 form input types and attributes. (Part 4 of 5.)

```
94      <p>
95          <label>Week:
96              <input type = "week" />
97                  (yyyy-Wnn, such as 2012-W01)
98          </label>
99      </p>
100     <p>
101         <input type = "submit" value = "Submit" />
102         <input type = "reset" value = "Clear" />
103     </p>
104     </form>
105 </body>
106 </html>
```

**Fig. 3.1** | New HTML5 form input types and attributes. (Part 5 of 5.)

### 3.2.1 input Type color

- The `color` input type enables the user to enter a color.
- At the time of this writing, most browsers render the color input type as a text field in which the user can enter a hexadecamal code or a color name.
- In the future, when you click a color input, browsers will likely display a *color picker* similar to the Microsoft Windows color dialog shown in Fig. 3.2.



**Fig. 3.2** | A dialog for choosing colors.

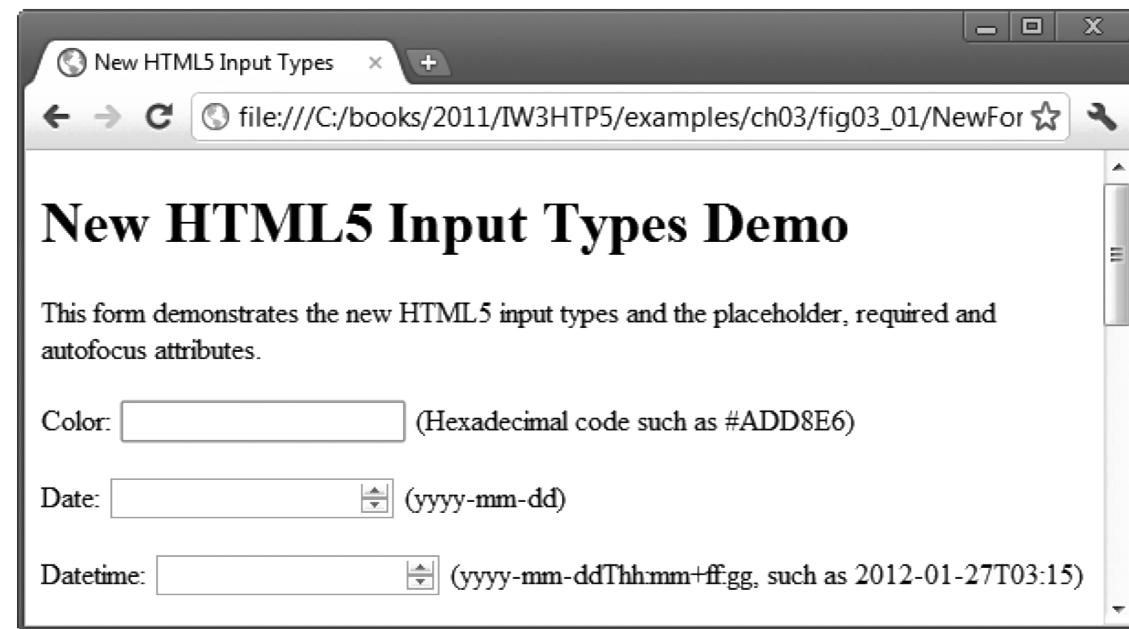
### 3.2.1 input Type color

#### *autofocus Attribute*

- The **autofocus attribute**—an optional attribute that can be used in only one input element on a form—automatically gives the focus to the input element, allowing the user to begin typing in that element immediately.

### 3.2.1 input Type color (cont.)

- Figure 3.3 shows autofocus on the color element—the first input element in our form—as rendered in Chrome. You do not need to include autofocus in your forms.

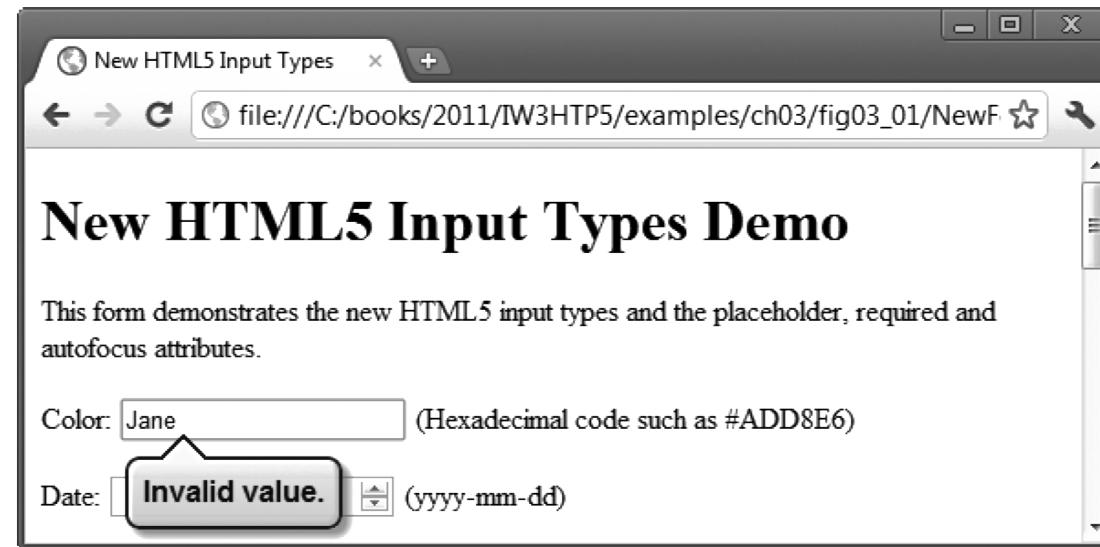


**Fig. 3.3** | Autofocus in the color input element using Chrome.

### 3.2.1 input Type color (cont.)

#### *validation*

- The new HTML 5 input types are *self validating* on the client side, eliminating the need to add complicated JavaScript code to your web pages to validate user input, reducing the amount of invalid data submitted and consequently reducing Internet traffic between the server and the client to correct invalid input.
- *The server should still validate all user input.*
- When a user enters data into a form then submits the form the browser immediately checks the self-validating elements to ensure that the data is correct (Fig. 3.4).



**Fig. 3.4 |** Validating a color input in Chrome.

### 3.2.1 input Type color (cont.)

- Figure 3.5 lists each of the new HTML5 input types and provides examples of the proper formats required for each type of data to be valid.

input type	Format
color	Hexadecimal code
date	yyyy-mm-dd
datetime	yyyy-mm-dd
datetime-local	yyyy-mm-ddThh:mm
month	yyyy-mm
number	Any numerical value
email	name@domain.com
url	http://www.domain-name.com
time	hh:mm
week	yyyy-Wnn

**Fig. 3.5 | Self-validating input types.**

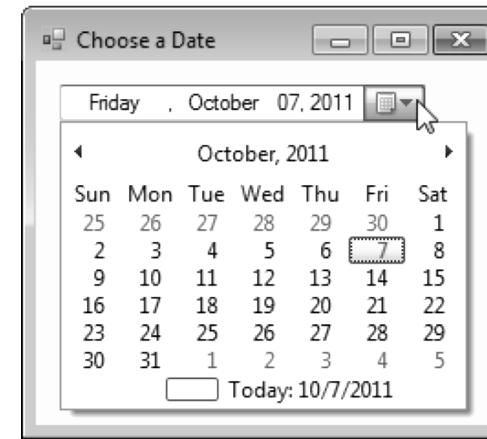
### 3.2.1 input Type color (cont.)

- If you want to bypass validation, you can add the **formnovalidate** attribute to input type submit in line 101:

```
<input type = "submit" value = "Submit" formnovalidate />
```

### 3.2.2 input Type date

- The **date input type** enables the user to enter a date in the form yyyy-mm-dd.
- Firefox and Internet Explorer display a text field in which a user can enter a date such as 2012-01-27.
- Chrome and Safari display a **spinner control**-a text field with an up-down arrow () on the right side-allowing the user to select a date by clicking the up or down arrow.
- The start date is the *current date*.
- Opera displays a calendar from which you can choose a date.
- In the future, when the user clicks a date input, browsers are likely to display a date control similar to the Microsoft windows one shown in Fig. 3.6.



**Fig. 3.6** | A date chooser control.

### 3.2.3 input Type `datetime`

- The `datetime` `input type` enables the user to enter a date (year, month, day), time (hour, minute, second, fraction of a second) and the time zone set to UTC (Coordinated Universal Time or Universal Time, Coordinated).
- Currently, most of the browsers render `datetime` as a text field; Chrome renders an up-down control and Opera renders a date and time control.

### 3.2.4 input Type `datetime-local`

- The `datetime-local` `input type` enables the user to enter the date and time in a *single* control.
- The data is entered as year, month, day, hour, minute, second and fraction of a second.
- Internet Explorer, Firefox and Safari all display a text field.
- Opera displays a date and time control.

### 3.2.5 input Type email

- The `email` `input type` enables the user to enter an e-mail address or a list of e-mail addresses separated by commas (if the `multiple` attribute is specified).
- Currently, all of the browsers display a text field.
- If the user enters an *invalid* e-mail address (i.e., the text entered is *not* in the proper format) and clicks the Submit button, a callout asking the user to enter an e-mail address is rendered pointing to the input element (Fig. 3.7).
- HTML5 does not check whether an e-mail address entered by the user actually exists—rather it just validates that the e-mail address is in the *proper format*.



**Fig. 3.7 |** Validating an e-mail address in Chrome.

### 3.2.5 input Type email (cont.)

#### *placeholder Attribute*

- The **placeholder attribute** allows you to place temporary text in a text field.
- Generally, placeholder text is *light gray* and provides an example of the text and/or text format the user should enter (Fig. 3.8).
- When the *focus* is placed in the text field (i.e., the cursor is in the text field), the placeholder text disappears—it's not “submitted” when the user clicks the Submit button (unless the user types the same text).

a) Text field with gray placeholder text



b) placeholder text disappears when the text field gets the focus



**Fig. 3.8 |** placeholder text disappears when the input element gets the focus.

### 3.2.5 input Type email (cont.)

- HTML5 supports placeholder text for only six input types—text, search, url, tel, email and password.

#### *required Attribute*

- The **required attribute** forces the user to enter a value before submitting the form.
- You can add required to any of the input types.
- In this example, the user *must* enter an e-mail address and a telephone number to submit the form (Fig. 3.9).

New HTML5 Input Types Demo

This form demonstrates the new HTML5 input types and the placeholder, required and autofocus attributes.

Color:  (Hexadecimal code such as #ADD8E6)

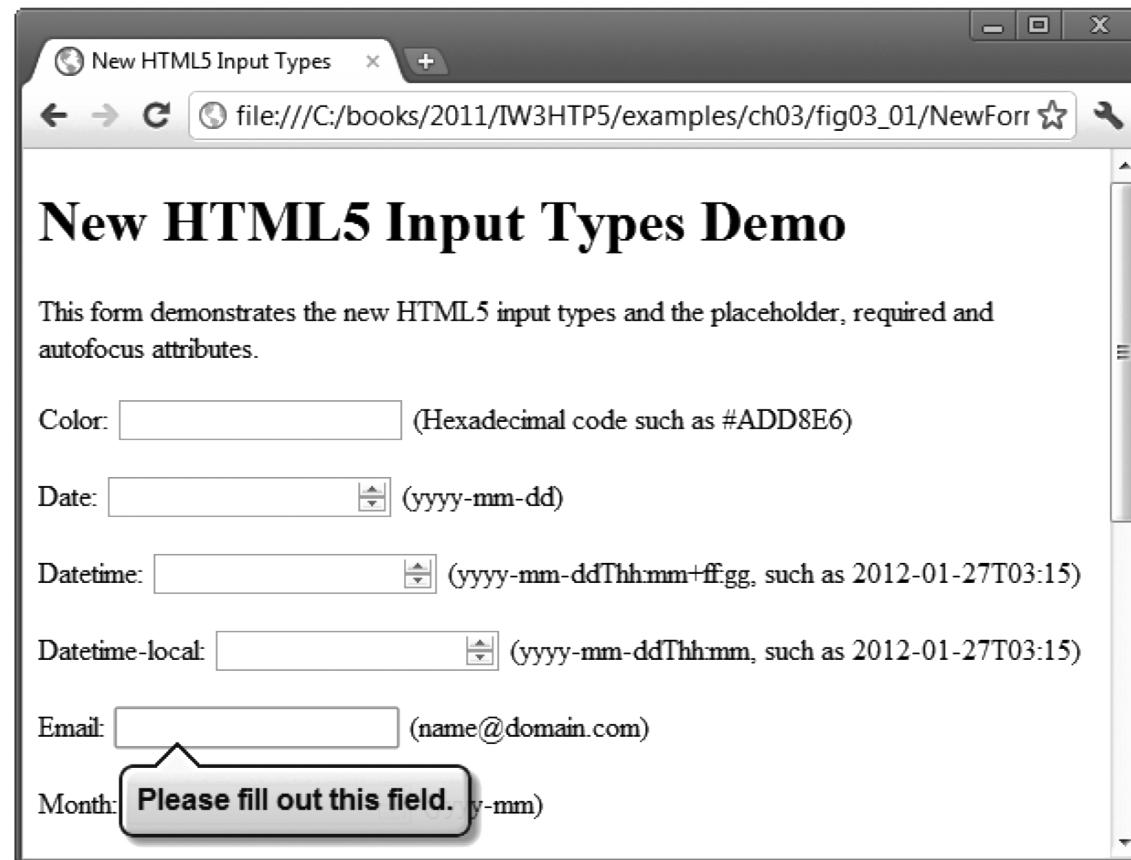
Date:  (yyyy-mm-dd)

Datetime:  (yyyy-mm-ddThhmm+ffgg, such as 2012-01-27T03:15)

Datetime-local:  (yyyy-mm-ddThhmm, such as 2012-01-27T03:15)

Email:  (name@domain.com)

Month:  Please fill out this field. (y-mm)



**Fig. 3.9 |** Demonstrating the required attribute in Chrome.

### 3.2.6 input Type month

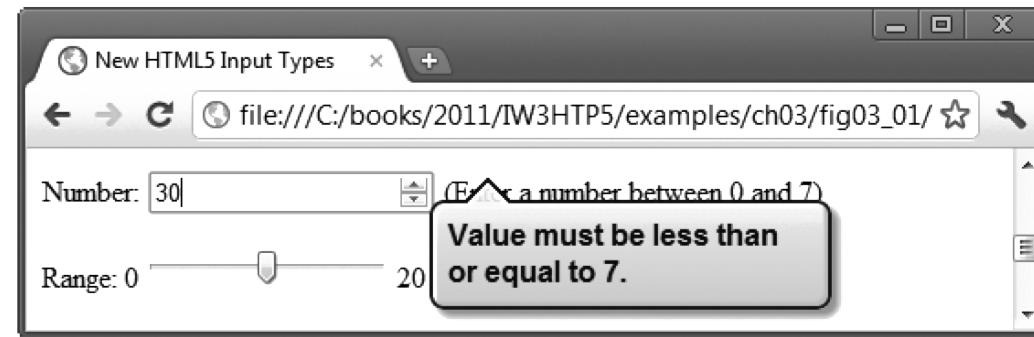
- The `month` `input type` enables the user to enter a year and month in the format yyyy-mm, such as 2012-01.
- If the user enters the data in an improper format (e.g., January 2012) and submits the form, a callout stating that an invalid value was entered appears.

### 3.2.7 input Type number

- The `number` `input type` enables the user to enter a numerical value—mobile browsers typically display a numeric keypad for this input type.
- Internet Explorer, Firefox and Safari display a text field in which the user can enter a number. Chrome and Opera render a spinner control for adjusting the number.
- The `min` attribute sets the minimum valid number.
- The `max` attribute sets the maximum valid number.
- The `step` attribute determines the increment in which the numbers increase.
- The `value` attribute sets the initial value displayed in the form (Fig. 3.20).
- The spinner control includes only the valid numbers.
- If the user attempts to enter an invalid value by typing in the text field, a callout pointing to the number input element will instruct the user to enter a valid value.



**Fig. 3.10** | `input` type `number` with a `value` attribute of 4 as rendered in Chrome.



**Fig. 3.11** | Chrome checking for a valid number.

### 3.2.8 input Type range

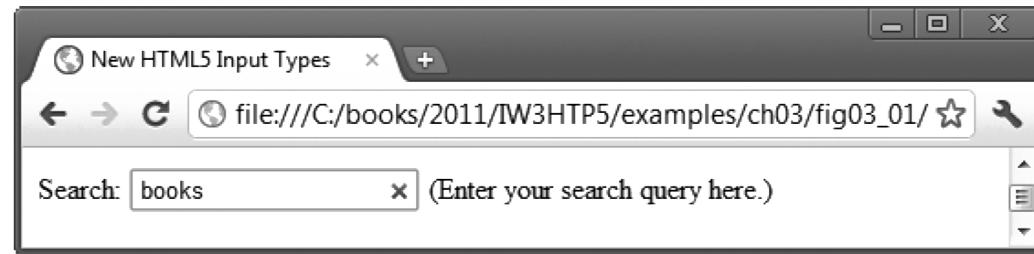
- The **range input type** appears as a *slider* control in Chrome, Safari and Opera (Fig. 3.22).
- You can set the minimum and maximum and specify a value.
- The range input type is *inherently self-validating* when it is rendered by the browser as a slider control, because *the user is unable to move the slider outside the bounds of the minimum or maximum value*.



**Fig. 3.12** | range slider with a `value` attribute of 10 as rendered in Chrome.

### 3.2.9 input Type search

- The **search** input type provides a search field for entering a query.
- This input element is functionally equivalent to an input of type text.
- When the user begins to type in the search field, Chrome and Safari display an X that can be clicked to clear the field (Fig. 3.23).
-



**Fig. 3.13** | Entering a search query in Chrome.

### 3.2.10 input Type tel

- The `tel` input type enables the user to enter a telephone number—mobile browsers typically display a keypad specific to entering phone numbers for this input type.
- At the time of this writing, the `tel` input type is rendered as a text field in all of the browsers.
- HTML5 does *not* self validate the `tel` input type.
- To ensure that the user enters a phone number in a proper format, we've added a pattern attribute that uses a *regular expression* to determine whether the number is in the format:
  - (555) 555-5555
- when the user enters a phone number in the wrong format, a callout appears requesting the proper format, pointing to the `tel` input element (Fig. 3.24).
-



**Fig. 3.14** | Validating a phone number using the pattern attribute in the tel input type.

### 3.2.11 input Type time

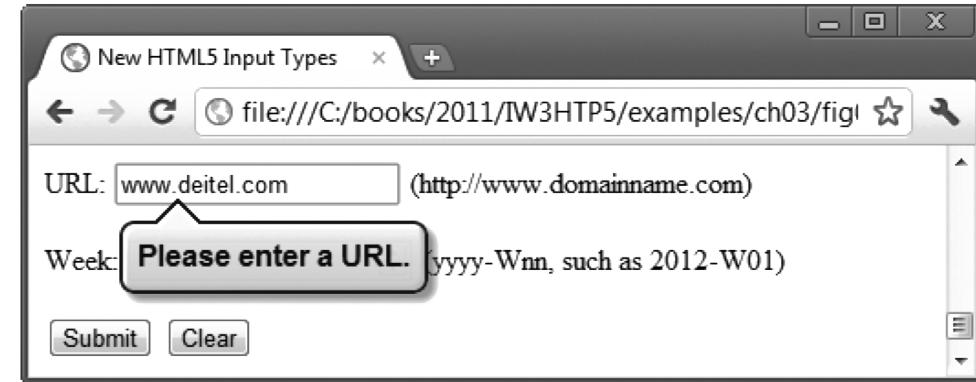
- The **time** **input type** enables the user to enter an hour, minute, seconds and fraction of second (Fig. 3.25).
- The HTML5 specification indicates that a time must have two digits representing the hour, followed by a colon (:) and two digits representing the minute.
- Optionally, you can also include a colon followed by two digits representing the seconds and a period followed by one or more digits representing a fraction of a second (shown as ff in our sample text to the right of the time input element in Fig. 3.25).



**Fig. 3.15** | time input as rendered in Chrome.

### 3.2.12 input Type url

- The `url` `input type` enables the user to enter a URL.
- The element is rendered as a text field, and the proper format is `http://www.deitel.com`.
- If the user enters an improperly formatted URL (e.g., `www.deitel.com` or `www.deitelcom`), the URL will *not* validate (Fig. 3.26).
- HTML5 does not check whether the URL entered is valid; rather it validates that the URL entered is in the proper format.



**Fig. 3.16 |** Validating a URL in Chrome.

### 3.2.13 input Type week

- The `week` `input type` enables the user to select a year and week number in the format `yyyy-wnn`, where `nn` is 01-53—for example, `2012-w01` represents the first week of 2012. Internet Explorer, Firefox and Safari render a text field.
- Chrome renders an up-down control.
- Opera renders `week control` with a down arrow that, when clicked, brings up a calendar for the current month with the corresponding week numbers listed down the left side.

## 3.2 `input` and `datalist` Elements and `autocomplete` Attribute

- Figure 3.27 shows how to use the new `autocomplete` attribute and `datalist` element.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 3.17: autocomplete.html -->
4  <!-- New HTML5 form autocomplete attribute and datalist element. -->
5  <html>
6      <head>
7          <meta charset="utf-8">
8          <title>New HTML5 autocomplete Attribute and datalist Element</title>
9      </head>
10
11     <body>
12         <h1>Autocomplete and Datalist Demo</h1>
13         <p>This form demonstrates the new HTML5 autocomplete attribute
14             and the datalist element.
15         </p>
16
17         <!-- turn autocomplete on -->
18         <form method = "post" autocomplete = "on">
19             <p><label>First Name:
20                 <input type = "text" id = "firstName"
21                     placeholder = "First name" /> (First name)
22             </label></p>
```

**Fig. 3.17 |** New HTML5 form autocomplete attribute and  
datalist element. (Part I of 6.)

```
23      <p><label>Last Name:  
24          <input type = "text" id = "lastName"  
25              placeholder = "Last name" /> (Last name)  
26          </label></p>  
27      <p><label>Email:  
28          <input type = "email" id = "email"  
29              placeholder = "name@domain.com" /> (name@domain.com)  
30          </label></p>  
31      <p><label for = "txtList">Birth Month:  
32          <input type = "text" id = "txtList"  
33              placeholder = "Select a month" list = "months" />  
34          <datalist id = "months">  
35              <option value = "January">  
36              <option value = "February">  
37              <option value = "March">  
38              <option value = "April">  
39              <option value = "May">  
40              <option value = "June">  
41              <option value = "July">  
42              <option value = "August">  
43              <option value = "September">  
44              <option value = "October">  
45              <option value = "November">
```

**Fig. 3.17** | New HTML5 form autocomplete attribute and datalist element. (Part 2 of 6.)

```
46          <option value = "December">
47      </datalist>
48  </label></p>
49  <p><input type = "submit" value = "Submit" />
50      <input type = "reset" value = "Clear" /></p>
51  </form>
52 </body>
53 </html>
```

**Fig. 3.17** | New HTML5 form autocomplete attribute and datalist element. (Part 3 of 6.)

a) Form rendered in Firefox before the user interacts with it

The screenshot shows a Firefox browser window with the title "New HTML5 autocomplete Attribute an...". The address bar displays "file:///C:/books/2011/IW3HTP5/examples/". The main content area contains the following text and form fields:

**Autocomplete and Datalist Demo**

This form demonstrates the new HTML5 autocomplete attribute and the datalist element.

First Name:  (First name)

Last Name:  (Last name)

Email:  (name@domain.com)

Birth Month:

**Fig. 3.17 |** New HTML5 form autocomplete attribute and datalist element. (Part 4 of 6.)

b) autocomplete automatically fills in the data when the user returns to a form submitted previously and begins typing in the **First Name** input element; clicking Jane inserts that value in the input

The screenshot shows a Firefox browser window with the title "New HTML5 autocomplete Attribute an...". The address bar indicates the page is located at "file:///C:/books/2011/IW3HTP5/examples/". The main content area displays a heading "Autocomplete and Datalist Demo" followed by a descriptive text: "This form demonstrates the new HTML5 autocomplete attribute and the datalist element." Below this, there are four form fields:

- First Name:** An input field containing "J" with the placeholder "(First name)".
- Last Name:** An input field containing "Jane" with the placeholder "(Last name)".
- Email:** An input field containing "name@domain.com" with the placeholder "(name@domain.com)".
- Birth Month:** A dropdown menu labeled "Select a month".

At the bottom of the form are two buttons: "Submit" and "Clear".

**Fig. 3.17 |** New HTML5 form autocomplete attribute and datalist element. (Part 5 of 6.)

c) autocomplete with a datalist showing the previously entered value (June) followed by all items that match what the user has typed so far; clicking an item in the autocomplete list inserts that value in the input

datalist values filtered by what's been typed so far

The screenshot shows a Firefox browser window with the title "New HTML5 autocomplete Attribute an...". The page content is titled "Autocomplete and Datalist Demo". It contains a form with four fields: "First Name" (Jane), "Last Name" (Blue), "Email" (jane@domain.com), and "Birth Month" (input field containing "j" with a dropdown menu showing "June", "January", "June", and "July"). A legend on the left explains the "dolist" values as being filtered by what's been typed so far.

First Name: Jane (First name)

Last Name: Blue (Last name)

Email: jane@domain.com (name@domain.com)

Birth Month: j

Submit

June  
January  
June  
July

**Fig. 3.17 |** New HTML5 form autocomplete attribute and datalist element. (Part 6 of 6.)

## 3.2.1 input Element autocomplete Attribute

- The **autocomplete attribute** can be used on input types to automatically fill in the user's information based on previous input—such as name, address or e-mail.
- You can enable autocomplete for an entire form or just for specific elements.
- For example, an online order form might set `autocomplete = "on"` for the name and address inputs and set `autocomplete = "off"` for the credit card and password inputs for security purposes.



### Error-Prevention Tip 3.1

The `autocomplete` attribute works only if you specify a `name` or `id` attribute for the `input` element.

## 3.2.2 `datalist` Element

- The `datalist` element provides input options for a text input element.
- At the time of this writing, `datalist` support varies by browser.
- In this example, we use a `datalist` element to obtain the user's birth month.
- Using Opera, when the user clicks in the text field, a drop-down list of the months of the year appears. If the user types "M" in the text field, the list of months is narrowed to March and May.
- When using Firefox, the drop-down list of months appears only after the user begins typing in the text field. If the user types "M", all months containing the letter "M" or "m" appear in the drop-down list—March, May, September, November and December.

```
<p>
    Academic Year:
    <input type="text" list="autolist">
    <datalist id="autolist">
        <option value="2021-2022">
        <option value="2022-2023">
        <option value="2023-2024">
        <option value="2024-2025">
        <option value="2025-2026">
    </datalist>
</p>
```

Academic Year:  ▼

2024-2025

2025-2026

### 3.2.3 pattern Property

- The pattern attribute specifies a regular expression that the <input> element's value is checked against on form submission.
- The pattern attribute works with the following input types: text, date, search, url, tel, email, and password.

```
<p>
|   Name: <input type="text" pattern="[a-zA-Z]{2,} [a-zA-Z]+" />
</p>
```

```
<p>
|   Telephone: <input type="tel" pattern="\(\d{3}\)\d{9}" />
</p>
```



# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY  
(U/S 3 of the UGC Act, 1956)



THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

*Topic*  
**HTML5 – Page Structure Elements**

# Topics

- ü table Element
- ü fieldset Element
- ü div Element
- ü section Element
- ü header Element
- ü footer Element
- ü figure & figcaption Element

# Tables

- A table represents information in a grid format.
- Mainly used in alignment of controls in web page.
- To create a basic table structure you need to use `<table>`, `<tr>` `<th>` and `<td>`. `<tr>` is used to create row and `<td>` will be used to create column.

# Tables

```
<table>
  <tr>
    <th>Header 1</th>
    <th>Header 2</th>
    <th>Header 3</th>
    <th>Header 4</th>
  </tr>
  <tr>
    <td>Data 1, 1</td>
    <td>Data 1, 2</td>
    <td>Data 1, 3</td>
    <td>Data 1, 4</td>
  </tr>
</table>
```

Header 1	Header 2	Header 3	Header 4
Data 1, 1	Data 1, 2	Data 1, 3	Data 1, 4

# Tables

```
<table>
  <tr>
    <th>Header 1</th>
    <th>Header 2</th>
    <th>Header 3</th>
    <th>Header 4</th>
  </tr>
  <tr>
    <td>Data 1, 1</td>
    <td>Data 1, 2</td>
    <td>Data 1, 3</td>
    <td>Data 1, 4</td>
  </tr>
  <tr>
    <td>Data 2, 1</td>
    <td>Data 2, 2</td>
    <td></td>
    <td>Data 2, 4</td>
  </tr>
</table>
```

Empty cell

Header 1	Header 2	Header 3	Header 4
Data 1, 1	Data 1, 2	Data 1, 3	Data 1, 4
Data 2, 1	Data 2, 2		Data 2, 4

# Tables

- Important Attributes:
- **<table>**: border, align, width, height etc.,
  - **Border** – Border width. A number.
- **<td>**: valign, halign, rowspan, colspan, width, height etc.,
  - **Align**, **valign** (vertical align) and **halign** (horizontal align): – center, left, right
  - **Rowspan** – to merge rows. A number.
    - Eg: if rowspan=2 for a particular td, current td will be merged with previous 1 row's td.
  - **Colspan** – to merge columns. A number.
    - Eg: if colspan=2 for a particular td, current

# Tables

```
<table border="1">
  <tr>
    <th>Header 1</th>
    <th>Header 2</th>
    <th>Header 3</th>
    <th>Header 4</th>
  </tr>
  <tr>
    <td>Data 1, 1</td>
    <td>Data 1, 2</td>
    <td>Data 1, 3</td>
    <td>Data 1, 4</td>
  </tr>
  <tr>
    <td>Data 2, 1</td>
    <td>Data 2, 2</td>
    <td></td>
    <td>Data 2, 4</td>
  </tr>
</table>
```

Border  
Property

Header 1	Header 2	Header 3	Header 4
Data 1, 1	Data 1, 2	Data 1, 3	Data 1, 4
Data 2, 1	Data 2, 2		Data 2, 4

# Tables

```
<table border="1" style="border-collapse: collapse">
  <tr>
    <th>Header 1</th>
    <th>Header 2</th>
    <th>Header 3</th>
    <th>Header 4</th>
  </tr>
  <tr>
    <td>Data 1, 1</td>
    <td>Data 1, 2</td>
    <td>Data 1, 3</td>
    <td>Data 1, 4</td>
  </tr>
  <tr>
    <td>Data 2, 1</td>
    <td>Data 2, 2</td>
    <td></td>
    <td>Data 2, 4</td>
  </tr>
</table>
```

CSS: Border  
Collapse

Header 1	Header 2	Header 3	Header 4
Data 1, 1	Data 1, 2	Data 1, 3	Data 1, 4
Data 2, 1	Data 2, 2		Data 2, 4

# Tables

```
<table border="1" style="border-collapse: collapse">
  <tr height="40px">
    <th width="100px">Header 1</th>
    <th width="100px">Header 2</th>
    <th width="100px">Header 3</th>
    <th width="100px">Header 4</th>
  </tr>
  <tr>
    <td>Data 1, 1</td>
    <td>Data 1, 2</td>
    <td>Data 1, 3</td>
    <td>Data 1, 4</td>
  </tr>
  <tr>
    <td>Data 2, 1</td>
    <td>Data 2, 2</td>
    <td></td>
    <td>Data 2, 4</td>
  </tr>
</table>
```

Specifying height for Head Row

Specifying width for the columns

Header 1	Header 2	Header 3	Header 4
Data 1, 1	Data 1, 2	Data 1, 3	Data 1, 4
Data 2, 1	Data 2, 2		Data 2, 4

# Tables

```
<table border="1" style="border-collapse: collapse">
    <tr height="40px">
        <th width="100px">Header 1</th>
        <th width="100px">Header 2</th>
        <th width="100px">Header 3</th>
        <th width="100px">Header 4</th>
    </tr>
    <tr height="40px">
        <td>Data 1, 1</td>
        <td>Data 1, 2</td>
        <td>Data 1, 3</td>
        <td>Data 1, 4</td>
    </tr>
    <tr height="40px">
        <td>Data 2, 1</td>
        <td>Data 2, 2</td>
        <td></td>
        <td>Data 2, 4</td>
    </tr>
</table>
```

Specifying Height for all rows

Header 1	Header 2	Header 3	Header 4
Data 1, 1	Data 1, 2	Data 1, 3	Data 1, 4
Data 2, 1	Data 2, 2		Data 2, 4

# Tables

```
<table border="1" style="border-collapse:  
collapse">
```

```
    <tr height="40px">  
        <th width="200px" colspan="2">Header
```

```
1</th>
```

```
        <th width="100px">Header 3</th>
```

```
        <th width="100px">Header 4</th>
```

```
</tr>
```

```
<tr height="40px">
```

```
    <td>Data 1, 1</td>
```

```
    <td>Data 1, 2</td>
```

```
    <td>Data 1, 3</td>
```

```
    <td>Data 1, 4</td>
```

```
</tr>
```

```
<tr height="40px">
```

```
    <td>Data 2, 1</td>
```

```
    <td>Data 2, 2</td>
```

```
    <td></td>
```

```
    <td>Data 2, 4</td>
```

```
</tr>
```

```
</table>
```

Merging cells  
from two  
columns

Next column in  
that row should  
be empty

Header 1	Header 3	Header 4
Data 1, 1	Data 1, 2	Data 1, 3
Data 2, 1	Data 2, 2	Data 2, 4

# Tables

```
<table border="1" style="border-collapse: collapse">
    <tr height="40px">
        <th width="200px" colspan="2">Header
1</th>
        <th width="100px">Header 3</th>
        <th width="100px">Header 4</th>
    </tr>
    <tr height="40px">
        <td>Data 1, 1</td>
        <td>Data 1, 2</td>
        <td rowspan="2">Data 1, 3</td>
        <td>Data 1, 4</td>
    </tr>
    <tr height="40px">
        <td>Data 2, 1</td>
        <td>Data 2, 2</td>
        <td>Data 2, 3</td>
        <td>Data 2, 4</td>
    </tr>
</table>
```

Merging cells  
from two rows

In next row,  
corresponding  
cell should be  
empty

Header 1		Header 3	Header 4
Data 1, 1	Data 1, 2	Data 1, 3	Data 1, 4
Data 2, 1	Data 2, 2		Data 2, 4

# Tables

```
<table>
  <tr>
    <th colspan="2">Header 1</th>
    <th>Header 3</th>
    <th>Header 4</th>
  </tr>
  <tr>
    <td>Data 1, 1</td>
    <td>Data 1, 2</td>
    <td rowspan="2">Data 1, 3</td>
    <td>Data 1, 4</td>
  </tr>
  <tr>
    <td>Data 2, 1</td>
    <td>Data 2, 2</td>
    <td>Data 2, 3</td>
    <td>Data 2, 4</td>
  </tr>
</table>
```

```
<style>
  table, th, td
  {
    border: 1px solid black;
    border-collapse: collapse;
  }
  th
  {
    width: 200px;
  }
  tr
  {
    height: 40px;
  }
</style>
```

Header 1	Header 3	Header 4
Data 1, 1	Data 1, 2	Data 1, 3
Data 2, 1	Data 2, 2	
		Data 2, 4

# Fieldset & Legend

- The `<fieldset>` tag is used to group related elements in a form.
- The `<fieldset>` tag draws a box around the related elements.
- The `<legend>` tag is used to define a caption for the `<fieldset>` element.

# Fieldset & Legend

```
<fieldset style="width: 300px">
    <legend><h3>Login</h3></legend>
    <p>
        User Name:<input type="text">
    </p>
    <p>
        Password:<input
type="password">
    </p>
</fieldset>
```

Login

User Name:

Password:

# Div Element

- The <div> tag defines a division or a section in an HTML document.
- The <div> tag is used as a container for HTML elements - which is then styled with CSS or manipulated with JavaScript.
- The <div> tag is easily styled by using the class or id attribute.
- Any sort of content can be put inside the <div> tag!
- By default, browsers always place a line break before and after the <div> element.

# Div Element

```
<div style="width: 300px;background-color: lightgray;">
  <h3>Login</h3>
  <p>
    User Name:<input type="text">
  </p>
  <p>
    Password:<input type="password">
  </p>
</div>
```

**Login**

User Name:

Password:

# Section Element

- The <section> HTML element represents a generic standalone section of a document, which doesn't have a more specific semantic element to represent it. Sections should always have a heading, with very few exception.

# section Element

```
<section id="1" style="background-color:orange; width: 300px;">
```

## WWF History

The World wide Fund for Nature (WWF) is an international organization working on issues regarding the conservation, research and restoration of the environment, formerly named the world wildlife Fund. WWF was founded in 1961.

```
</section>
```

```
<section id="2" style="background-color:aqua; width: 300px;">
```

## WWF's Symbol

The Panda has become the symbol of WWF. The well-known panda logo of WWF originated from a panda named Chi Chi that was transferred from the Beijing Zoo to the London Zoo in the same year of the establishment of

WWF.

```
</section>
```

## WWF History

The World Wide Fund for Nature (WWF) is an international organization working on issues regarding the conservation, research and restoration of the environment, formerly named the World Wildlife Fund. WWF was founded in 1961.

## WWF's Symbol

The Panda has become the symbol of WWF. The well-known panda logo of WWF originated from a panda named Chi Chi that was transferred from the Beijing Zoo to the London Zoo in the same year of the establishment of WWF.

# Header & Footer Elements

- The header element creates a header for this page that contains both text and graphics.
- The header element can be used multiple times on a page and can include HTML headings (<h1> through <h6>), navigation, images and logos and more.

## time Element

- The footer element describes a footer-content that usually appears at the bottom of the content or section element.

# figure Element and figcaption Element

- The **figure element** describes a figure (such as an image, chart or table) in the document so that it could be moved to the side of the page or to another page.
- The **figcaption element** provides a caption for the image in the figure element.

# figure Element and figcaption Element

```
<figure align="center">
    
    <figcaption>Fig.1 Flowchar for Sum of N
numbers</figcaption>
</figure>
```

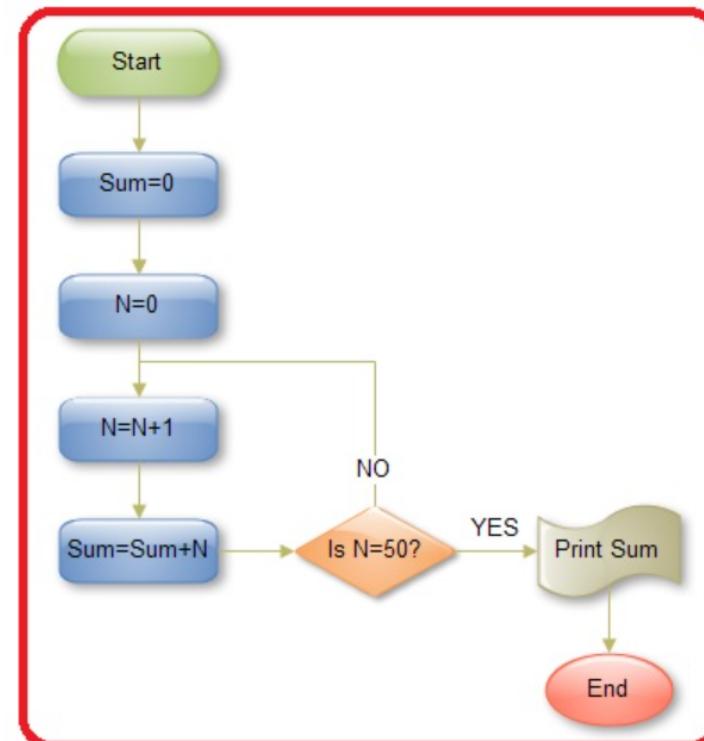


Fig.1 Flowchar for Sum of N numbers

# other Page Structure Elements

# Page-Structure Elements

- HTML5 introduces several new page-structure elements (Fig. 3.18) that meaningfully identify areas of the page as headers, footers, articles, navigation areas, asides, figures and more.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 3.18: sectionelements.html -->
4  <!-- New HTML5 section elements. -->
5  <html>
6      <head>
7          <meta charset="utf-8">
8          <title>New HTML5 Section Elements</title>
9      </head>
10
11     <body>
12         <header> <!-- header element creates a header for the page -->
13             <img src = "deitellogo.png" alt = "Deitel logo" />
14             <h1>Welcome to the Deitel Buzz Online<h1>
15
16             <!-- time element inserts a date and/or time -->
17             <time>2012-01-17</time>
18
19     </header>
20
21     <section id = "1"> <!-- Begin section 1 -->
22         <nav> <!-- nav element groups navigation links -->
23             <h2> Recent Publications</h2>
```

**Fig. 3.18** | New HTML5 section elements. (Part 1 of 13.)

```
24    <ul>
25        <li><a href = "http://www.deitel.com/books/iw3htp5">
26            Internet & World Wide Web How to Program, 5/e</a></li>
27        <li><a href = "http://www.deitel.com/books/androidfp/">
28            Android for Programmers: An App-Driven Approach</a>
29        </li>
30        <li><a href = "http://www.deitel.com/books/iphonefp">
31            iPhone for Programmers: An App-Driven Approach</a></li>
32        <li><a href = "http://www.deitel.com/books/jhttp9/">
33            Java How to Program, 9/e</a></li>
34        <li><a href = "http://www.deitel.com/books/cpphtp8/">
35            C++ How to Program, 8/e</a></li>
36        <li>
37            <a href = "http://www.deitel.com/books/vcsharp2010htp">
38                Visual C# 2010 How to Program, 4/e</a></li>
39        <li><a href = "http://www.deitel.com/books/vb2010htp">
40            Visual Basic 2010 How to Program</a></li>
41    </ul>
42    </nav>
43 </section>
44
45 <section id = "2"> <!-- Begin section 2 -->
46     <h2>How to Program Series Books</h2>
47     <h3><em>Java How to Program, 9/e</em></h3>
```

Fig. 3.18 | New HTML5 section elements. (Part 2 of 13.)

```
48
49      <figure> <!-- figure element describes the image -->
50          <img src = "jhtp.jpg" alt = "Java How to Program, 9/e" />
51
52          <!-- figurecaption element inserts a figure caption -->
53          <figcaption><em>Java How to Program, 9/e</em>
54              cover.</figcaption>
55      </figure>
56
57      <!--article element represents content from another source -->
58      <article>
59          <header>
60              <h5>From
61                  <em>
62                      <a href = "http://www.deitel.com/books/jhtp9/">
63                          Java How to program, 9/e: </a>
64                  </em>
65              </h5>
66          </header>
67
```

Fig. 3.18 | New HTML5 section elements. (Part 3 of 13.)

```
68 <p>Features include:  
69 <ul>  
70     <li>Rich coverage of fundamentals, including  
71         <!-- mark element highlights text -->  
72         <mark>two chapters on control statements.</mark></li>  
73     <li>Focus on <mark>real-world examples.</mark></li>  
74     <li><mark>Making a Difference exercises set.</mark></li>  
75     <li>Early introduction to classes, objects,  
76         methods and strings.</li>  
77     <li>Integrated exception handling.</li>  
78     <li>Files, streams and object serialization.</li>  
79     <li>Optional modular sections on language and  
80         library features of the new Java SE 7.</li>  
81     <li>Other topics include: Recursion, searching,  
82         sorting, generic collections, generics, data  
83         structures, applets, multimedia,  
84         multithreading, databases/JDBC&trade;, web-app  
85         development, web services and an optional  
86         ATM Object-Oriented Design case study.</li>  
87 </ul>  
88
```

**Fig. 3.18 |** New HTML5 section elements. (Part 4 of 13.)

```
89      <!-- summary element represents a summary for the -->
90      <!-- content of the details element -->
91      <details>
92          <summary>Recent Edition Testimonials</summary>
93          <ul>
94              <li>"Updated to reflect the state of the
95                  art in Java technologies; its deep and
96                  crystal clear explanations make it
97                  indispensable. The social-consciousness
98                  [Making a Difference] exercises are
99                  something really new and refreshing."
100             <strong>&mdash;Jos&acute;s Antonio
101                 Gonz&aacute;lez Seco, Parliament of
102                 Andalusia</strong></li>
103             <li>"Gives new programmers the benefit of the
104                 wisdom derived from many years of software
105                 development experience."<strong>
106                 &mdash;Edward F. Gehringer, North Carolina
107                 State University</strong></li>
108             <li>"Introduces good design practices and
109                 methodologies right from the beginning.
110                 An excellent starting point for developing
111                 high-quality robust Java applications."
112             <strong>&mdash;Simon Ritter,
113                 Oracle Corporation</strong></li>
```

Fig. 3.18 | New HTML5 section elements. (Part 5 of 13.)

```
114 <li>"An easy-to-read conversational style.  
115     Clear code examples propel readers to  
116     become proficient in Java."  
117     <strong>&mdash;Patty Kraft, San Diego State  
118     University</strong></li>  
119     <li>"A great textbook with a myriad of examples  
120     from various application domains&mdash;  
121     excellent for a typical CS1 or CS2 course."  
122     <strong>&mdash;William E. Duncan, Louisiana  
123     State University</strong></li>  
124             </ul>  
125         </details>  
126     </p>  
127 </article>  
128  
129     <!-- aside element represents content in a sidebar that's -->  
130     <!-- related to the content around the element -->  
131     <aside>  
132         The aside element is not formatted by the browsers.  
133     </aside>  
134
```

Fig. 3.18 | New HTML5 section elements. (Part 6 of 13.)

```
135      <h2>Deitel Developer Series Books</h2>
136      <h3><em>Android for Programmers: An App-Driven Approach
137          </em></h3>
138          Click <a href = "http://www.deitel.com/books/androidfp/">
139              here</a> for more information or to order this book.
140
141      <h2>LiveLessons Videos</h2>
142      <h3><em>C# 2010 Fundamentals LiveLessons</em></h3>
143          Click <a href = "http://www.deitel.com/Books/LiveLessons/">
144              here</a> for more information about our LiveLessons videos.
145      </section>
146
147      <section id = "3"> <!-- Begin section 3 -->
148          <h2>Results from our Facebook Survey</h2>
149          <p>If you were a nonprogrammer about to learn Java for the first
150              time, would you prefer a course that taught Java in the
151              context of Android app development? Here are the results from
152              our survey:</p>
153
154          <!-- meter element represents a scale within a range -->
155          0 <meter min = "0"
156              max = "54"
157              value = "14"></meter> 54
```

Fig. 3.18 | New HTML5 section elements. (Part 7 of 13.)

```
158      <p>Of the 54 responders, 14 (green) would prefer to
159          learn Java in the context of Android app development.</p>
160      </section>
161
162      <!-- footer element represents a footer to a section or page, -->
163      <!-- usually containing information such as author name, -->
164      <!-- copyright, etc. -->
165      <footer>
166          <!-- wbr element indicates the appropriate place to break a -->
167          <!-- word when the text wraps -->
168          <h6>&copy; 1992-2012 by Deitel &amp; Associates, Inc.
169          All Rights Reserved.<h6>
170          <!-- address element represents contact information for a -->
171          <!-- document or the nearest body element or article -->
172          <address>
173              Contact us at <a href = "mailto:deitel@deitel.com">
174                  deitel@deitel.com</a>
175          </address>
176      </footer>
177  </body>
178 </html>
```

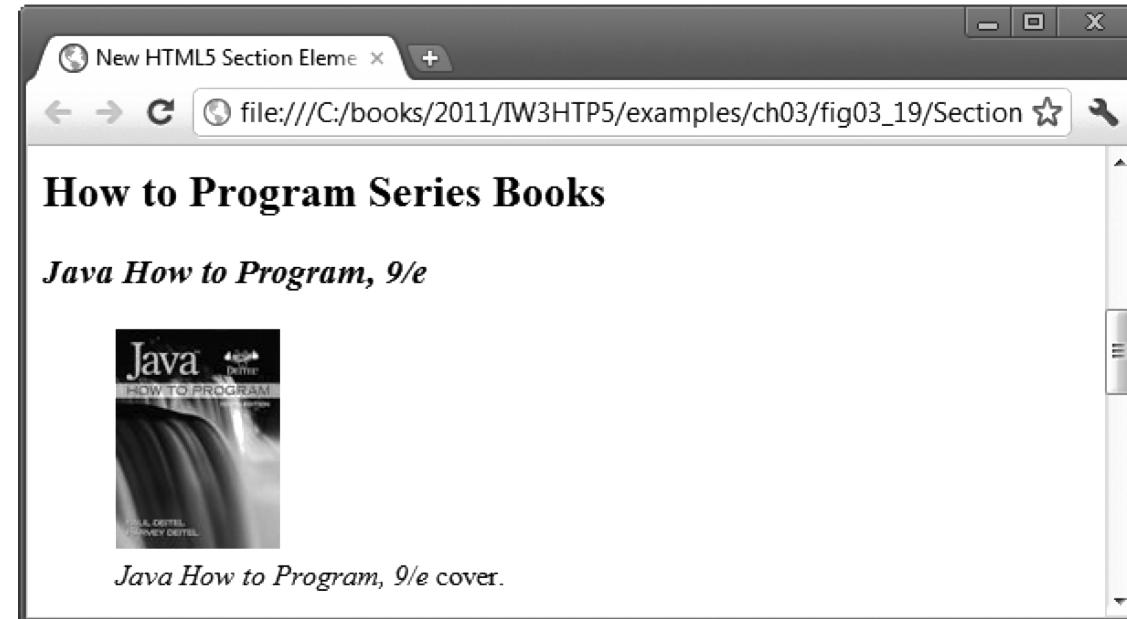
**Fig. 3.18** | New HTML5 section elements. (Part 8 of 13.)

a) Chrome browser showing the header element and a nav element that contains an unordered list of links



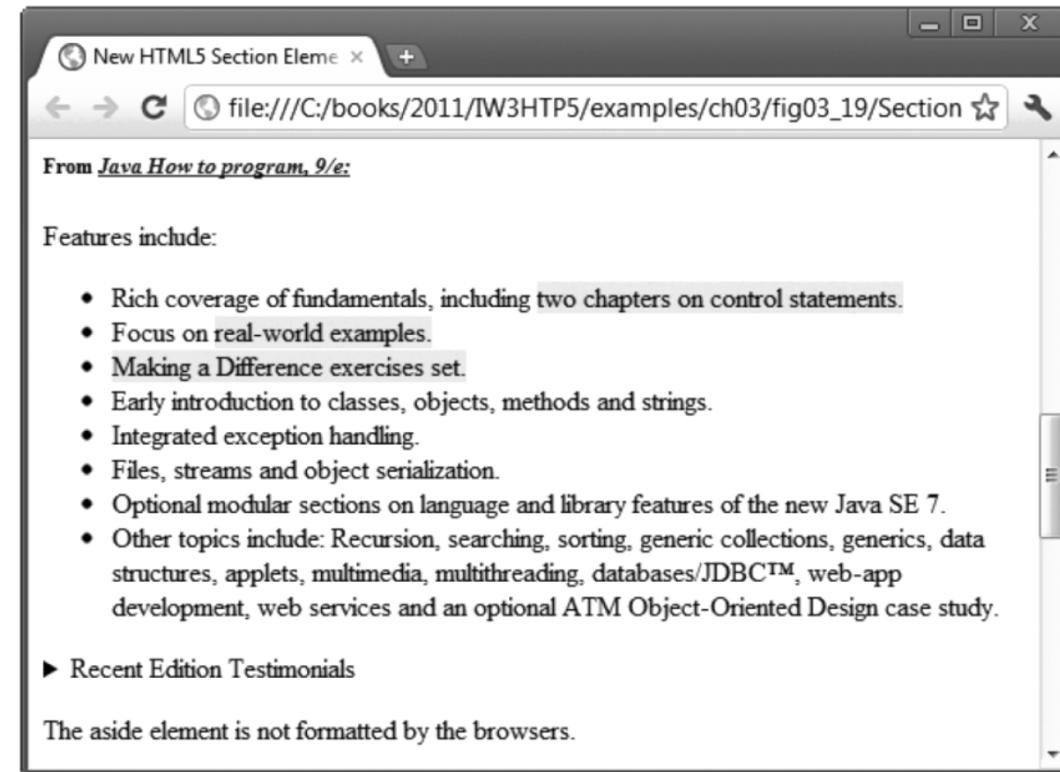
**Fig. 3.18** | New HTML5 section elements. (Part 9 of 13.)

b) Chrome browser showing the beginning of a section containing a figure and a figurecaption



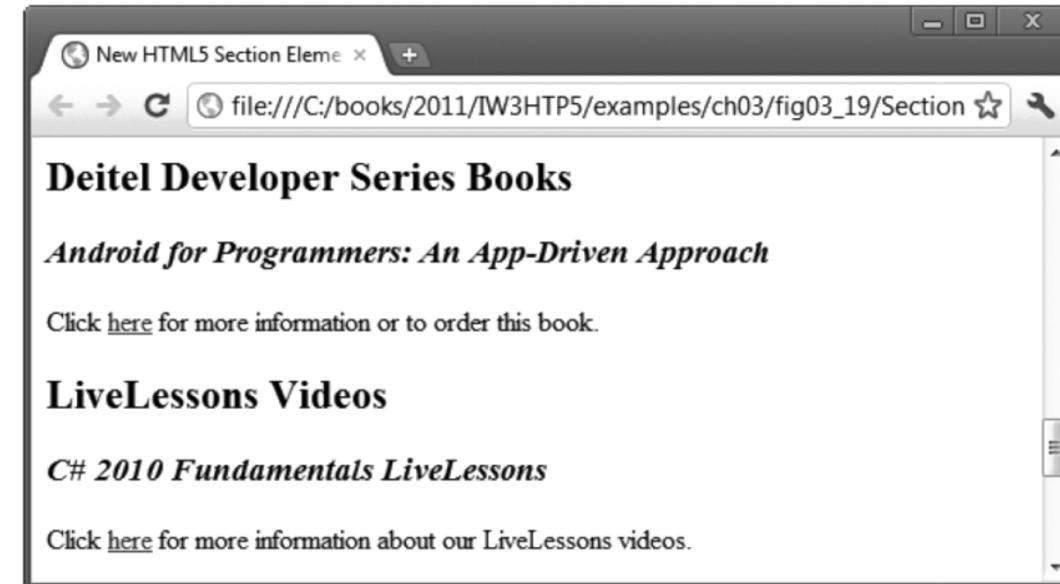
**Fig. 3.18 |** New HTML5 section elements. (Part 10 of 13.)

c) Chrome browser showing an **article** containing a **header**, some content and a collapsed **details** element, followed by an **aside** element



**Fig. 3.18 |** New HTML5 section elements. (Part 11 of 13.)

d) Chrome browser showing the end of the **section** that started in part (b)



**Fig. 3.18** | New HTML5 section elements. (Part 12 of 13.)

e) Chrome browser showing the last **section** containing a **meter** element, followed by a **footer** element



**Fig. 3.18 |** New HTML5 section elements. (Part 13 of 13.)

## nav Element

- The `nav element` groups navigation links.
- In this example, we used the heading Recent Publications and created a `ul` element with seven `li` elements that link to the corresponding web pages for each book.

## article Element

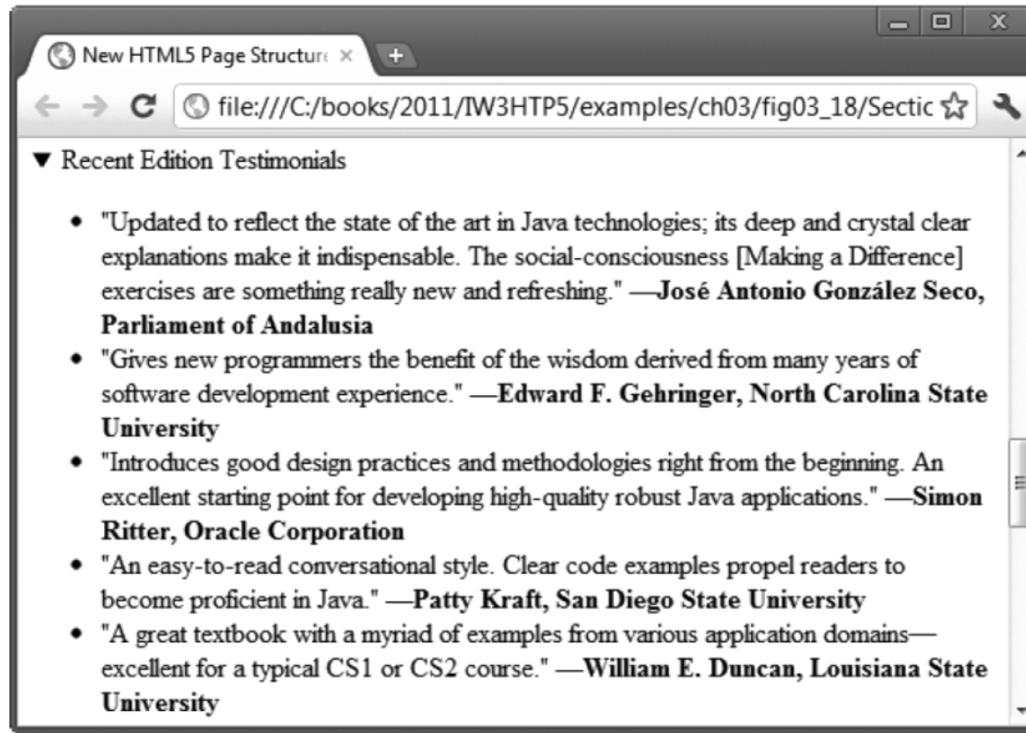
- The **article** element describes standalone content that could potentially be used or distributed elsewhere, such as a news article, forum post or blog entry.
- You can nest article elements. For example, you might have reader comments about a magazine nested as an article within the magazine article.

## summary Element and details Element

- The **summary element** displays a right-pointing arrow next to a summary or caption when the document is rendered in a browser (Fig. 3.19).
- When clicked, the arrow points downward and reveals the content in the **details element**.



**Fig. 3.19** | Demonstrating the summary and detail elements.  
(Part I of 2.)



**Fig. 3.19** | Demonstrating the summary and detail elements.  
(Part 2 of 2.)

## aside Element

- The `aside element` describes content that's related to the surrounding content (such as an article) but is somewhat separate from the flow of the text.
- For example, an aside in a news story might include some background history.

## **meter Element**

- The **meter element** renders a visual representation of a measure within a range (Fig. 3.20).
- In this example, we show the results of a recent web survey we did.
- The min attribute is "0" and a max attribute is "54" – indicating the total number of responses to our survey.
- The value attribute is "14", representing the total number of people who responded “yes” to our survey question.



**Fig. 3.20** | Chrome rendering the `meter` element.

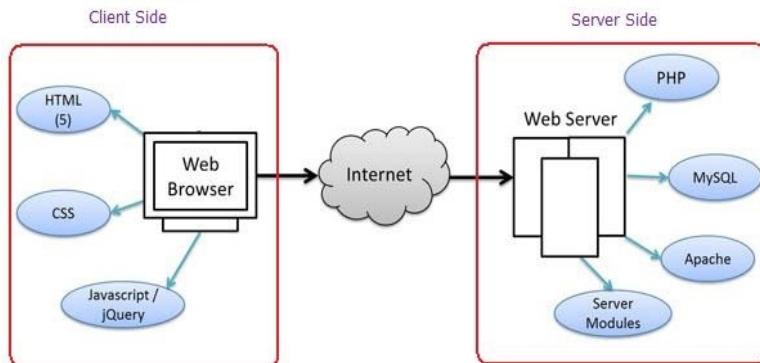
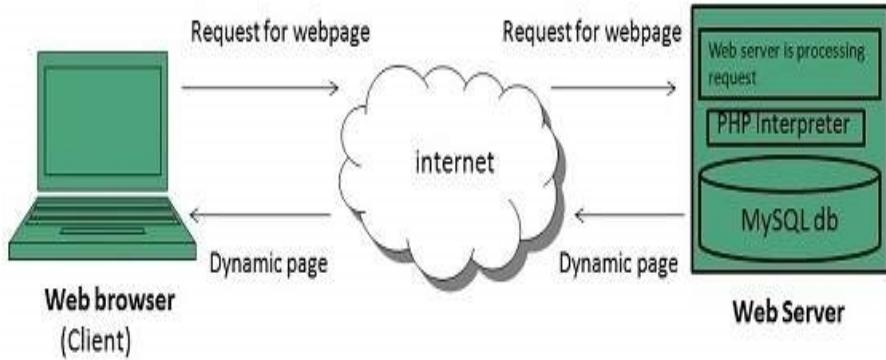
## Text-Level Semantics: mark Element and wbr Element

- The `mark element` highlights the text that's enclosed in the element.
- The `wbr element` indicates the appropriate place to break a word when the text wraps to multiple lines.
- You might use wbr to prevent a word from breaking in an awkward place.

# JavaScript

Java Script

# Web Technology



- Various tools and techniques that are utilized in the process of communication between different types of devices over the internet
- **Web Development can be classified into two ways**
  - Frontend Development
    - Part of a website that the user interacts
    - Eg: HTML, CSS, JavaScript
  - Backend Development
    - Backend is the server side of a website that users cannot see and interact
    - Eg. PHP, Java, Python, C#

# Scripting Language

- Programming languages, a language where instructions are written for a runtime environment and do not require the compilation step and are rather interpreted

Scripting Language	Programming Language
Based on the Interpreter	Based on the compiler
Runs inside the program and is dependent on it	Independent of a parent program
Does not require compiling the file and running directly	Requires to compile the file first
Users can easily write and use it	Difficult to use and write
Requires low maintenance	Requires high maintenance
Examples of scripting languages include VB Script, JavaScript, Perl, Ruby, and PHP.	Examples of programming languages include COBOL, Java, VB, Basic, C, C++, C#, and Pascal.

# JavaScript - Introduction

- ø Lightweight programming language ("scripting language")
  - used to make web pages interactive
  - insert dynamic text into HTML (ex: user name)
  - **react to events** (ex: page load user click)
  - get information about a user's computer (ex: browser type)
  - perform calculations on user's computer (ex: form validation)
  - saves server traffic
- ø Are JavaScript and HTML the same?
  - Javascript is an advanced programming language that makes web pages more interactive and dynamic whereas HTML is a standard markup language that provides the primary structure of a website
- ø Today, JavaScript can execute not only in the browser, but also on the server, or actually on any device that has a special program called the JavaScript engine.
- ø Browser has an embedded engine sometimes called a “JavaScript virtual machine”.

# JavaScript Vs Java

- ø NOT related to Java other than by name and some syntactic similarities
- ø interpreted, not compiled
- ø more relaxed syntax and rules
  - ø fewer and "looser" data types
  - ø variables don't need to be declared
  - ø errors often silent (few exceptions)
- ø contained within a web page and integrates with its HTML/CSS content

## Rules for writing the JavaScript code

- Script should be placed inside the <script> tag
- Semicolon at the end of each statement is optional
- Use 'document.write' for writing a string into HTML document.
- JavaScript is case sensitive
- You can insert special characters with backslash (\& or \\$).

### Simple JavaScript Program

```
<html>
    <body>
        <script
        type="text/javascript">
            document.write("Career
Ride Info");
        </script>
    </body>
</html>
```

# Adding JavaScript to HTML

## Three Ways

### 1. Simple JavaScript Program using <HEAD> tag

```
<html>
  <head>
    <script type = "text/javascript">
      function abc()
      {
        document.write("CareerRide
Info");
      }
    </script>
  </head>
  <body>
    click the button
    <input type=button
      onclick="abc()" value="click">
  </body>
</html>
```

### 2. Simple JavaScript Program using <BODY> tag

```
<html>
  <body>
    <script type="text/javascript">
      document.write("CareerRide
Info");
    </script>
  </body>
</html>
```

### 3. Simple JavaScript Program using External File

```
<html>
  <body>
    <script type="text/javascript"
src="abc.js">
    </script>
  </body>
</html>
```

--  
abc.js //External File Name

```
document.write("CareerRide Info");
```

# JavaScript Programming Constructs

## Java Script Basics

- Comment
- Variable
- Data Types
- Operators
- If Statements
- Switch
- Loop
- Function

## Java Script Objects

- Object
- Array
- String
- Date
- Math
- Number
- Boolean

# JavaScript Comments

There are two types of comments in JavaScript.

- 1.Single-line Comment
- 2.Multi-line Comment

```
<script>  
// It is single line comment  
document.write("hello javascript");  
</script>
```

```
<script>  
/* It is multi line comment.  
It will not be displayed */  
document.write("example of javascript multiline comment");  
</script>
```

# JavaScript Data Types

- Ø Provides different data types to hold different types of values
- Ø Two types of data types in JavaScript.
  - Ø Primitive data type
  - Ø Non-primitive (reference) data type
- Ø JavaScript is a dynamic type language
  - Ø don't need to specify type of the variable because it is dynamically used by JavaScript engine
  - Ø need to use var here to specify the data type

## Example: Loosely-typed JavaScript      Primitive and Non Primitive data types

var myVar = 100;	String	Object
myVar = true;	Number	Array
myVar = null;	Boolean	RegExp
myVar = undefined;	Undefined	
myVar = "Steve";		
alert(myvar); // stev	Null	

# Variables

- Ø Storage location
- Ø Two types of variables in JavaScript : local variable and global variable.
- Ø Rules while declaring a JavaScript variable (also known as identifiers).
  - Name must start with a letter (a to z or A to Z), underscore( \_ ), or dollar ( \$ ) sign
  - After first letter we can use digits (0 to 9), for example value1
  - Case sensitive, for example x and X are different variables
- Ø JavaScript global variable is declared **outside the function or declared with window object**

## Example : Simple Program on Local variable

```
<html>
  <head>
    <script type="text/javascript">
      function funccount(a) // Function with Argument
      {
        var count=5; // Local variable
        count+=2;
        document.write("<b>Inside Count: </b>" + count + "<br>");
        a+=3;
        document.write("<b>Inside A: </b>" + a + "<br>");
      }
    </script>
  </head>
  <body>
    <script type="text/javascript">
      var a=3, count = 0;
      funccount(a);
      document.write("<b>Outside Count: </b>" + count + "<br>");
      document.write("<b>Outside A: </b>" + a + "<br>");
    </script>
  </body>
</html>
```

variable.htm  
1

### Output:

**Inside  
Count: 7  
Inside A: 6  
Outside  
Count: 0  
Outside  
A: 3**

## Example : Simple Program on Global variable

```
<html>
  <head>
    <script type = "text/javascript">
      count = 5;          //Global variable
      var a = 4;          //Global variable
      function funccount() // Function Declaration
      {
        count+=5;         // Local variable
        a+=4;
        document.write("<b>Inside function Global Count: </b>" + count + "<br>");
        document.write("<b>Inside function Global a: </b>" + a + "<br>");
      }
    </script>
  </head>
  <body>
    <script type="text/javascript">
      document.write("<b>Outside function Global Count: </b>" + count + "<br>");
      document.write("<b>Outside function Global a: </b>" + a + "<br>");
      funccount();
    </script>
  </body>
</html>
```

Output:

Outside function Global  
Count: 5  
Outside function Global  
A:</b>10<br>  
Inside function Global  
Count: 10  
Inside function Global  
a: 8

# JavaScript Operators

## Arithmetic Operators

Operator	Description	Example
+	Addition	$10+20 = 30$
-	Subtraction	$20-10 = 10$
*	Multiplication	$10*20 = 200$
/	Division	$20/10 = 2$
%	Modulus	$20\%10 = 0$
++	Increment	<code>var a=10; a++; Now a = 11</code>
--	Decrement	<code>var a=10; a--; Now a = 9</code>

## Relational Operators

Operator	Description	Example
==	Is equal to	$10==20 = \text{false}$
===	Identical (equal and of same type)	$10==20 = \text{false}$
!=	Not equal to	$10!=20 = \text{true}$
!==	Not Identical	$20!==20 = \text{false}$
>	Greater than	$20>10 = \text{true}$
>=	Greater than or equal to	$20>=10 = \text{true}$
<	Less than	$20<10 = \text{false}$
<=	Less than or equal to	$20<=10 = \text{false}$

## Bitwise Operators

Operator	Description	Example
&	Bitwise AND	$(10==20 \& 20==33) = \text{false}$
	Bitwise OR	$(10==20   20==33) = \text{false}$
^	Bitwise XOR	$(10==20 ^ 20==33) = \text{false}$
~	Bitwise NOT	$(\sim 10) = -10$
<<	Bitwise Left Shift	$(10<<2) = 40$
>>	Bitwise Right Shift	$(10>>2) = 2$
>>>	Bitwise Right Shift with Zero	$(10>>>2) = 2$

# JavaScript Operators

## Assignment Operators

Operator	Description	Example
=	Assign	$10+10 = 20$
+=	Add and assign	<code>var a=10; a+=20; Now a = 30</code>
-=	Subtract and assign	<code>var a=20; a-=10; Now a = 10</code>
*=	Multiply and assign	<code>var a=10; a*=20; Now a = 200</code>
/=	Divide and assign	<code>var a=10; a/=2; Now a = 5</code>
%=	Modulus and assign	<code>var a=10; a%=-2; Now a = 0</code>

## Logical Operators

Operator	Description	Example
&&	Logical AND	<code>(10==20 &amp;&amp; 20==33) = false</code>
	Logical OR	<code>(10==20    20==33) = true</code>
!	Logical Not	<code>!(10==20) = true</code>

## Special Operators

Operator	Description
(?:)	Conditional Operator returns value based on the condition. It is like if-else.
,	Comma Operator allows multiple expressions to be evaluated as single statement.
delete	Delete Operator deletes a property from the object.
in	In Operator checks if object has the given property
instanceof	checks if the object is an instance of given type
new	creates an instance (object)
typeof	checks the type of object.
void	it discards the expression's return value.
yield	checks what is returned in a generator by the generator's iterator.

# Arithmetic Operators - Example

```
<html>
<body>
<script type="text/javascript">
    document.write("a + b + c = ");
    result = a + b + c;
    document.write(result);
    document.write(linebreak);

    var a = 33;
    var b = 10;
    var c = "Test";
    var linebreak = "<br />";

    document.write("a + b = ");
    result = a + b;
    document.write(result);
    document.write(linebreak);

    document.write("a - b = ");
    result = a - b;
    document.write(result);
    document.write(linebreak);

    document.write("a / b = ");
    result = a / b;
    document.write(result);
    document.write(linebreak); Set the variables to different values
    document.write("a % b = "); Set the variables to different values and
    result = a % b;
    document.write(result);
    document.write(linebreak);
</script>
</body>
</html>
```

arith.html

Output  
a + b = 43  
a - b = 23  
a / b = 3.3  
a % b = 3  
a + b + c = 43Test  
++a = 35  
--b = 8

Set the variables to different values  
and then try...

# Bitwise Operators - Example

```
<html>
  <body>

    <script type="text/javascript">
      var a = 2; // Bit presentation
      var b = 3; // Bit presentation
      var linebreak = "<br />";

      document.write("(a & b) => ");
      result = (a & b);
      document.write(result);
      document.write(linebreak);

      document.write("(a | b) => ");
      result = (a | b);  </script>
      document.write(result);
      document.write(linebreak);<p>Set the variables to different
                                values and different operators and then
      document.write("(a ^ b) => ");try...</p>
      result = (a ^ b);      </body>
      document.write(result);</html>
      document.write(linebreak); Java Script
```

document.write("(~b) => ");
result = (~b);
document.write(result);
document.write(linebreak);

document.write("(a << b) => "
result = (a << b); Output
(a & b) => 2
document.write(result); (a | b) => 3
document.write(linebreak); (a ^ b) => 1
(~b) => -4
(a << b) => 16
document.write("(a >> b) => "
result = (a >> b); (ab) => 0
Set the variables to different values
and different operators and then
document.write(result);try...
document.write(linebreak);

# JavaScript If Statements

JavaScript if-else statement is used *to execute the code whether condition is true or false.* There are three forms of if statement in JavaScript

- 1.If Statement
- 2.If else statement
- 3.if else if statement

# JavaScript Switch Statement

## The switch statement

```
switch (//Some expression) {  
    case 'option1':  
        // Do something  
        break;  
    case 'option2':  
        // Do something else  
        break;  
    default:  
        // Do yet another thing  
        break;  
}
```

# JavaScript Loop Statements

Four types of loops in JavaScript

1. for loop
2. while loop
3. do-while loop
4. for-in loop

# JavaScript Functions

Mainly two advantages of JavaScript functions

1. **Code reusability:** We can call a function several times so it save coding.

2. **Less coding:** It makes our program compact. We don't need to write many

lines of code each time ~~to perform a common task~~  
**function functionName([arg1, arg2, ...argN])**  
  {   //code to be executed   }

Example: Define and call a Function

```
function  
  ShowMessage()  
{  
    alert("Hello  
  world!");  
}  
ShowMessage();
```

```
<html>  
<body>  
<script>  
function msg(){  
alert("hello! this is message");  
}  
</script>  
<input type="button" onclick="msg()"  
value="call function"/>  
</body>  
</html>
```

# JavaScript Functions

## Function Parameters

- A function can have one or more parameters, which will be supplied by the calling code and can be used inside a function. JavaScript is a dynamic type scripting language, so a function parameter can have value of any data type.

Example:

```
function  
  ShowMessage(firstName,  
  lastName) {  
    alert("Hello " + firstName  
    + " " + lastName);  
}
```

```
ShowMessage("Steve", "Jobs");  
ShowMessage("Bill", "Gates");  
ShowMessage(100, 200);
```

## The Arguments Object

- All the functions in JavaScript can use arguments object by default. An arguments object includes value of each parameter.
- The arguments object is an array like object. You can access its values using index similar to array. However, it does not support array methods.

Example:

```
function ShowMessage(firstName,  
  lastName) {  
    alert("Hello " +  
    arguments[0] + " " +  
    arguments[1]);
```

```
ShowMessage("Steve", "Jobs");  
ShowMessage("Bill", "Gates");  
ShowMessage(100, 200);
```

Java Script }

176

# JavaScript Functions

## Function Expression

- JavaScript allows us to assign a function to a variable and then use that variable as a function. It is called function expression.

### Example

```
var add = function sum(val1,  
    val2) {  
    return val1 + val2;  
};  
  
var result1 = add(10,20);  
var result2 = sum(10,20); //  
not valid
```

## Anonymous Function

- JavaScript allows us to define a function without any name. This unnamed function is called anonymous function. Anonymous function must be assigned to a variable.

### Example:

```
var showMessage = function  
(){  
    alert("Hello world!");  
};  
showMessage();  
var sayHello = function  
(firstName) {  
    alert("Hello " +  
        firstName);  
};  
showMessage();  
sayHello("Bill");
```

# JavaScript Functions

## Nested Functions

- In JavaScript, a function can have one or more inner functions. These nested functions are in the scope of outer function. Inner function can access variables and parameters of outer function. However, outer function cannot access variables defined inside inner functions.

### Example:

```
function  
    ShowMessage(firstName)  
{  
    function SayHello() {  
        alert("Hello " +  
    firstName);  
    }  
    return SayHello();  
}  
ShowMessage("Steve");
```

## Points to Remember

- JavaScript a function allows you to define a block of code, give it a name and then execute it as many times as you want.
- A function can be defined using function keyword and can be executed using () operator.
- A function can include one or more parameters. It is optional to specify function parameter values while executing it.
- JavaScript is a loosely-typed language. A function parameter can hold value of any data type.
- You can specify less or more arguments while calling function.
- All the functions can access arguments object by default instead of parameter names.
- A function can return a literal value or another function.
- A function can be assigned to a variable with different name.
- JavaScript allows you to create anonymous functions that must be assigned to a variable.

# JavaScript Functions

## Points to Remember

- JavaScript a function allows you to define a block of code, give it a name and then execute it as many times as you want.
- A function can be defined using function keyword and can be executed using () operator.
- A function can include one or more parameters. It is optional to specify function parameter values while executing it.
- JavaScript is a loosely-typed language. A function parameter can hold value of any data type.
- You can specify less or more arguments while calling function.
- All the functions can access arguments object by default instead of parameter names.
- A function can return a literal value or another function.
- A function can be assigned to a variable with different name.
- JavaScript allows you to create anonymous functions that must be assigned to a variable.

# JavaScript Objects

- JavaScript is an Object Oriented Programming (OOP) language. A programming language can be called object-oriented if it provides four basic capabilities to developers –
- **Encapsulation** – the capability to store related information, whether data or methods, together in an object.
- **Aggregation** – the capability to store one object inside another object.
- **Inheritance** – the capability of a class to rely upon another class (or number of classes) for some of its properties and methods.
- **Polymorphism** – the capability to write one function or method that works in a variety of different ways.
- Objects are composed of attributes. If an attribute contains a function, it is considered to be a method of the object, otherwise the attribute is considered a property.

# JavaScript Objects

## Object Properties

- Object properties can be any of the three primitive data types, or any of the abstract data types, such as another object.
- Object properties are usually variables that are used internally in the object's methods, but can also be globally visible variables that are used throughout the page.

syntax for adding a property to an object is:

objectName.objectProperty =  
propertyValue;

## Object Methods

- Methods are the functions that let the object do something or let something be done to it. There is a small difference between a function and a method - at a function is a standalone unit of statements and a method is attached to an object and can be referenced by the `this` keyword.
- Methods are useful for everything from displaying the contents of the object to the screen to performing complex mathematical operations on a group of local properties and parameters.
- **For example** – Following is a simple example to show how to use the `write()` method of document object to write any content on the document.

`document.write("This is test");`

# JavaScript Objects

## User-Defined Objects:

- All user-defined objects and built-in objects are descendants of an object called **Object**.

## The new Operator:

- The **new** operator is used to create an instance of an object. To create an object, the **new** operator is followed by the constructor method.
- In the following example, the constructor methods are **Object()**, **Array()**, and **Date()**. These constructors are built-in JavaScript functions.

```
var employee = new Object();
var books = new Array("C++", "Perl", "Java");
var day = new Date("August 15, 1947");
```

# JavaScript Objects

Example : how to create an object -Java Script:

```
<html>
  <head>
    <title>User-defined objects</title>

    <script type="text/javascript">
      var book = new Object(); // Create the object
      book.subject = "Perl"; // Assign properties
object
      book.author = "Mohtashim";
    </script>

  </head>
  <body>
    <script type="text/javascript">
      document.write("Book name is : " + book.subject +
"<br>");
      document.write("Book author is : " + book.author +
"<br>");
    </script>
  </body>
</html>
```

Output:

Book name is : Perl  
Book author is :  
Mohtashim

## JavaScript Objects

Create an object with a User-Defined Function. Here this keyword is used to refer to the object that has been passed to a function.

```
<html>
  <head>
    <title>User-defined objects</title>
    <script type="text/javascript">
      function book(title, author){
        this.title = title;
        this.author = author;
      }
    </script>

  </head>
  <body>
    <script type="text/javascript">
      var myBook = new book("Perl", "Mohtashim");
      document.write("Book title is : " + myBook.title + "<br>");
      document.write("Book author is : " + myBook.author + "<br>");
    </script>

  </body>
</html>
```

Output:

Book name is : Perl  
Book author is :  
Mohtashim

# JavaScript Objects

## Defining Methods for an Object

```
<html>
  <head>
    <title>User-defined objects</title>
  </head>
  <body>

    <script type="text/javascript">
      // Define a function which will work as a method
      function addPrice(amount){
        this.price = amount;
      }

      function book(title, author){
        this.title = title;
        this.author = author;
        this.addPrice = addPrice;
      }
      // Assign that method as property.
      book.prototype.addPrice = addPrice;
    </script>
  </body>
</html>
```

```
<script type="text/javascript">
  var myBook = new book("Perl",
  "Mohtashim");
  myBook.addPrice(100);

  document.write("Book title is : " +
myBook.title + "<br>");
  document.write("Book author is : " +
myBook.author + "<br>");
  document.write("Book price is : " +
myBook.price + "<br>");

</script>
```

**Output:**  
Book name is : Perl  
Book author is : Mohtashim  
Book price is : 100

# JavaScript Objects

## JavaScript Native Objects

- JavaScript has several built-in or native objects. These objects are accessible anywhere in your program and will work the same way in any browser running in any operating system.
- Here is the list of all important Javascript Native Objects –
  - [JavaScript Number Object](#)
  - [JavaScript Boolean Object](#)
  - [JavaScript String Object](#)
  - [JavaScript Array Object](#)
  - [JavaScript Date Object](#)
  - [JavaScript Math Object](#)
  - [JavaScript RegExp Object](#)

\*

# JavaScript String Object

- The **String** object lets you work with a series of characters; it wraps Javascript's string primitive data type with a number of helper methods.

Syntax: var val = new String(string);

## String Properties

- **Constructor** : Returns a reference to the String function that created the object.
- **Length** : Returns the length of the string.

## String Methods

- **charAt()** : Returns the character at the specified index.
- **charCodeAt()** : Returns a number indicating the Unicode value of the character at the given index.
- **concat()** : Combines the text of two strings and returns a new string.
- **indexOf()** : Returns the index within the calling string object of the first occurrence of the specified value, or -1 if not found.
- **lastIndexOf()** : Returns the index within the calling string object of the last occurrence of the specified value, or -1 if not found.

# JavaScript String Object

## String Methods

- localeCompare() : Returns a number indicating whether a reference string comes before or after or is the same as the given string in sort order.
- match() : used to match a regular expression against a string.
- replace() : Used to find a match between a regular expression and a string, and to replace the matched substring with a new substring.
- search() : Executes the search for a match between a regular expression and a specified string.
- slice() : Extracts a section of a string and returns a new string.
- split() : Splits a String object into an array of strings by separating the string into substrings.
- substr() : Returns the characters in a string beginning at the specified location through the specified number of characters.
- substring() : Returns the characters in a string between two indexes into the string.
- toLocaleLowerCase() : The characters within a string are converted to lower case while respecting the current locale.
- toLocaleUpperCase() : The characters within a string are converted to upper case while respecting the current locale.
- toLowerCase() : Returns the calling string value converted to lower case.
- toString() : Returns a string representing the specified object.
- toUpperCase() : Returns the calling string value converted to uppercase.
- valueOf() : Returns the primitive value of the specified object.

# JavaScript Array Object

- The **Array** object lets you store multiple values in a single variable. It stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Syntax :

```
var fruits = new Array( "apple", "orange", "mango" );
```

```
var fruits = [ "apple", "orange", "mango" ];
```

fruits[0] is the first element

fruits[1] is the second element

fruits[2] is the third element

\*

# JavaScript Array Object

## Array Properties

- Constructor : Returns a reference to the array function that created the object.
- index : The property represents the zero-based index of the match in the string
- Input : This property is only present in arrays created by regular expression matches.
- length : Reflects the number of elements in an array.
- Prototype : The prototype property allows you to add properties and methods to an object.

## Array Methods

- concat() : Returns a new array comprised of this array joined with other array(s) and/or value(s).
- every() : Returns true if every element in this array satisfies the provided testing function.
- filter() : Creates a new array with all of the elements of this array for which the provided filtering function returns true.
- forEach() : Calls a function for each element in the array.
- indexOf()\* : Returns the first (0-based) index of an element within<sup>190</sup> the array equal to the specified value, or -1 if none is found.
- join() : Joins all elements of an array into a string.

# JavaScript Array Object

## Array Methods

- lastIndexOf() : Returns the last (greatest) index of an element within the array equal to the specified value, or -1 if none is found.
- map() : Creates a new array with the results of calling a provided function on every element in this array.
- pop() : Removes the last element from an array and returns that element.
- push() : Adds one or more elements to the end of an array and returns the new length of the array.
- reduce() : Apply a function simultaneously against two values of the array (from left-to-right) as to reduce it to a single value.
- reduceRight() : Apply a function simultaneously against two values of the array (from right-to-left) as to reduce it to a single value.
- reverse() : Reverses the order of the elements of an array -- the first becomes the last, and the last becomes the first.
- shift() : Removes the first element from an array and returns that element.
- slice() : Extracts a section of an array and returns a new array.
- some() : Returns true if at least one element in this array satisfies the provided testing function.
- toSource() : Represents the source code of an object
- sort() : Sorts the elements of an array
- splice() : Adds and/or removes elements from an array.
- toString() : Returns a string representing the array and its elements.
- unshift() : Adds one or more elements to the front of an array and returns the new length of the array.

# JavaScript Date Object

- The Date object is a datatype built into the JavaScript language. Date objects are created with the `new Date( )` as shown below.
- Once a Date object is created, a number of methods allow you to operate on it. Most methods simply allow you to get and set the year, month, day, hour, minute, second, and millisecond fields of the object, using either local time or UTC (universal, or GMT) time.

## Syntax

```
new Date( )
```

```
new Date(milliseconds)
```

```
new Date(datestring)
```

```
new Date(year,month,date[,hour,minute,second,millisecond])
```

Brackets are always optional.

\*

# JavaScript Date Object

## Date Properties

- Constructor : Specifies the function that creates an object's prototype.
- prototype : The prototype property allows you to add properties and methods to an object

## Date Methods

- date() : Returns today's date and time
- getDate() : Returns the day of the month for the specified date according to local time.
- getDay() : Returns the day of the week for the specified date according to local time.
- getFullYear() : Returns the year of the specified date according to local time.
- getHours() : Returns the hour in the specified date according to local time.
- getMilliseconds() : Returns the milliseconds in the specified date according to local time.
- getMinutes() : Returns the minutes in the specified date according to local time.
- getMonth() : Returns the month in the specified date according to local time.
- getSeconds() : Returns the seconds in the specified date according to local time.
- getTime() : Returns the numeric value of the specified date as the number of milliseconds since January 1, 1970, 00:00:00 UTC.
- getTimezoneOffset() : Returns the time-zone offset in minutes for the current locale.
- getUTCDate() : Returns the day (date) of the month in the specified date according to universal time

\*

# JavaScript Math Object

- The `math` object provides you properties and methods for mathematical constants and functions. Unlike other global objects, Math is not a constructor. All the properties and methods of `Math` are static and can be called by using `Math` as an object without creating it.
- Thus, you refer to the constant `pi` as `Math.PI` and you call the `sine` function as `Math.sin(x)`, where `x` is the method's argument.

## Syntax

```
var pi_val = Math.PI;  
var sine_val = Math.sin(30);
```

\*

# JavaScript Math Object

## Math Properties

- E : Euler's constant and the base of natural logarithms, approximately 2.718.
- LN2 : Natural logarithm of 2, approximately 0.693.
- LN10 : Natural logarithm of 10, approximately 2.302.
- LOG2E : Base 2 logarithm of E, approximately 1.442.
- LOG10E : Base 10 logarithm of E, approximately 0.434.
- PI : Ratio of the circumference of a circle to its diameter, approximately 3.14159.
- SQRT1\_2 : Square root of 1/2; equivalently, 1 over the square root of 2, approximately 0.707.
- SQRT2 : Square root of 2, approximately 1.414.

\*

# JavaScript Math Object

## Math Methods

- abs() : Returns the absolute value of a number.
- acos() : Returns the arccosine (in radians) of a number.
- asin() : Returns the arcsine (in radians) of a number.
- atan() : Returns the arctangent (in radians) of a number.
- atan2() : Returns the arctangent of the quotient of its arguments.
- ceil() : Returns the smallest integer greater than or equal to a number.
- cos() : Returns the cosine of a number.
- exp() : Returns  $E^N$ , where N is the argument, and E is Euler's constant, the base of the natural logarithm.
- floor() : Returns the largest integer less than or equal to a number.
- log() : Returns the natural logarithm (base E) of a number.
- max() : Returns the largest of zero or more numbers.
- min() : Returns the smallest of zero or more numbers.
- pow() : Returns base to the exponent power, that is, base exponent.
- random() : Returns a pseudo-random number between 0 and 1.
- round() : Returns the value of a number rounded to the nearest integer.
- sin() : Returns the sine of a number.
- sqrt() : Returns the square root of a number.
- tan() : Returns the tangent of a number.
- toSource() : Returns the string "Math".

\*

## JavaScript Number Object

- The **Number** object represents numerical date, either integers or floating-point numbers. In general, you do not need to worry about **Number** objects because the browser automatically converts number literals to instances of the number class.

```
var val = new Number(number);
```

- In the place of number, if you provide any non-number argument, then the argument cannot be converted into a number, it returns NaN (Not-a-Number).

# JavaScript Number Object

## Number Properties

- MAX\_VALUE : The largest possible value a number in JavaScript can have  
1.7976931348623157E+308
- MIN\_VALUE : The smallest possible value a number in JavaScript can have 5E-324
- NaN : Equal to a value that is not a number.
- NEGATIVE\_INFINITY : value that is less than MIN\_VALUE.
- POSITIVE\_INFINITY : value that is greater than MAX\_VALUE
- prototype : static property of the Number object. Use the prototype property to assign new properties and methods to the Number object in the current document
- Constructor : Returns the function that created this object's instance. By default this is the Number object

# JavaScript Number Object

## Number Methods

- `toExponential()` : Forces a number to display in exponential notation, even if the number is in the range in which JavaScript normally uses standard notation.
- `toFixed()` : Formats a number with a specific number of digits to the right of the decimal.
- `toLocalestring()` : Returns a string value version of the current number in a format that may vary according to a browser's local settings.
- `toPrecision()` : Defines how many total digits (including digits to the left and right of the decimal) to display of a number.
- `toString()` : Returns the string representation of the number's value.
- `valueOf()` : Returns the number's value.

\*

# JavaScript Boolean Object

- The Boolean object represents two values, either "true" or "false". 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.

Syntax ; var val = new Boolean(value);

## Boolean Properties

- constructor : Returns a reference to the Boolean function that created the object.
- prototype : The prototype property allows you to add properties and methods to an object.

## Boolean Methods

- toSource() : Returns a string containing the source of the Boolean object; you can use this string to create an equivalent object.
- 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.

# JavaScript RegExp Object

- A regular expression is an object that describes a pattern of characters.
- The JavaScript **RegExp** class represents regular expressions, and both **String** and **RegExp** define methods that use regular expressions to perform powerful pattern-matching and search-and-replace functions on text.

## Syntax

```
var pattern = new RegExp(pattern, attributes);
```

or simply

```
var pattern = /pattern/attributes;
```

\*

## References

- Jeffrey C. Jackson, *WEB TECHNOLOGIES A Computer Science Perspective*, Pearson Prentice Hall, Pearson Education, 2007
- <https://www.javatpoint.com/javascript-tutorial>

# THANK YOU

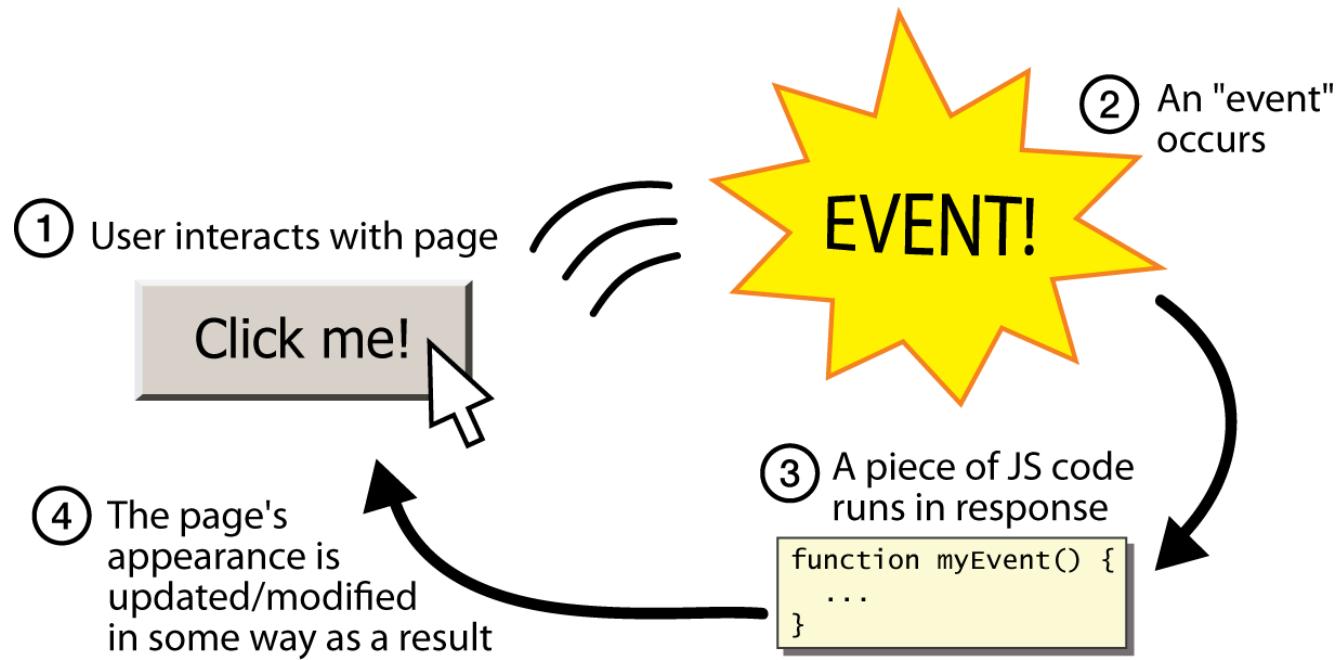
# JavaScript

Java Script

- **JavaScript Events**
- **BOM objects**
- **DOM objects**
- **JavaScript**

**validation**

# Event Driven programming



### what is an Event ?

- JavaScript's interaction with HTML is handled through events that occur when the user or the browser manipulates a page.
- when the page loads, it is called an event. when the user clicks a button, that click too is an event. other examples include events like pressing any key, closing a window, resizing a window, etc.
- Developers can use these events to execute JavaScript coded responses, which cause buttons to close windows, messages to be displayed to users, data to be validated, and virtually any other type of response imaginable.

# JavaScript - Events

## Mouse Event

Event Performed	Event Handler	Description
click	onclick	when mouse click on an element
mouseover	onmouseover	when the cursor of the mouse comes over the element
mouseout	onmouseout	when the cursor of the mouse leaves an element
mousedown	onmousedown	when the mouse button is pressed over the element
mouseup	onmouseup	when the mouse button is released over the element
mousemove	onmousemove	when the mouse movement takes place.

## Keyboard Event

Event Performed	Event Handler	Description
keydown & keyup	onkeydown & onkeyup	when the user press and then release the key

\*

# JavaScript - Events

## Form Event

Event Performed	Event Handler	Description
focus	onfocus	when the user focuses on an element
submit	onsubmit	when the user submits the form
blur	onblur	when the focus is away from a form element
change	onchange	when the user modifies or changes the value of a form element

## Windows Event

Event Performed	Event Handler	Description
load	onload	when the browser finishes the loading of the page
unload	onunload	when the visitor leaves the current webpage, the browser unloads it
resize	onresize	when the visitor resizes the window of the browser

## onClick Event Type

```
<html>
  <head>
    <script type="text/javascript">

      function sayHello() {
        alert("Hello World")
      }

    </script>
  </head>
  <body>
    <p>Click the following button and see result</p>

    <form>
      <input type="button" onclick="sayHello()" value="Say Hello" />
    </form>

  </body>
</html>
```

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

# JavaScript - Events

## onsubmit Event type

- **onsubmit** is an event that occurs when you try to submit a form. You can put your form validation against this event type.

```
<html>
  <head>
    <script type="text/javascript">
      <
        function validation() {
          all validation goes here
          .....
          return either true or false
        }
      </script>
    </head>
    <body>
      <form method="POST" action="t.cgi" onsubmit="return validate();">
        .....
        <input type="submit" value="Submit" />
      </form>
    </body>
  </html>
```

Here we are calling a **validate()**function before submitting a form data to the webserver. If **validate()** function returns true, the form will be submitted, otherwise it will not submit the data.

\*

# JavaScript - Events

## onmouseover and onmouseout

```
<html>
  <head>

    <script type="text/javascript">
      <!--
        function over() {
          document.write ("Mouse Over");
        }

        function out() {
          document.write ("Mouse out");
        }

      //-->
    </script>

  </head>
  <body>
    <p>Bring your mouse inside the division to see the result:</p>
    <div onmouseover="over()" onmouseout="out()">
      <h2> This is inside the division </h2>
    </div>

  </body>
</html>
```

- These two event types will help you create nice effects with images or even with text as well.
- The **onmouseover** event triggers when you bring your mouse over any element and the **onmouseout** triggers when you move your mouse out from that element.

## JavaScript - Events

```
<html>
<head>Javascript Events</head>
<br>
<body onload="window.alert('Page successfully loaded');">
<script>
<!--
document.write("The page is loaded successfully");
//-->
</script>
</body>
</html>
```

# JavaScript - BOM

- **Browser Object Model (BOM)** is used to interact with the browser
- Default object of browser is window means you can call all the functions of window by specifying window or directly.  
`window.alert("hello javatpoint");  
alert("hello javatpoint");`

## Window Object

Method	Description
alert()	displays the alert box containing message with ok button.
confirm()	displays the confirm dialog box containing message with ok and cancel button.
prompt()	displays a dialog box to get input from the user.
open()	opens the new window.
close()	closes the current window.
setTimeout()	performs action after specified time like calling function, evaluating expressions etc.

# JavaScript - BOM

```
<script type="text/javascript">  
function msg(){  
var v= confirm("Are u sure?");  
if(v==true){  
alert("ok");  
}  
else{  
alert("cancel");  
}  
  
}  
</script>  
  
<input type="button" value="delete record" onclick="msg()"/>
```

```
<script type="text/javascript">  
function msg(){  
var v= prompt("Who are you?");  
alert("I am "+v);  
  
}  
</script>  
  
<input type="button" value="click" onclick="msg()"/>
```

## JavaScript - BOM

- Ø **JavaScript history object** represents an array of URLs visited by the user. By using this object, you can load previous, forward or any particular page.

```
history.back() //for previous page  
history.forward() //for next page  
history.go(2) //for next 2nd page  
history.go(-2) //for previous 2nd page
```

- Ø **JavaScript navigator object** is used for browser detection. It can be used to get browser information such as appName, appCodeName, userAgent etc.

```
<script>  
document.writeln("<br/>navigator.appCodeName: "+navigator.appCodeName);  
document.writeln("<br/>navigator.appName: "+navigator.appName);  
document.writeln("<br/>navigator.appVersion: "+navigator.appVersion);  
;  
document.writeln("<br/>navigator.cookieEnabled: "+navigator.cookieEnabled);  
document.writeln("<br/>navigator.language: "+navigator.language);  
document.writeln("<br/>navigator.userAgent: "+navigator.userAgent);  
  
document.writeln("<br/>navigator.platform: "+navigator.platform);  
document.writeln("<br/>navigator.onLine: "+navigator.onLine);  
</script>
```

- Ø **JavaScript screen object** holds information of browser screen. It can be used to display screen width, height, colorDepth, pixelDepth etc.

# JavaScript - DOM

- **Document object** represents the whole html document.
- When html document is loaded in the browser, it becomes a document object.
- It is the **root element** that represents the html document. It has properties and methods. By the help of document object, we can add dynamic content to our web page.

## Methods of document object

Method	Description
<code>write("string")</code>	writes the given string on the document.
<code>writeln("string")</code>	writes the given string on the document with newline character at the end.
<code>getElementById()</code>	returns the element having the given id value.
<code>getElementsByName()</code>	returns all the elements having the given name value.
<code>getElementsByTagName()</code>	returns all the elements having the given tag name.
<code>getElementsByClassName()</code>	returns all the elements having the given class name.

# JavaScript - DOM

`document.getElementById()` method returns the element of specified id

```
<script type="text/javascript">  
function printvalue(){  
var name=document.form1.name.value;  
alert("Welcome: "+name);  
}  
</script>
```

```
<form name="form1">  
Enter Name:<input type="text" name="name"/>  
<input type="button" onclick="printvalue()" value="print name"/>  
</form>
```

```
<script type="text/javascript">  
function getcube(){  
var number=document.getElementById("number").value;  
alert(number*number*number);  
}  
</script>
```

```
<form>  
Enter No:<input type="text" id="number" name="number"/><br/>  
<input type="button" value="cube" onclick="getcube()"/>  
</form>
```

# JavaScript - DOM

`document.getElementsByName()` method returns all the element of specified name.

```
<script type="text/javascript">
function totalelements()
{
var allgenders=document.getElementsByName("gender");
alert("Total Genders:" +allgenders.length);
}
</script>
<form>
Male:<input type="radio" name="gender" value="male">
Female:<input type="radio" name="gender" value="female">

<input type="button" onclick="totalelements()" value="Total Genders">
</form>
```

## Example of document.getElementsByTagName() method

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

function countpara(){

var totalpara=document.getElementsByTagName("p");

alert("total p tags are: "+totalpara.length);

}

</script>

<p>This is a paragraph</p>

<p>Here we are going to count total number of paragraphs by getElementByTagName() method.

<p>Let's see the simple example</p>

<button onclick="countpara()">count paragraph</button>
```

## JavaScript - DOM

- **innerText** property can be used to write the dynamic text on the html document. Here, text will not be interpreted as html text but a normal text.
- Mostly in the web pages to generate the dynamic content such as writing the validation message, password strength etc.

```
<script type="text/javascript" >  
function validate0 {  
var msg;  
if(document.myForm.userPass.value.length>5){  
msg="good";  
}  
else{  
msg="poor";  
}  
document.getElementById('mylocation').innerText=msg;  
}  
  
</script>  
<form name="myForm">  
<input type="password" value="" name="userPass" onkeyup="validate0">  
Strength:<span id="mylocation">no strength</span>  
</form>
```

## JavaScript - DOM

`innerHTML` property can be used to write the dynamic html on the html document.

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

function showcommentform() {
var data="Name:<input type='text' name='name'><br>C
<br><textarea rows='5' cols='80'></textarea>
<br><input type='submit' value='Post Comment'>";
document.getElementById('mylocation').innerHTML=data;
}

</script>
<form name="myForm">
<input type="button" value="comment" onclick="showcommentform()">
<div id="mylocation"></div>
</form>
```

## JavaScript Form validation:

- It is important to validate the form submitted by the user because it can have inappropriate values. So validation is must.
- The JavaScript provides you the facility to validate the form on the client side so processing will be fast than server-side validation. So, most of the web developers prefer Javascript form validation.
- Through JavaScript, we can validate name, password, email, date, mobile number etc fields.

## JavaScript form validation example

- Next example, we are going to validate the name and password. The name can't be empty and password can't be less than 6 characters long.
- Here, we are validating the form on form submit. The user will not be forwarded to the next page until given values are correct.

# JavaScript – Form validation

## JavaScript Name & Password validation

```
<html>
<body>
<script>
function validateform(){
var name=document.myform.name.value;
var password=document.myform.password.value;

if (name==null || name==""){
    alert("Name can't be blank");
    return false;
} else if(password.length<6){
    alert("Password must be at least 6 characters long");
    return false;
}
}

</script>
</body>
<form name="myform" method="post"
      action="http://www.javatpoint.com/javascriptpages
      /valid.jsp" onsubmit="return validateform()" >
Name: <input type="text" name="name"><br/>
Password: <input type="password"
              name="password"><br/>
<input type="submit" value="register">
</form>
</body>
</html>
```

\*

# JavaScript – Retype Password validation

```
<!DOCTYPE html>
<html>
<head>
<script type="text/javascript">
function matchpass(){
var firstpassword=document.f1.password.value;
var secondpassword=document.f1.password2.value;
if(firstpassword==secondpassword){
return true;
}
else{
alert("password must be same!");
return false;
}
}
</script>
```

```
</head>
<body>
<form name="f1"
action="http://www.javatpoint.com/javascriptpages/valid.jsp" onsubmit="return matchpass()">
Password:<input type="password" name="password"
/><br/>
Re-enter Password:<input type="password"
name="password2"/><br/>
<input type="submit">
</form>
</body>
</html>
```

# JavaScript – Form validation

## JavaScript Number Validation

```
<script>
function validate(){
var num=document.myform.num.value;
if (isNaN(num)){
    document.getElementById("numloc").innerHTML="Enter Numeric value only";

    return false;
} else{
    return true;
}
}
</script>
<form name="myform" onsubmit="return validate()" >
Number: <input type="text" name="num"><span id="numloc"></span><br/>
<input type="submit" value="submit">
</form>
```

## JavaScript email validation

- There are many criteria that need to be follow to validate the email id such as:
  - email id must contain the @ and . character
  - There must be at least one character before and after the @.
  - There must be at least two characters after . (dot).
  - Let's see the simple example to validate the email field.

# JavaScript – Email validation

```
<script>
function validateemail()
{
var x=document.myform.email.value;
var atposition=x.indexOf("@");
var dotposition=x.lastIndexOf(".");
if (atposition<1 || dotposition<atposition+2 || dotposition+2>=x.length){
  alert("Please enter a valid e-
mail address \n atpostion:"+atposition+"\n dotposition:"+dotposition);
  return false;
}
</script>
<body>
<form name="myform" method="post" action="#" onsubmit="return validateemail(
 );">
Email: <input type="text" name="email"><br/>

<input type="submit" value="register">
</form>
```

\*

## References

- Jeffrey C. Jackson, WEB TECHNOLOGIES A Computer Science Perspective, Pearson Prentice Hall, Pearson Education, 2007
- <https://www.javatpoint.com/javascript-tutorial>

# THANK YOU

Java Script

230  
230



# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

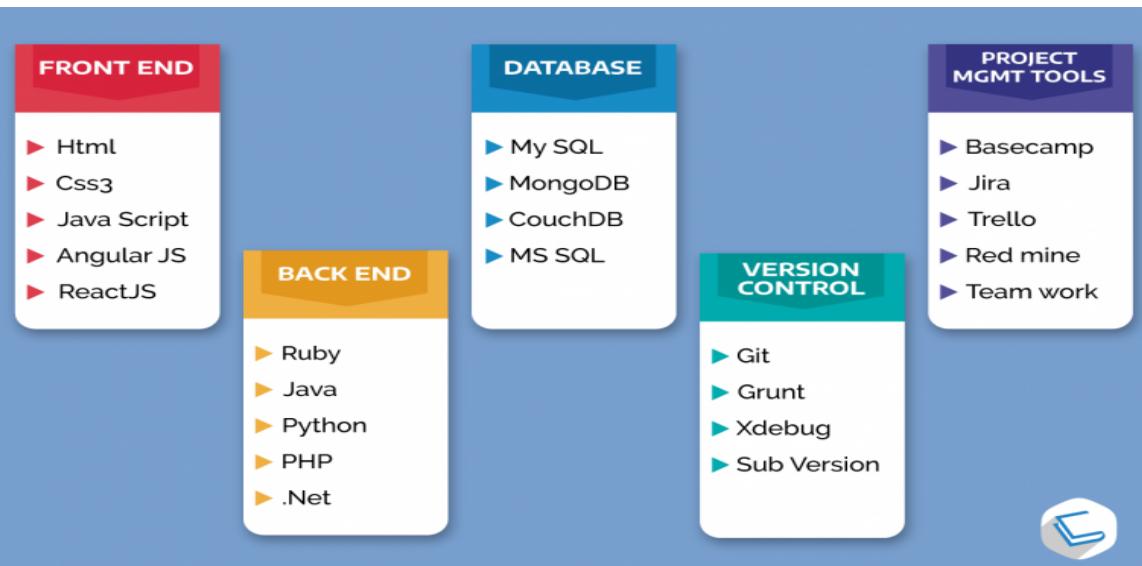
(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



# INT436

## FULL STACK WEB APPLICATION DEVELOPMENT

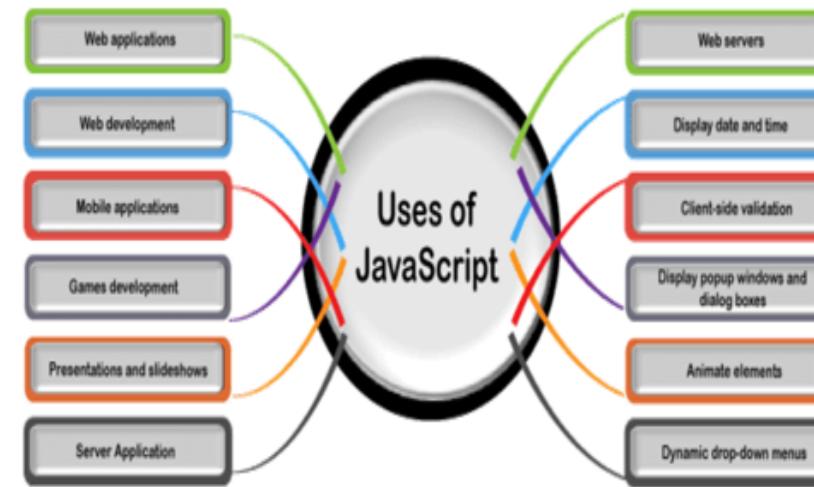




- Ø Java Script – Need
- Ø Java vs Java Script
- Ø Java Script
  - Ø Input
  - Ø Output
  - Ø Data Types
  - Ø Function
  - Ø Objects

# JavaScript - Need

- Ø JavaScript is a **light-weight object-oriented programming language**
  - Ø JavaScript enables dynamic interactivity on websites when it is applied to an HTML document.
  - Ø JavaScript helps the users to build **modern web applications** to interact directly without reloading the page every time.
  - Ø JavaScript is commonly used to dynamically modify HTML and CSS to update a user interface
- 14/08/2024  
11:20:57 am  
DOM API



# Java vs Java Script

Java	JavaScript
A general-purpose, object-oriented programming language	A lightweight, interpreted <b>scripting language</b> primarily used to add interactivity to web pages.
Runs on the JVM	An interpreted language, - the code is executed directly by the <b>browser's</b> JavaScript engine.
Strongly typed, with explicit declaration of variables	Loosely typed, with dynamic typing
Uses multithreading to handle multiple tasks simultaneously	Uses <b>event-driven</b> programming model

# JavaScript Input

- Ø Interact with the user - input
- Ø The **prompt method** is part of the JavaScript window object.
- Ø It displays a
  - Ø **dialog box** with a message to the user,
  - Ø an input field, and an **OK** and **Cancel** button.
- Ø Users can enter text into the input field, and when they press OK, the input is **returned as a string**.
- Ø If they press Cancel or close the dialog, null is returned.
  
- Ø The **prompt is a blocking function**, which means it halts JavaScript execution until the user interacts with the dialogue.

# JavaScript Input

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

<h1>The window Object</h1>
<h2>The prompt() Method</h2>

<p>click the button to demonstrate the prompt box.</p>

<button onclick="myFunction()">Try it</button>

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

<script>
function myFunction() {
  let person = prompt("Please enter your name", "SASTRA");
  if (person != null) {
    document.getElementById("demo").innerHTML =
    "Hello " + person + "! How are you today?";
  }
}
</script>
```

# JavaScript Output

- JavaScript can "display" data in different ways:
  - Ø Writing into an HTML element, using `innerHTML`.
  - Ø Writing into the HTML output using `document.write()`.
  - Ø Writing into an alert box, using `window.alert()`.
  - Ø Writing into the browser console, using `console.log()`.

[Code](#)

# JavaScript Output

## 1. Writing into an HTML element, using innerHTML.

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

<h1>My First web Page</h1>
<p>My First Paragraph</p>

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

<script>
document.getElementById("demo").innerHTML = 5
+ 6;
</script>

</body>
</html>
```

# JavaScript Output

## 2. writing into the HTML output using `document.write()`.

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

<h1>My First web Page</h1>
<p>My first paragraph.</p>

<script>
document.write(5 + 6);
</script>

</body>
</html>
```

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

<h1>My First web Page</h1>
<p>My first paragraph.</p>

<button type="button" onclick="document.write(5 + 6)">Try
it</button>

</body>
</html>
```

- Using `document.write()` after an HTML document is loaded, will delete all existing HTML - Testing

# JavaScript Output

3. writing into an alert box, using window.alert().

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

<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<script>
window.alert(5 + 6);
</script>

</body>
</html>
```

# JavaScript Output

4. writing into the browser console, using `console.log()`.

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

<script>
console.log(5 + 6);
</script>

</body>
</html>
```

# JavaScript Data Types

# JavaScript Data Types

Data Type	Description	Example
String	Textual data.	'hello' , "hello world!" , etc.
Number	An integer or a floating-point number.	3 , 3.234 , 3e-2 , etc.
BigInt	An integer with arbitrary precision.	900719925124740999n , 1n , etc.
Boolean	Any of two values: true or false .	true and false
undefined	A data type whose variable is not initialized.	let a;
null	Denotes a null value.	let a = null;
Symbol	A data type whose instances are unique and immutable.	let value = Symbol('hello');
Object	Key-value pairs of collection of data.	let student = {name: "John"};

# JavaScript Data Types

```
let name;  
console.log(name); // undefined
```

```
let name = "Felix";  
// assigning undefined to the name variable  
name = undefined  
console.log(name); // returns undefined
```

```
let number = null;  
  
if(null || undefined )  
{ console.log('null is true');}  
else  
{ console.log('null is false');}
```

# JavaScript Function

# JavaScript Function

- A function is an independent block of code that performs a specific task.

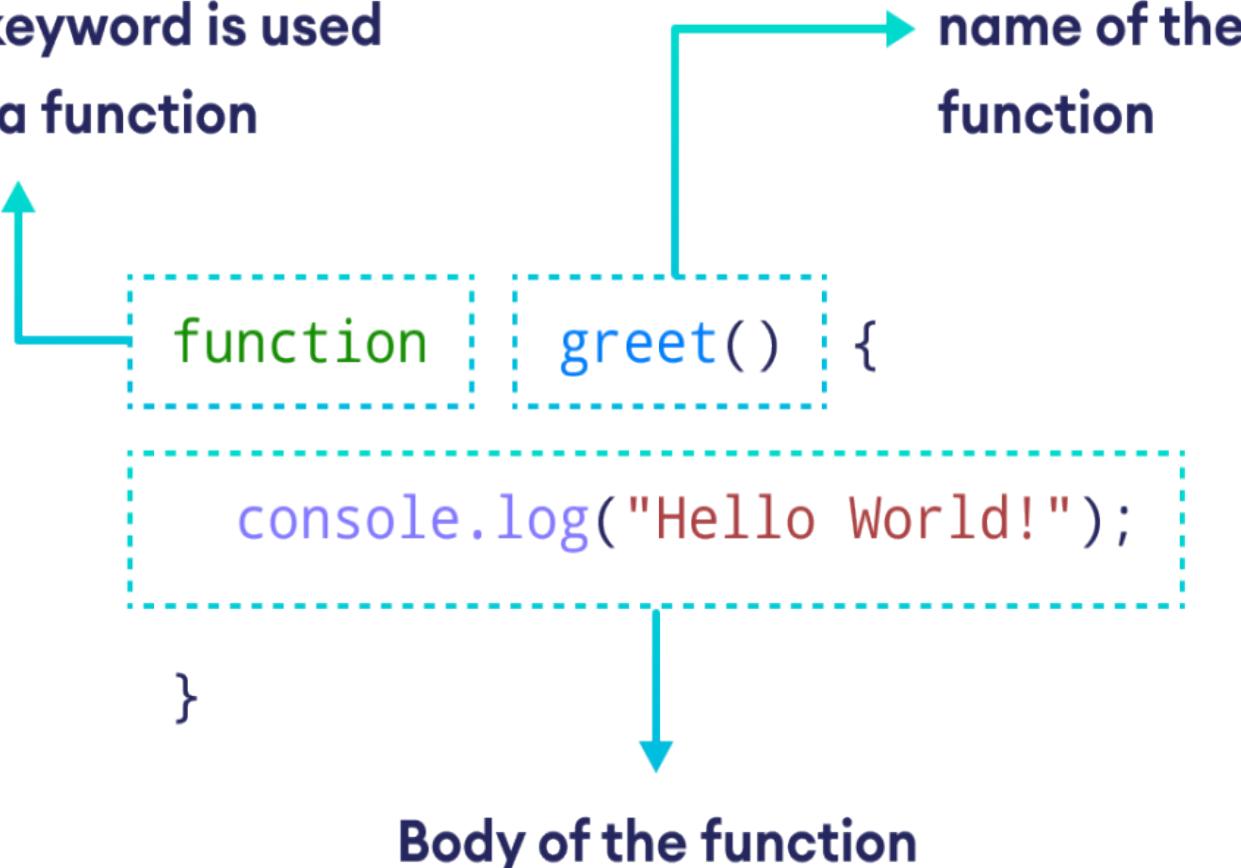
```
// create a function named greet()
```

```
function greet() {  
    console.log("Hello world!");  
}
```

```
// call the greet() function  
greet();
```

# JavaScript Function

**function keyword is used  
to create a function**



# JavaScript Function

```
function greet() { ←  
    // code  
}  
  
greet(); ←  
// code →
```

function call

# JavaScript Function

```
// function with a parameter called 'name'

function greet(name) {
    console.log(`Hello ${name}`);
}

// pass argument to the function
greet("John");

// pass "David" as argument
greet("David");

// Output: Hello John
Hello David
```

# JavaScript Function

```
// function with two arguments
function addNumbers(num1, num2) {
    let sum = num1 + num2;
    console.log(`Sum: ${sum}`);
}

// call function by passing two arguments
addNumbers(5, 4);

// Output:
// Sum: 9
```

# Function Expressions

- Ø In JavaScript, a function expression is a way to store functions in variables.

```
// store a function in the square variable
let square = function(num) {
    return num * num;
};

console.log(square(5));

// Output: 25
```

# JavaScript Objects

# JavaScript Objects

- JavaScript object is a variable that can store multiple data in **key-value pairs**.
- In JavaScript, the key-value pairs of an object are referred to as **properties**

```
// student object
const student = {
  firstName: "Jack",
  rollNo: 32
};
```

```
console.log(student);
```

```
// Output: { firstName: 'Jack', rollNo: 32 }
```

```
const objectName = {
  key1: value1,
  key2: value2,
  ...,
  keyN: valueN
};
```

# Access Object Properties

- You can access the value of a property by using its key.

```
const dog = {  
    name: "Rocky"  
};
```

```
// access property  
console.log(dog.name);
```

// Output: Rocky

```
const cat = {  
    name: "Luna",  
};
```

```
// access property  
console.log(cat["name"]);
```

// Output: Luna

# JavaScript Object Operations

```
const person = {  
    name: "Bobby",  
    hobby:  
    "Dancing",  
};  
  
// modify property  
person.hobby =  
    "Singing";  
  
// display the  
object  
console.log(person);  
  
// Output: { name:  
'Bobby', hobby:  
'Singing' }  
14/08/2024 13:20:53 am
```

```
const student = {  
    name: "John",  
    age: 20,  
};  
  
// add properties  
student.rollNo = 14;  
student.faculty =  
    "Science";  
  
// display the object  
console.log(student);  
  
// Output: { name:  
'John', age: 20, rollNo:  
14, faculty: 'Science' }
```

# JavaScript Object Operations

```
const employee = {  
    name: "Tony",  
    position: "officer",  
    salary: 30000,  
};  
  
// delete object property  
delete employee.salary  
  
// display the object  
console.log(employee);  
  
// Output: { name: 'Tony', position: 'officer' }
```

# JavaScript Object Methods

- Ø We can also include functions inside an object.
- Ø Functions that are defined inside objects are called methods.

```
const person = {  
    name: "Bob",  
    age: 30,  
  
    // use function as value  
    greet: function () {  
        console.log("Bob says Hi!");  
    }  
};  
  
// call object method  
person.greet(); // Bob says Hi!
```

objectName.methodKey  
()

# Javascript this Keyword

- Use 'this' keyword in an object method to access a property of the same object.

```
// person object
const person = { name: "John", age: 30,
// method
introduce: function () {
  console.log(`My name is ${this.name} and I'm
${this.age} years old.`);
}
};

// access the introduce() method
person.introduce();

// Output: My name is John and I'm 30 years old.
```

# Add Methods to an Object

```
// student object
let student = {
    name: "John"
};

// add new method
student.greet = function () {
    console.log("Hello");
};

// access greet() method
student.greet();

// Output: Hello
```

# JavaScript Constructor Function

# JavaScript Constructor Function

- A constructor function is used to create and initialize objects.

```
// constructor function
function Person () {
    this.name = "John",
    this.age = 23,
    this.greet =
    function () {
        console.log("Hello");
    }
}
```

```
// create objects
const person1 = new Person();
const person2 = new Person();

// access properties
console.log(person1.name);
// John
console.log(person2.name);
// John
```

# References

- [https://www.w3schools.com/js/js\\_output.asp](https://www.w3schools.com/js/js_output.asp)
- <https://www.javatpoint.com/what-are-the-uses-of-javascript>
- <https://www.programiz.com/javascript/get-started>





# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



## INT436

# FULL STACK WEB APPLICATION DEVELOPMENT



JAVASCRIPT



## Ø JavaScript ES6

Ø Declaration with let and const  
Keyword

Ø Arrow Function

Ø JavaScript Destructuring

Ø Rest Parameter

Ø Template Literals

Ø import and export

# JavaScript ES6

- JavaScript ES6 (also known as ECMAScript2015 or ECMAScript6) is the sixth edition of JavaScript introduced in June 2015.
- ECMAScript (European Computer Manufacturers Association Script) is the standard specification of ~~JavaScript~~ to ensure compatibility in all browsers and environments.

# JavaScript Declarations

- Previously, JavaScript only allowed variable declarations using the **var** keyword
- ES6 now allows you to declare variables using two more keywords: **let** and **const**
- The “**let**” keyword creates **block-scoped** variables, which means they are **only** accessible within a particular block of code.

```
{  
    // block of code  
  
    // declare variable  
    with let  
    let name = "Peter";  
  
    // can be accessed  
    here  
    console.log(name);  
// Peter  
}
```

Petean't be accessed  
ERROR!  
console.log(name);  
ReferenceError: name is not  
defined

# JavaScript let vs var

let

var

let is block-scoped.

var is function scoped.

let does not allow to redeclare variables.

var allows to redeclare variables.

Hoisting does not occur in let.

Hoisting occurs in var.

```
var a = 5;
// 5
var a = 3;
// 3
let a = 5;
let a = 3;
// error
```

```
var a = 5;
console.log(a);
// 5
{
    var a = 3;
    console.log(a);
    // 3
}
console.log(a);
// 3
```

```
Console.log(a);
var a; // undefined (not
an error)
console.log(a);
let a; // Uncaught
ReferenceError: a is
not defined
```

# Declaration With const Keyword

- The `const` keyword creates constant variables that cannot be changed after declaration.

```
// declare variable with const
const fruit = "Apple";
```

```
console.log(fruit);
```

```
// reassign fruit
// this code causes an error
fruit = "Banana";
```

```
console.log(fruit);
```

## Output:

Apple

Error: Assignment to constant  
variable

# JavaScript Arrow Function

- ES6 introduces a new way to **write function and function expressions** using => called arrow function.
- JavaScript arrow functions are a **concise syntax** for writing function expressions.

```
// an arrow function to add  
two numbers  
const addNumbers = (a, b) => a  
+ b;  
  
// call the function with two  
numbers  
const result = addNumbers(5,  
3);  
console.log(result);  
  
// output: 8
```

## Syntax:

```
let myFunction = (arg1, arg2,  
...argN) => {  
    statement(s)  
}
```

# JavaScript Arrow Function

```
const sayHello = () => "Hello,  
world!";
```

```
// call the arrow function and print  
its return value  
console.log(sayHello());
```

// Output: Hello, World!

```
const square = x => x * x;
```

```
// use the arrow function to  
square a number  
console.log(square(5));
```

// Output: 25

# JavaScript Arrow Function

```
let sum = (a, b) =>
{
    let result = a +
b;
    return result;
};
```

```
let result1 =
sum(5,7);
console.log(result1)
;
```

```
// Output: 12
```

# JavaScript Destructuring

- Ø The destructuring syntax makes it easier to **extract values from arrays or objects into individual variables.**

```
// assigning object attributes  
to variables  
const person = {  
    name: 'Sara',  
    age: 25,  
    gender: 'female'  
}  
  
let name = person.name;  
let age = person.age;  
let gender = person.gender;  
  
console.log(name); // Sara  
console.log(age); // 25  
console.log(gender); // female
```

```
// assigning object  
attributes to variables  
const person = {  
    name: 'Sara',  
    age: 25,  
    gender: 'female'  
}  
  
// destructuring assignment  
let { name, age, gender } =  
person;  
  
console.log(name); // Sara  
console.log(age); // 25  
console.log(gender); //
```

# JavaScript Destructuring

- The **order of the name does not matter** in object destructuring.

```
let { age, gender, name } = person;  
console.log(name); // Sara
```

- When destructuring objects, you **should use the same name** for the variable as the corresponding object key.

```
let {name1, age, gender} = person;  
console.log(name1); // undefined
```

- Ø **destructuring assignment using different variable names**

```
let { name: name1, age: age1, gender:gender1 } =  
person;
```

```
console.log(name1); // Sara
```

```
console.log(age1); // 25
```

# Array Destructuring

- You can also perform array destructuring in a similar way

```
const arrvalue = ['one', 'two', 'three'];

// destructuring assignment in arrays
const [x, y, z] = arrvalue;

console.log(x); // one
console.log(y); // two
console.log(z); // three
```

- You can assign the default values for variables while using destructuring

```
let arrvalue = [10];

// assigning default value 5 and 7
let [x = 5, y = 7] = arrvalue;

console.log(x); // 10
console.log(y); // 7
```

# JavaScript Destructuring

- Ø In object destructuring, you can pass **default values** in a similar way

```
const person = {  
    name: 'Jack',  
}
```

```
// assign default value 26 to age if undefined  
const { name, age = 26} = person;
```

```
console.log(name); // Jack  
console.log(age); // 26
```

## Destructuring

- Ø You can skip unwanted items in an array without assigning them to local variables.

```
const arrValue = ['one', 'two', 'three'];

// destructuring assignment in arrays
const [x, , z] = arrValue;

console.log(x); // one
console.log(z); // three
```

- You can assign the remaining elements of an array to a variable using the **spread** syntax

```
const arrValue = ['one', 'two', 'three', 'four'];
```

```
// destructuring assignment in arrays
// assigning remaining elements to y
const [x, ...y] = arrValue; //... operator expands
an iterable
```

```
console.log(x); // one
14/08/2024 console.log(y); // ["two", "three", "four"]
```

# JavaScript Destructuring

- Ø You can also assign the rest of the object properties to a single variable

```
const person = {  
    name: 'Sara',  
    age: 25,  
    gender: 'female'  
}
```

```
// destructuring assignment  
// assigning remaining properties to rest  
let { name, ...rest } = person;  
  
console.log(name); // sara  
console.log(rest); // {age: 25, gender:  
"female"}
```

# JAVASCRIPT Destructuring

- Ø The variable with the spread syntax **cannot have a trailing comma**.
- Ø You should **use this rest element** (variable with spread syntax) as the **last variable**.

```
const arrvalue = ['one', 'two', 'three', 'four'];
```

```
// throws an error
const [...x, y] = arrvalue;
```

```
console.log(x); // eror
```

# JAVASCRIPT REST Parameter

- Ø You can use the rest parameter (...) to represent an **infinite number of arguments** as an array. For example,

```
// function with ...args rest parameter
function show(a, b, ...args) {
    console.log("a:", a);
    console.log("b:", b);
    console.log("args:", args);
}
```

```
// call function with extra parameters
show(1, 2, 3, 4, 5);
```

**Output:**

```
a: 1
b: 2
args: [ 3, 4, 5
]
```

# JavaScript Template Literals

- The template literal makes it easier to include variables inside a string.
- For example, this was how we concatenated strings and variables before:

```
const firstName = "Jack";
const lastName = "Sparrow";

console.log("Hello " + firstName + " " + lastName);

// Output: Hello Jack Sparrow

const firstName = "Jack";
const lastName = "Sparrow";

console.log(`Hello ${firstName} ${lastName}`);

// Output: Hello Jack Sparrow
```

# JavaScript Import and export

- Before ES6, there was no standard way for developers to manage their code in separate files as modules.
- with ES6, we can finally manage modules with the import and export syntax.
- For example, suppose you have two JavaScript files named person.js and action.js.

```
//action.js
export default function
greet(name) {
    console.log(`Hi
${name}!`);
};
```

```
//person.js
import greet from
'./action.js';
console.log(greet("Abcd"
));
```

# JavaScript Import and export

Ø //Execute the index.html in browser and check the console in the browser

Ø O/p:

Hello, Abcd!

```
//index.html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>JavaScript Modules Example</title>
</head>
<h1>Import and export...</h1>
<body>
  <script type="module"
src="person.js"></script>
</body>
</html>
```

# Javascript classes

- In JavaScript ES6, classes provide a way to create blueprints for objects, similar to traditional object-oriented programming languages like C++ or Java.

```
//class1.js
// define a class named 'Person'
class Person {
    // class constructor to initialize the
    'name' and 'age' properties
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    // method to display a message
    greet() {
        console.log(`Hello, my name is
        ${this.name} and I am ${this.age} years old.`);
    }
}
```

# Javascript classes

```
// create two instances of the Person
class
let person1 = new Person("Jack", 30);
let person2 = new Person("Tina", 33);

// call greet() method on two instances
person1.greet();
person2.greet();
```

## Output

Hello, my name is Jack and I am 30 years old.

Hello, my name is Tina and I am 33 years old.

# References

- [https://www.w3schools.com/js/js\\_output.asp](https://www.w3schools.com/js/js_output.asp)
- <https://www.javatpoint.com/what-are-the-uses-of-javascript>
- <https://www.programiz.com/javascript/get-started>



- ∅ **what Is MERN?**
- ∅ **MERN Components**
- ∅ **why MERN?**





# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



## INT436

# FULL STACK WEB APPLICATION DEVELOPMENT



JAVASCRIPT



- Ø JavaScript ES6
- Ø for...in loop
- Ø for...of loop
- Ø Map
- Ø Java Script Object Notation
- Ø Asynchronous Programming
- Ø Callback Function

- The JavaScript `for...in` loop iterates over the keys of an object.

```
for (key in object)
{
    // body of
for...in
};
```

- Here,
- object** - The object whose keys we want to iterate over.

14 **key** - A variable  
11 that stores a

# for...in loop

Jack:  
\$24000,  
Paul:  
\$34000,  
Monica:  
\$55000

```
const salaries = {
    Jack: 24000,
    Paul: 34000,
    Monica: 55000
};

// use for...in to loop
// through
// properties of salaries
for (let i in salaries) {

    // access object key
    // using []
    // add a $ symbol
    // before the key
    let salary = "$" +
    salaries[i];

    // display the values
    console.log(`$${i}: ${salary}`);
}
```

INT436 - FS~~WAD~~ 29 // display the values  
1

# for...of loop

- The `for..of` loop in JavaScript allows you to iterate over iterable objects (arrays, sets, maps, strings etc).

```
for (element      of
     iterable) {
    // body      of
for...of
}
```

Here,  
**iterable** - an  
iterable object  
(array, set, strings,  
etc)  
10/8/2024  
11:20:59 am

```
// array
const students = ['John',
'Sara', 'Jack'];

// using for...of
for (let element of students )
{
    // display the values
    console.log(element);
}
```

**Output**  
John  
Sara  
Jack

# JavaScript Map

- The JavaScript ES6 has introduced two new data structures, i.e Map and weakMap.
- Map is similar to objects in JavaScript that allows us to store elements in a key/value pair.
- The elements in a Map are inserted in an insertion order.
- However, unlike an object, a map can contain objects, functions and other data types as key.

```
// create a Map
const map1 = new Map(); // an
empty map
console.log(map1); // Map {}
```

```
let map1 = new Map();
// insert key-value pair
map1.set('name', 'Jack');
console.log(map1); // Map {'name'=> 'Jack'}
```

# JavaScript Map

```
// Map with object key
let map2 = new Map();

let obj = {};
map2.set(obj, {name: 'Jack', age: "26"});

console.log(map2); // Map {} => {name:
"Jack", age: "26"}
```

```
console.log(map1.get('info')); // {name:
"Jack", age: "26"}
console.log(map1.has('info')); // true
map1.delete('info'); // true
map1.clear();
console.log(map1.size);
```

# JavaScript Map

```
let map1 = new Map();
map1.set('name', 'Jack');
map1.set('age', '27');
map1.set('salary',
'27000');
map1.set('Gender', M');
map1.set('city',
'Thanjavur');
```

```
// Looping through Map
for (let [key, value] of
map1) {
  console.log(key + ' -
' + value);
}
```

## Output

```
name - Jack
age - 27
salary -
27000
Gender - M
City -
Thanjavur
```

# Review Question

```
let map2= new Map();
```

```
map2.set('name' ,  

'Jack');  

map2.set('name' ,  

'Jack1');  

map2.set('name' ,  

'Jack2');  

Map(1) { 'name' =>  

'Jack2'}  

console.log(map2);  

let map2= new Map();
```

```
map2.set('name1' , 'Jack');  

map2.set('name2' , 'Jack');  

map2.set('name3' , 'Jack');  

console.log(map2);
```

```
Map(3) { 'name1' => 'Jack',  

'name2' => 'Jack', 'name3' =>  

'Jack'}
```

14/08/2024

11:20:59 am

```
let map2= new Map();
```

```
map2.set('name' , 'Jack');  

map2.set('name' , 'Jack');  

map2.set('name' , 'Jack');  

console.log(map2);
```

```
Map(1) { 'name' =>  

'Jack' }
```

```
let map2= new Map();
```

```
map2.set('name1' , 'Jack');  

map2.set(10 , 'Jack');  

map2.set(undefined ,  

'Jack');
```

```
console.log(map2);
```

```
Map(3) { 'name1' => 'Jack', 10 =>  

'Jack', undefined => 'Jack' }
```

INT436 – FSWAD

29

6

# JavaScript WeakMap

- The WeakMap is similar to a Map.
- However, WeakMap can only contain objects as keys.
- WeakMaps are not iterable

# Java Script Object Notation

- JSON stands for Javascript Object Notation.
- JSON is a text-based data format that is used to store and transfer data from a server to a client and vice-versa.
- In JSON, the data are in key/value pairs separated by a comma.
- JSON was derived from Javascript.
- So, the JSON syntax resembles JavaScript object literal syntax.
- However, the JSON format can be accessed and be created by other programming languages too.

```
// JSON Syntax
{
    "name": "John",
    "age": 22,
    "gender": "male",
}
```

# JSON Data & Object

- JSON data consists of key/value pairs similar to JavaScript object properties.
- The key and values are written in double quotes separated by a colon

```
// JSON data
"name": "John"
```

- The JSON object is written inside curly braces { }.
- JSON objects can contain multiple key/value pairs.
- For example

```
// JSON object
{ "name": "John", "age": 22 }
```

# Converting JSON to JavaScript Object

- You can convert JSON data to a JavaScript object using the built-in `JSON.parse()` function

```
// json object
const jsonData = '{ "name": "John",
"age": 22 }';

// converting to JavaScript object
const obj = JSON.parse(jsonData);

// accessing the data
console.log(obj.name); // John
```

# Converting JavaScript Object to JSON

- You can also convert JavaScript objects to JSON format using the JavaScript built-in `JSON.stringify()` function.
- For example,

```
// JavaScript object
const jsonData = { "name": "John",
"age": 22 };

// converting to JSON
const obj = JSON.stringify(jsonData);

// accessing the data
console.log(obj); // 
"{"name":"John","age":22}"
```

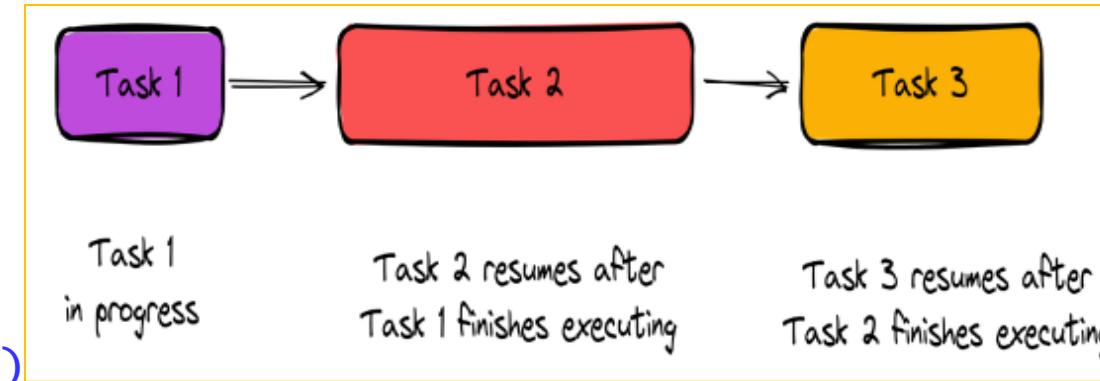
# What is Synchronous Programming?

```
// Define three functions
function firstTask()
  console.log("Task 1");
}
```

```
function secondTask()
  console.log("Task 2");
}
```

```
function thirdTask() {
  console.log("Task 3");
}
```

```
// Execute the functions
firstTask();
secondTask();
thirdTask();
14/08/2024
11:20:59 am
```



**Output**  
"Task 1"  
"Task 2"  
"Task 3"

# ASynchronous Programming

## - Need

- Ø Asynchronous programming is a way for a computer program to handle multiple tasks simultaneously rather than executing them one after the other.



```
console.log("Start of script");

setTimeout(function() {
    console.log("First timeout completed");
}, 2000);

console.log("End of script");
```

**Output:**

Start of script  
End of script  
First timeout completed

# CallBack Function

- In JavaScript, you can also pass a function as an argument to a function.
- This function that is passed as an argument inside of another function is called a callback function.

```
// function
function greet(name) {
    console.log('Hi' +
      ' ' + name);
}

greet('Peter'); // Hi
Peter
```

```
// function
function greet(name, callback)
{
    console.log('Hi' + ' ' +
name);
    callback();
}
```

```
// callback function
function callMe() {
    console.log('I am callback
function');
}
```

```
// passing function as an
argument
```

# Async Functions with async/await

- Async/Await is a feature that allows you to write asynchronous code in a more synchronous, readable way.
- `async` is a keyword that is used to declare a function as asynchronous.
- `await` is a keyword that is used inside an `async` function to pause the execution of the function until a promise is resolved.

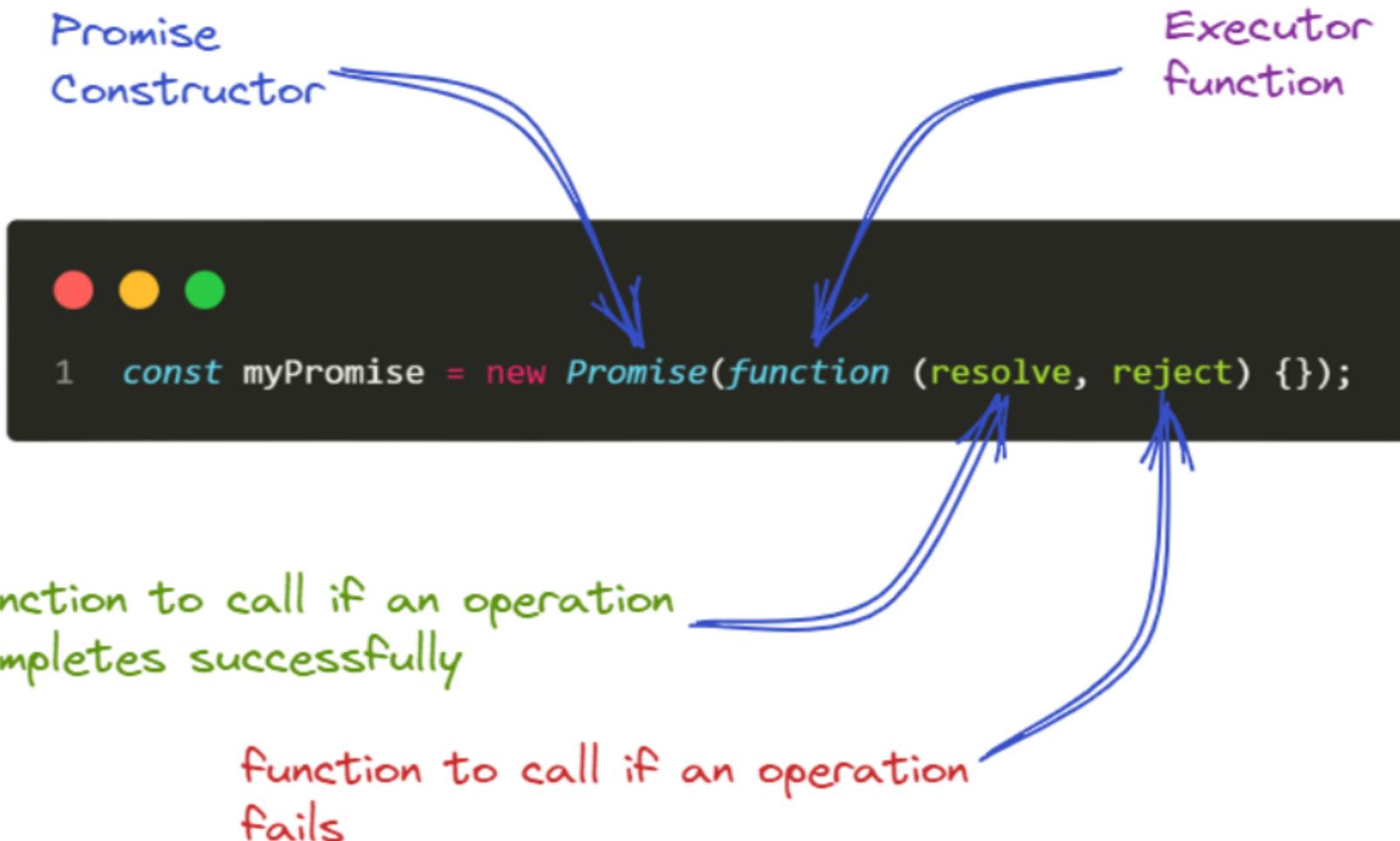
```
const response = await
fetch('https://jsonplaceholder.typicode.com/posts/1');
const data = await response.json();
console.log(data);
}

getData();
```

e

- A promise represents a way of handling asynchronous operations in a more organized way.
- It serves the same purpose as a callback but offers many additional capabilities and a more readable syntax.
- A promise in JavaScript is a placeholder for a future value or action.
- By creating a promise, you are essentially telling the JavaScript engine to "promise" to perform a specific action and notify you once it is completed or fails.
- Next, callback functions are then attached to the promise to handle the outcome of the action.
- These callbacks will be invoked when the promise is fulfilled (action completed successfully) or rejected (action failed).

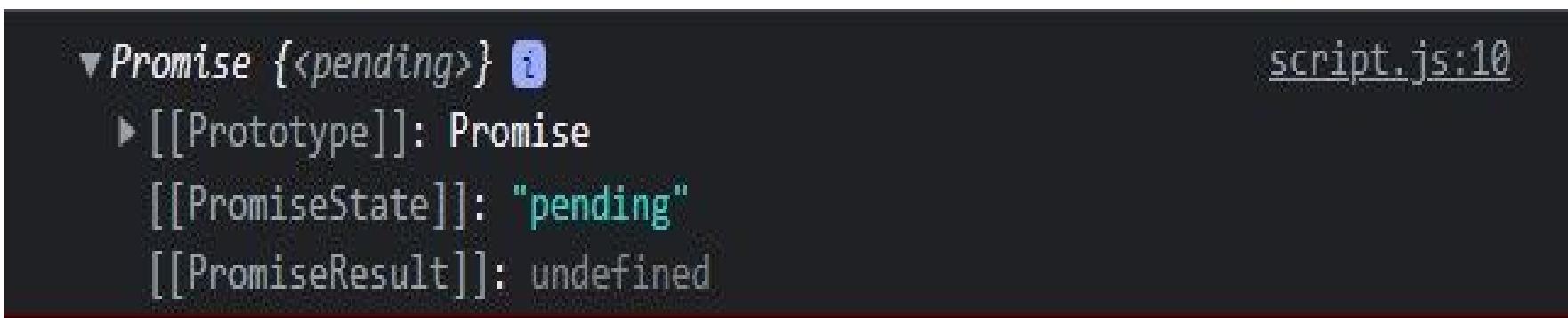
# How to Create a Promise?



# Promis e

```
// Initialize a promise
const myPromise = new Promise(function(resolve, reject)
=> {})

console.log(myPromise);
```



A screenshot of a browser's developer tools console. It shows the output of the code above. The promise object has a pending state and no result.

```
▼Promise {<pending>} ⓘ script.js:10
▶[[Prototype]]: Promise
[[PromiseState]]: "pending"
[[PromiseResult]]: undefined
```

- Ø As you can see, the promise has a *pending* status and an *undefined* value.
- Ø This is because nothing has been set up for the promise object yet
- Ø so it's going to sit there in a pending state forever

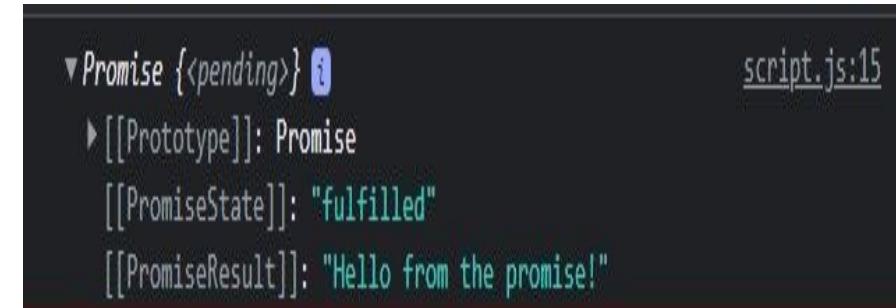
14 without any value or result.

T436 – FSWAD  
11:21:00 am

# Promis e

- Now, let's set up myPromise to resolve with a string printed to the console after 2 seconds.

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Hello from the promise!");
  }, 2000);
});
```



A promise has three states:

- Pending:** initial state, neither fulfilled nor rejected.
- Fulfilled:** meaning that an operation was completed successfully.
- Rejected:** meaning that an operation failed.

# How to Consume a Promise

- Consuming a promise involves the following steps:
- obtain a reference to the promise:
  - To consume a promise, you first need to obtain a reference to it.
  - Based on the code from the previous section, our reference to a promise will be the myPromise object.
- Attach callbacks to the promise:
  - Once you have a reference, you can attach callback functions by using the .then and .catch methods.
  - The .then method is called when a promise is fulfilled and the .catch method is called when a promise is rejected.
- wait for the promise to be fulfilled or rejected:

– Once you've attached callbacks to the promise, you can wait for the promise to be fulfilled or rejected.

- Once the promise is fulfilled,
  - the .then callback method will be called with the resolved value.
  - And if the promise is rejected,
    - the .catch method will be called with an error

myPromise.message.

```
.then((result) => {  
  console.log(result);  
})  
.catch((error) => {  
  console.log(error);  
})  
.finally(() => {  
  //code here will be executed  
  //regardless of the status  
  //of a promise (fulfilled or  
  //rejected)  
});
```

## Output

Hello from the  
promise!

# Example

```
// returns a promise
```

```
let countValue = new Promise(function  
(resolve, reject) {  
    resolve("Promise resolved");  
});
```

```
// executes when promise is resolved  
successfully
```

```
countValue  
.then(function successValue(result) {  
    console.log(result);  
})
```

```
.then(function successValue1() {  
    console.log("You can call multiple  
functions this way.");  
});
```

Output:

Promise resolved  
You can call multiple  
functions this way.

# Example

```
// a promise
let promise = new Promise(function
(resolve, reject) {
  setTimeout(function () {
    resolve('Promise resolved'), 4000);
});
```

```
// async function
async function asyncFunc() {

  // wait until the promise resolves
  let result = await promise;

  console.log(result);
  console.log('hello');
}
```

```
// calling the async function
asyncFunc();
```

**Output**

Promise  
resolved  
hello

# References

- <https://www.programiz.com/javascript/promise>
- <https://www.freecodecamp.org/news/asynchronous-programming-in-javascript/#>



- ∅ **what Is MERN?**
- ∅ **MERN Components**
- ∅ **why MERN?**





# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

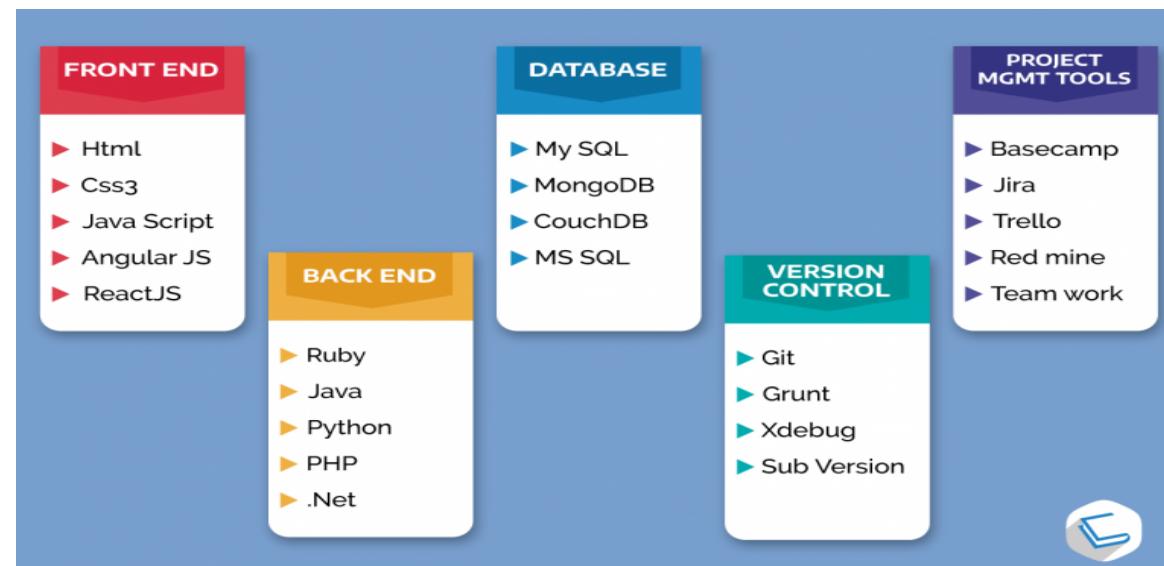
DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



# INT436 FULL STACK WEB APPLICATION DEVELOPMENT



Dr.  
Karthikeyan  
B



- Ø COURSE OBJECTIVES
- Ø SYLLABUS
- Ø TEXT AND REFERENCE BOOKS
- Ø PREREQUISITE
- Ø ASSIGNMENT
- Ø CIA
- Ø Web Application



# Overview



# Back End

- The back end refers to the **server-side** part of a web application.
- It includes everything that the user does not see, encompassing the **server, application, and database**

Components:

1. **Server:** The hardware or software that provides services to other software programs.
2. **Application:** The software that runs on the server and handles business logic, processes requests, and returns responses to the client.
3. **Database:** While the database itself can be considered part of the back end, the back end encompasses more than just the database. It includes the entire infrastructure that handles data processing and business logic.

# Back End (Contd..)

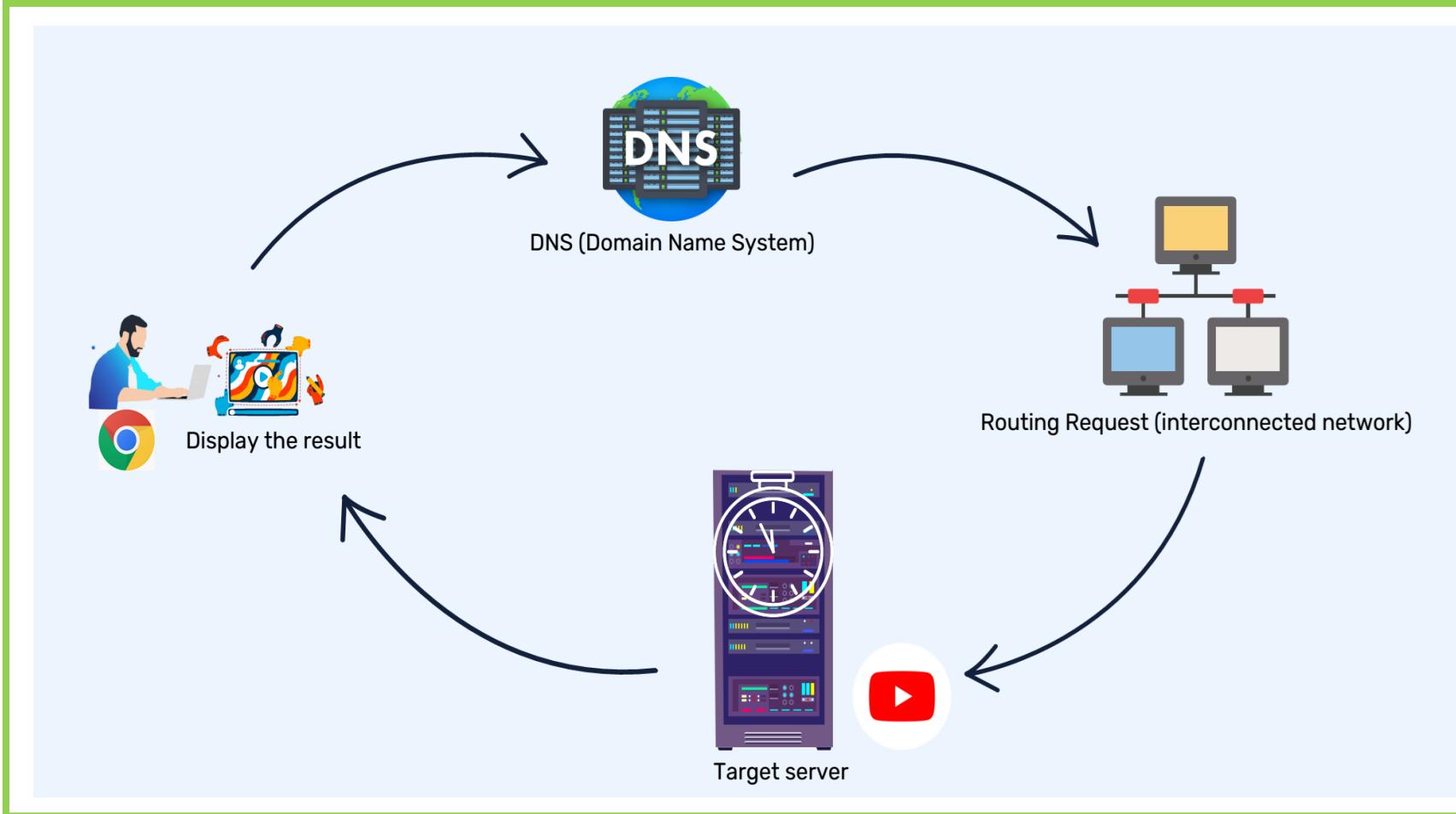
## Technologies:

- Programming languages: Python, Java, Ruby, PHP, Node.js
- Frameworks: Django, Flask, Spring, Rails, Express.js
- Servers: Apache, Nginx
- APIs: RESTful APIs, GraphQL

## Responsibilities:

- Processing user requests
- Managing application logic
- Interacting with the database
- Ensuring security, authentication, and authorization
- Handling business logic

# How internet works



<https://weqtechnologies.com/why-should-businesses-use-web-applications/>



- A **web application** is software that runs in your web browser
  - A **web application** is an application program that is stored on a remote server and delivered over the internet through a browser interface.
- <https://www.techtarget.com/searchsoftwarequality/definition/web-application-web-app>



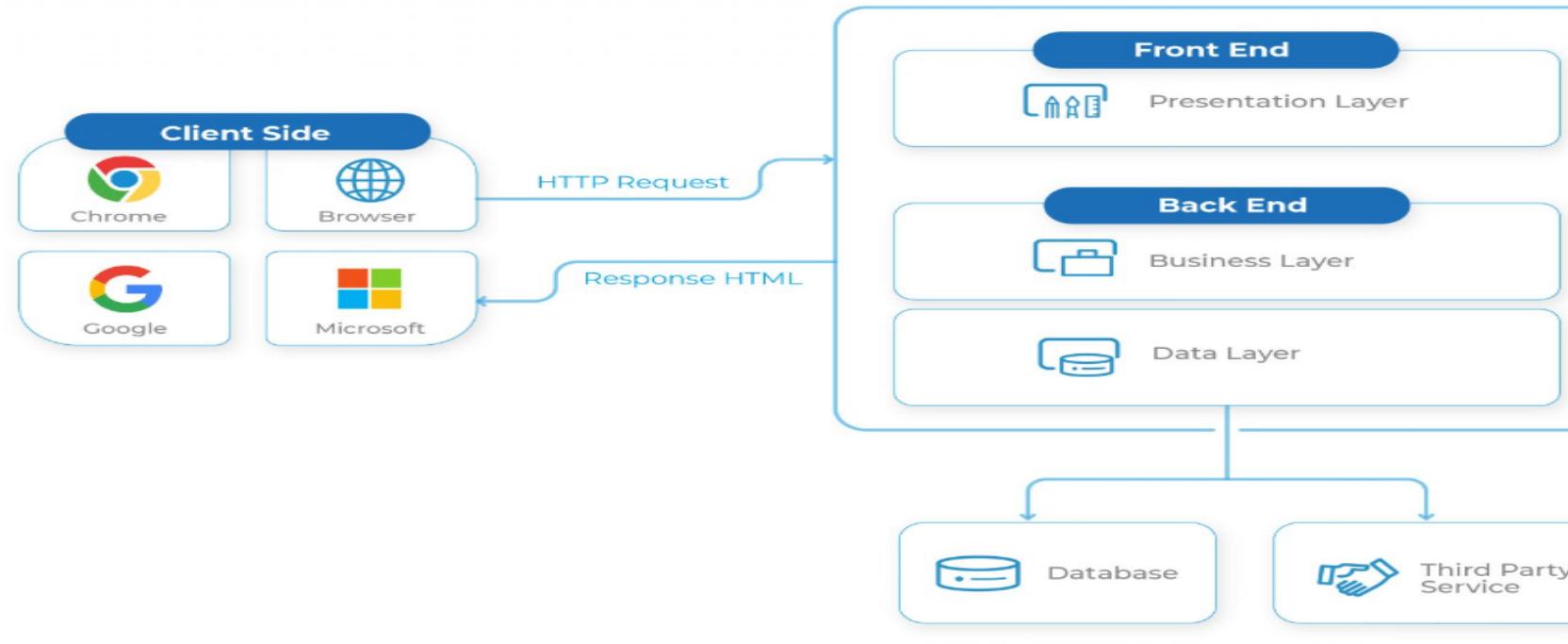
- **web application** needs authentication.
- The web application uses a combination of server-side scripts and client-side scripts to present information.
- It requires a server to manage requests from the users.
- A **website** provides visual and text content that users can view and read.

## What is the difference between a web app and a native app?

- A native app is a computer program that has been specifically designed for a particular user environment.
- One of the most **common types of native apps** are **mobile apps**, which are developed using specific programming code.
- Unlike web apps, native or mobile apps are downloaded by the user to their mobile device, usually through app stores.
- Native apps can only be accessed on the device they have been downloaded

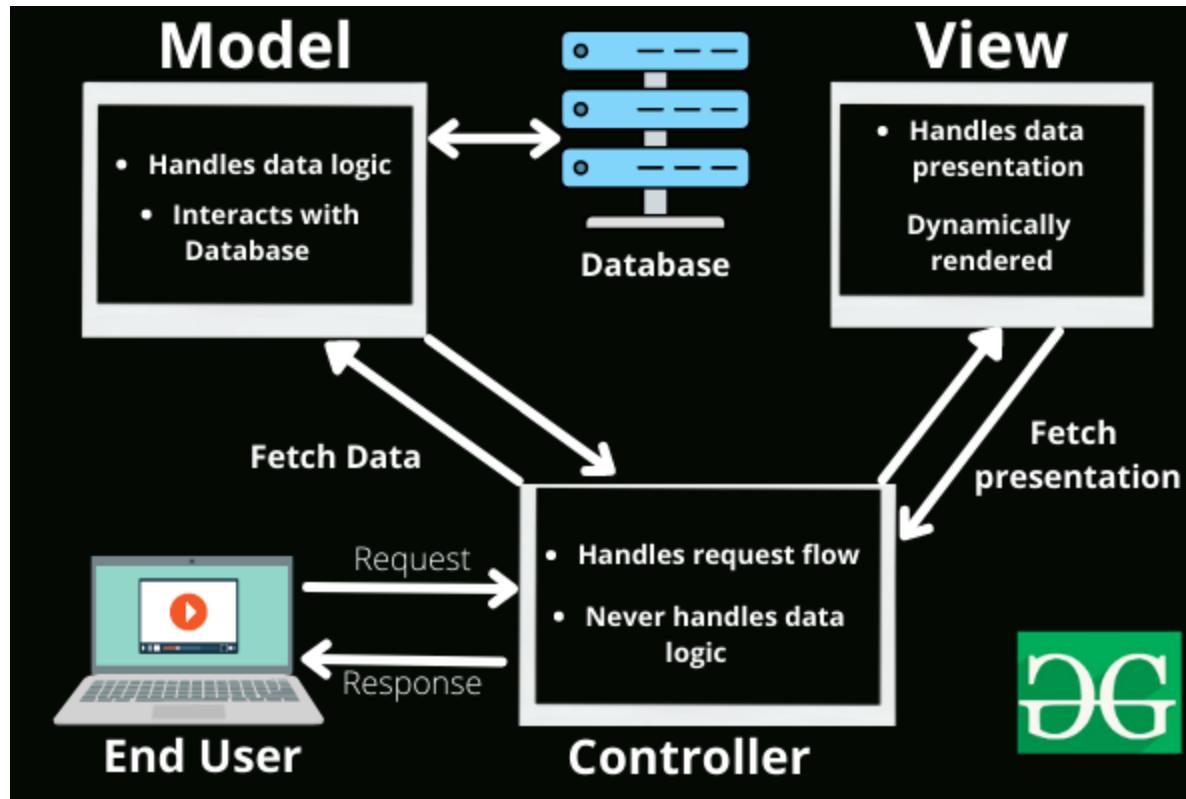
# Web Application Components

## Standard Web Application Architecture



<https://www.clickittech.com/devops/web-application-architecture/>

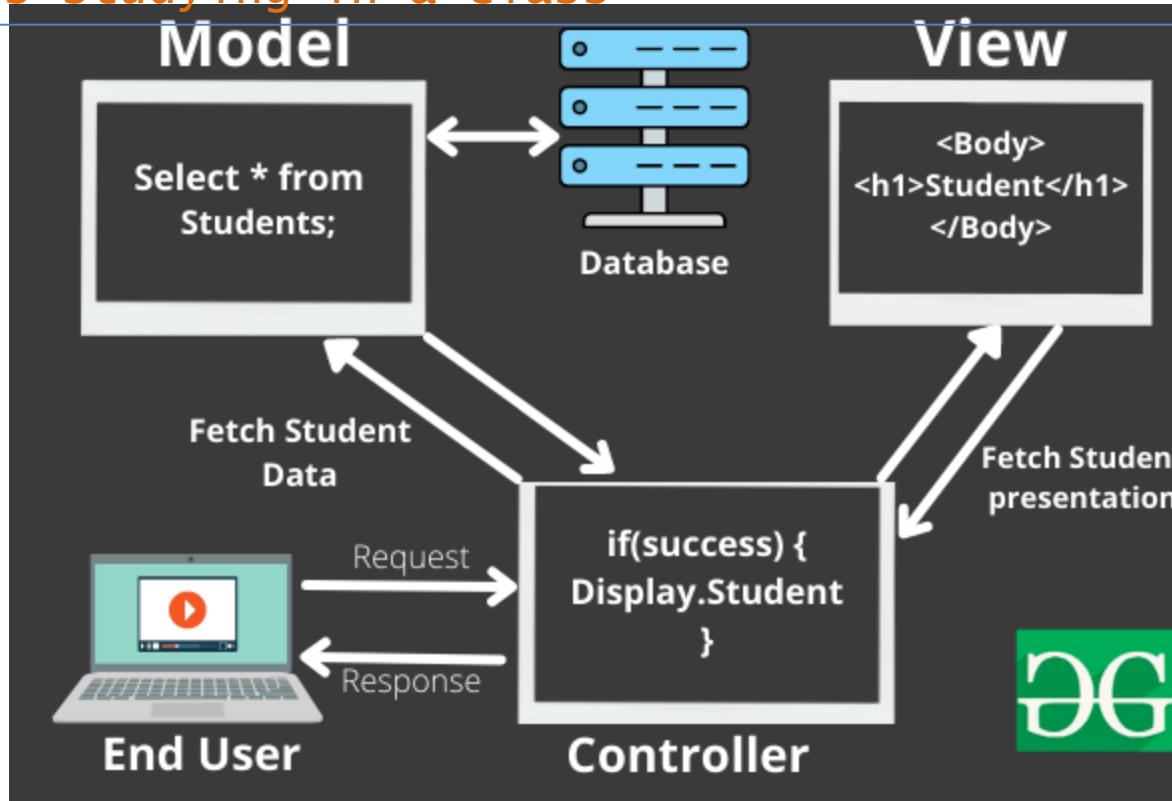
# MVC Framework



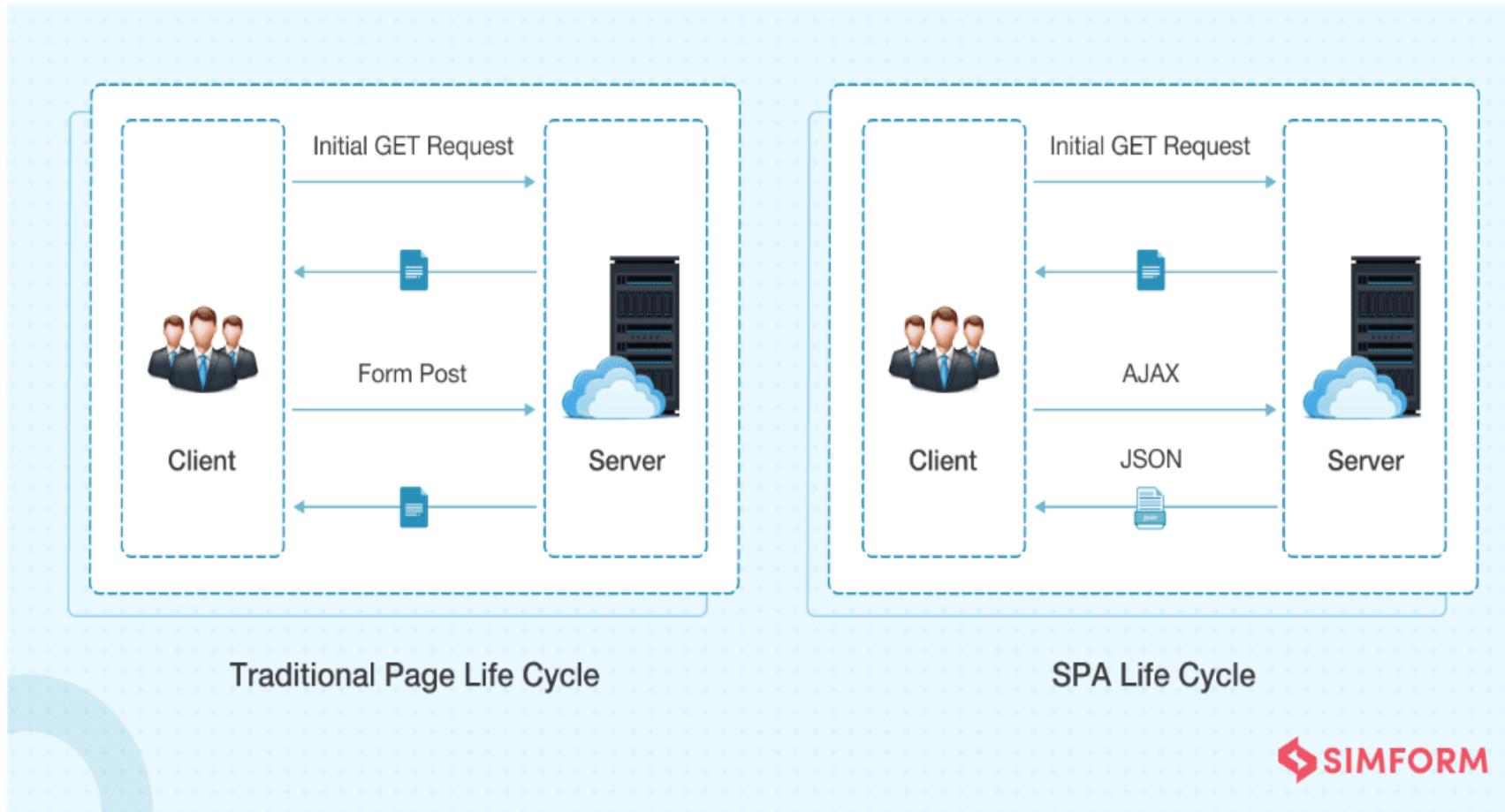
<https://www.geeksforgeeks.org/mvc-framework-introduction/>

# MVC Framework

- Ø End-user sends a request to a server to get a list of students studying in a class



<https://www.geeksforgeeks.org/mvc-framework-introduction/>



<https://www.simform.com/blog/web-application-architecture>

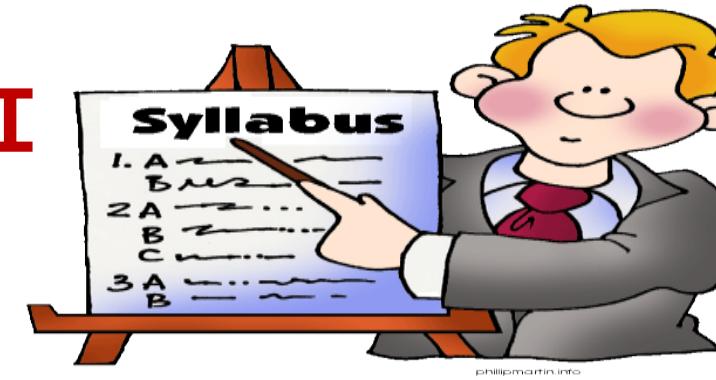
# Course Objectiv



- To help the learners
  - To develop modern, complex, responsive and scalable web applications.

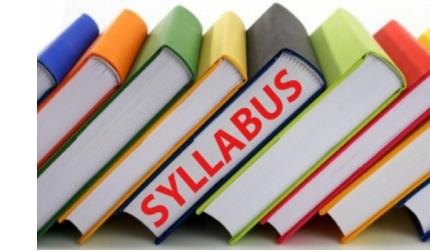
- The learner will be able to
  - Build single-page client-side web applications using React.
  - Organize large volumes of data and manage them using MongoDB
  - Create Restful API design with Node, Express, and MongoDB
  - Illustrate the fundamental elements of Client-side routing
  - Develop applications that use the same code base in the server as well as the client.

# UNIT - I



- **Introduction**
  - what Is MERN? - MERN Components - why MERN?
  - Server-Less Hello World - Server Setup - Build-Time JSX Compilation.
- **React Components**
  - Issue Tracker - React classes - Composing Components - Passing Data - Dynamic Composition.

# UNIT - II



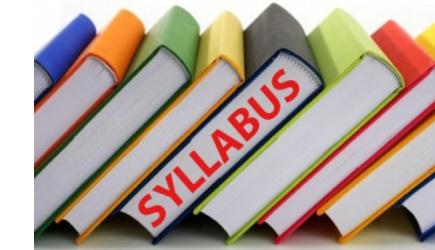
- **React State:**
  - Setting State - Async State Initialization - Event Handling - Communicating from Child to Parent - Stateless Components - Designing Components.
- **Hooks:**
  - Introducing Hooks - Hooks at a Glance - Using the State Hook - Using the Effect Hook - Rules of Hooks.
- **Express REST APIs:**
  - REST - Express - The List API - The Create API - Using the List API - Using the Create API - Error Handling
- **Using MongoDB:**
  - MongoDB Basics - MongoDB Node.js Driver - Reading from MongoDB - Writing to MongoDB

# UNIT - III



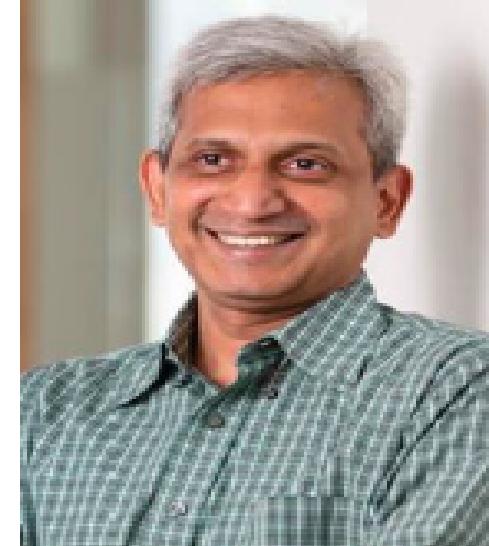
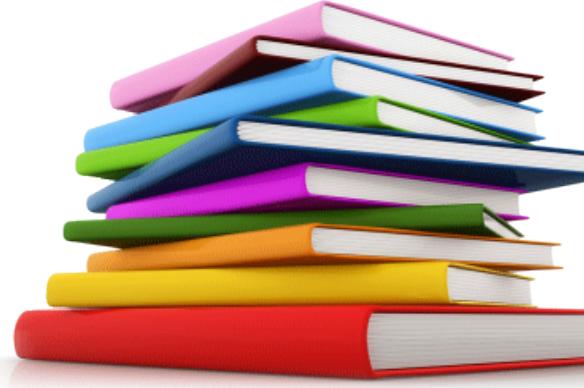
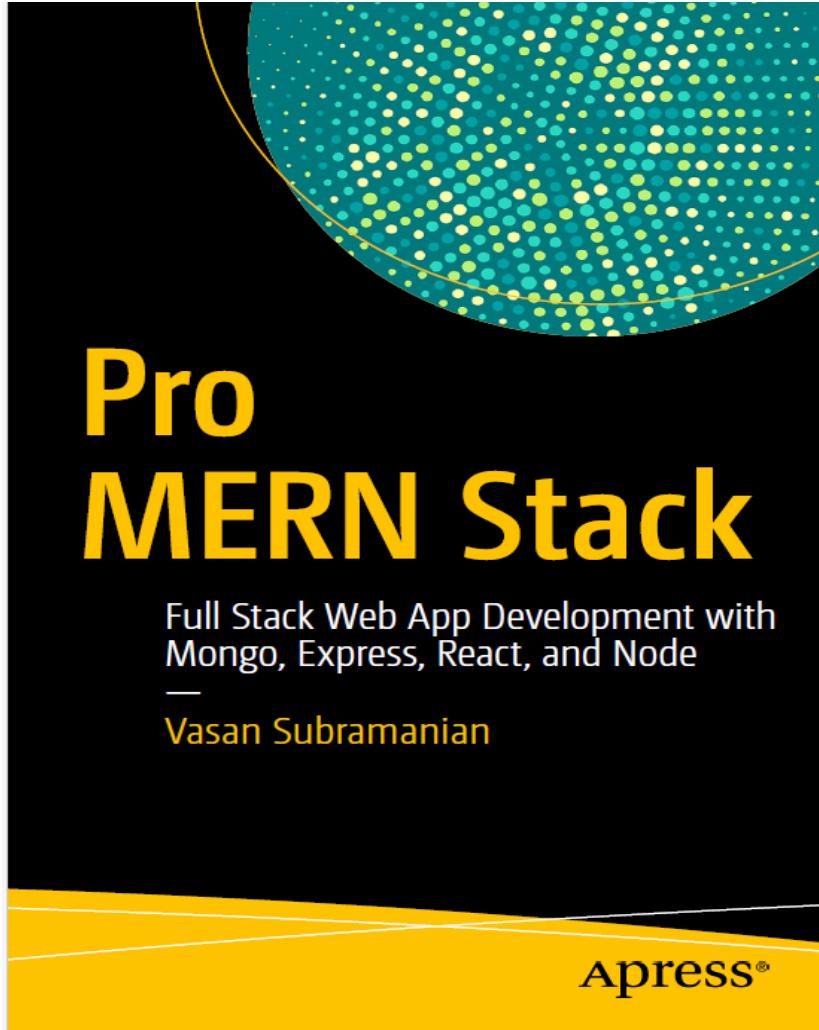
- **Modularization and webpack:**
  - Server-Side Modules - Introduction to Webpack
  - Using Webpack Manually - Transform and Bundle
  - Libraries Bundle - Hot Module Replacement HMR
  - Using Middleware - Debugging
  - Server-Side ES2015 - ESLint
- **Routing with React Router:**
  - Routing Techniques - Simple Routing - Route Parameters
  - Route Query String - Programmatic Navigation
  - Nested Routes - Browser History
- **Forms:**
  - More Filters in the List API - Filter Form
  - The Get API - Edit Page - UI Components
  - Update API - Using Update API - Delete API - Using the Delete API

# UNIT - IV



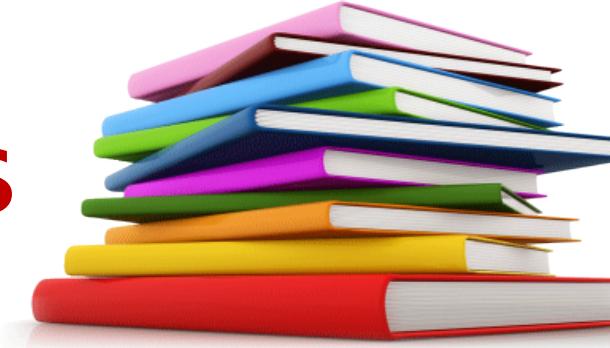
- **React-Bootstrap:**
  - Bootstrap Installation – Navigation – Table and Panel – Forms – Alerts – Modals.
- **Server Rendering:**
  - Basic Server Rendering – Handling State – Initial State – Server-Side Bundle Back-End HMR
  - Routed Server Rendering – Encapsulated Fetch
- **Node.js:**
  - Introduction – First Application – NPM – Call back Concept – Event Loop – Event Emitter – web Module – Express Framework.

# TEXT BOOK



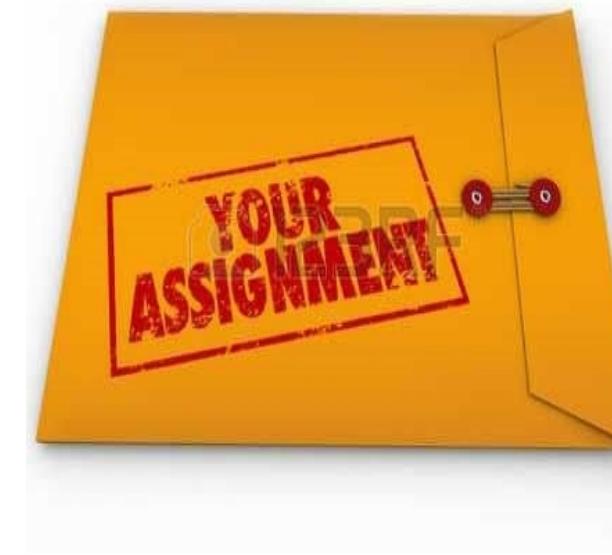
**Vasan Subramanian,**  
“Pro MERN Stack”,  
Apress, 2017.

# REFERENCES



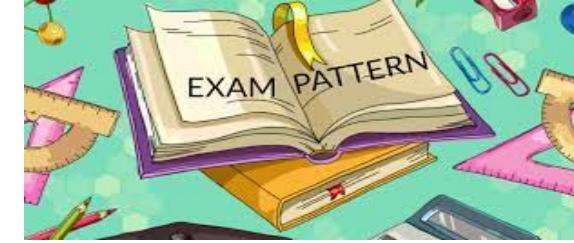
1. Rader Dabit, React Native IN ACTION.  
Manning, 2019.

# ASSIGNMENT



- **ASSIGNMENT 1**
- **ASSIGNMENT 2**
- **ASSIGNMENT 3**

# Internal Assessment



- Ø Maximum marks - 50.
- Ø 40 marks for CIA
- Ø 5 marks for assignments / Quiz.
- Ø 5 marks for model test (Lab)

# End Semester Exam



- Problem Based Assessment
  - 3 Hours
  - 1 Scenario Based Question





# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

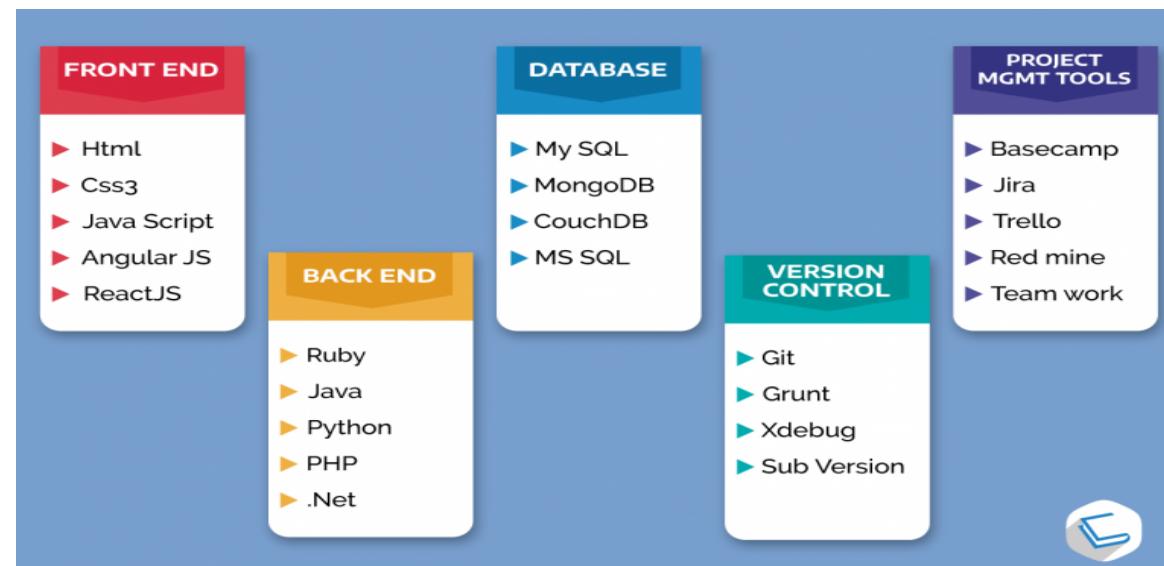
DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



# INT436 FULL STACK WEB APPLICATION DEVELOPMENT



Dr.  
Karthikeyan  
B

- Introduction
- What is MERN?
- MERN Components
- Why MERN?

# Introduction

- Full stack development
  - refers to the end-to-end application software development, including the front end and back end.
- Web application development

# what is MERN?

- Any web application is made by using **multiple technologies**. The combination of these technologies is called a “**stack**”.
- LAMP Stack
  - Linux, Apache, MySQL and PHP
  - All open-source components
- MEAN Stack
  - MongoDB, Express, AngularJS and Node.js
- MERN Stack
  - MongoDB, Express, ReactJS and Node.js
- Single page applications (SPA)
  - Avoids refreshing a web page to display new content

# what is MERN? (Contd..)

- ReactJS / AngularJS
  - A front-end framework based on Model-view-Controller (MVC) design pattern
- MongoDB
  - NOSQL database
- Node.js
  - a server-side JavaScript runtime environment that allows you to run JavaScript on the server side.
  - Event-Driven Architecture, Single-Threaded, Scalability
- Express
  - a web application framework built on top of Node.js

# MERN Components

- React
  - why Facebook invented
  - Declarative
  - Component-Based
  - No templates
  - Isomorphic
- Node.js
- Express
- MongoDB
- Tools and Libraries

# MERN Components

- React
  - An **open-source JavaScript library** maintained by Facebook
  - React is **not a full-fledged MVC framework**. It is a **JavaScript library**
- React to **render** a view (the v in MVC)
  - but how to tie the rest of the application together is completely up to you

# why Facebook invented

- React was born in Facebook's Ads organization
  - a typical client-side MVC model
  - **cascading updates** became difficult to maintain
  - **two-way data binding**
    - sharing data between a component class and its template
    - change data in one place, it will automatically reflate at the other end
    - **automatic synchronization** of data happens between the **Model** and the **view**
- **Two-way data binding** refers to the synchronization of data between the **model** and the **view** in both directions. This means that **any change in the UI** (view) **will update the model**, and any change in **the model** will update the UI.

# Declarative

- React views are **declarative**
  - As a programmer, **don't** have to **worry about** managing the effect of **changes** in the view's state or the data
  - You **don't worry about** transitions or mutations in the **DOM**
  - A React component declares how the view looks like, given the data.
  - The React library figures out how the new view looks, and just applies the changes between the old view and the new view.
  - entire screen may not be **refreshed** – using **virtual DOM**

# Component-Based

- The **fundamental building block** of React is a component, which **maintains its own state** and **renders itself**
- In React, all you do is **build components**.
- Then, you **put components together** to make another **component** that depicts a complete view or page
- A **component encapsulates** the state of data and the view, or how it is rendered.
- **Components talk** to each other by sharing state information

# No Templates

- Many web application frameworks rely on templates to automate the task of creating repetitive HTML or DOM elements.
- React uses a full-featured programming language(**Java Script**) to construct repetitive or conditional DOM elements.
- [Understanding the Role of Templates and Components](#)
- [JS Vs JSX](#)

# Isomorphic

- React can be run on the server too.
  - i.e the same code can run on both server and the browser.
  - [Isomorphic](#)

# MERN Components

- React
  - why Facebook invented
  - Declarative
  - Component-Based
  - No templates
  - Isomorphic
- Node.js
- Express
- MongoDB
- Tools and Libraries

# Node.js

- Node.js is JavaScript outside of a browser.
- The Node.js runtime runs JavaScript programs.
- In a browser, you can load multiple JavaScript files, but you need an HTML page to do all that.
- But for Node.js, there is no HTML page that starts it all.
- Node.js ships with a bunch of core modules compiled into the binary.
  - These modules provide access to the operating system elements such as the file system, networking, input/output, etc

# Node.js

- Node.js and npm
  - **npm is the default package manager for Node.js.**
  - npm registry ([www.npmjs.com](http://www.npmjs.com)) is a public repository
  - npm tops the list of module or package repositories, having more than 250,000 packages
    - **Maven**, which used to be the biggest two years back, has just half the number now
    - This shows that **npm is not just the largest, but also the fastest growing repository**
- Node.js Is Event Driven
  - Node.js has an asynchronous, event-driven, non-blocking input/output (I/O) model
  - Node.js, has no threads

# MERN Components

- React
  - why Facebook invented
  - Declarative
  - Component-Based
  - No templates
  - Isomorphic
- Node.js
- Express
- MongoDB
- Tools and Libraries

# Express

- Node.js is just a runtime environment that can run JavaScript
- **Express is the framework** that simplifies the task of writing your **server code**.
- The **Express framework** lets you **define routes**, specifications of **what to do** when a **HTTP request** matching a certain pattern arrives.
- Express **parses request URL**, headers, and parameters for you.

# MERN Components

- React
  - why Facebook invented
  - Declarative
  - Component-Based
  - No templates
  - Isomorphic
- Node.js
- Express
- **MongoDB**
- Tools and Libraries

# MongoDB

- It is a **NoSQL** document-oriented database, with a flexible schema and a JSON-based query language
- NoSQL stands for “**non-relational**”
- There are **two attributes** of NoSQL
  - The first is the ability to horizontally scale by **distributing the load over multiple servers**
  - NoSQL databases are **not necessarily relational databases.**
    - The difference in the representation in the application and on disk is sometimes called **impedance mismatch**.
    - MongoDB avoids object relational mapping (**ORM**)
      - i.e no need to convert or map the objects that the code deals with to relational tables

# MongoDB (Contd..)

- Document-Oriented
  - The **unit of storage** (comparable to a row) is a **document**, or an **object**, and **multiple documents** are stored in **collections**
  - Every document in a collection has a **unique identifier**.
  - In relational DBs – need primary and foreign key concepts
    - In MongoDB, the entire details as a **single document**, fetch it, and update it in an **atomic** operation.
  - MongoDB allowing nesting by array fields and **JSON** fields
  - MongoDB has the ability to index on **deeply nested fields**

# MongoDB (Contd..)

- Schema-Less
  - Storing an object in a MongoDB database **does not have to follow a prescribed schema.**
  - you don't need to add/rename columns in the schema.
  - During database migration may have some problem. So it is better to have a strict or semi-strict schema.
- Java Script Based
  - For **relational databases**, there is a query language called **SQL**.
  - For **MongoDB**, the query language is based on **JSON**.
  - Data is also interchanged in **JSON**(**JavaScript Object Notation**) format.

# MERN Components

- React
  - why Facebook invented
  - Declarative
  - Component-Based
  - No templates
  - Isomorphic
- Node.js
- Express
- MongoDB
- Tools and Libraries

# Tools and Libraries

- React-Router
  - React supplies only the **view rendering capability** and helps manage interactions in a single component.
  - **Routing (Navigation)**
    - transitioning between different views of the component and **keeping the browser URL in sync**
- React-Bootstrap
  - Bootstrap, the most popular open source **CSS framework**(front-end toolkit)
  - CSS libraries built for React(Material-UI)
- Webpack
  - a **JavaScript module bundler** that is commonly used with React to bundle and manage

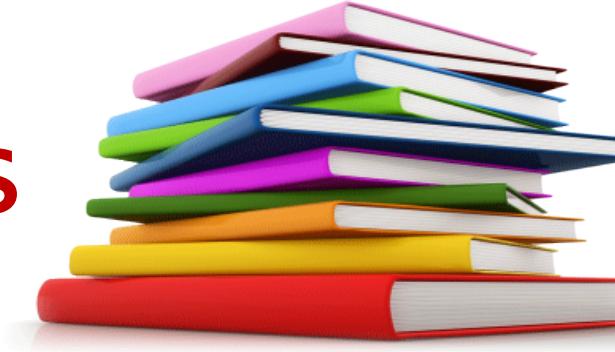
# why MERN?

- JavaScript Everywhere
  - The best part about MERN is that there is a **single language used everywhere**. (client-side, server-side, DB scripts)
  - having a **single language across tiers** also lets you share code between them
- JSON Everywhere
  - object representation is **JSON everywhere**: in the **database**, in the **application server**, and on **the client**
  - **No** object relational mapping (**ORM**)
  - **No** having to force fit an object model into rows and columns
- Node.js Performance

# why MERN? (Contd..)

- The npm Ecosystem
  - huge number of **npm packages** available freely for everyone to use
  - Any **problem** that you face will have an **npm package already**; you can fork it and make your own npm package
- It's not a Framework
  - framework asks you to fill in variations of what it thinks you want to get done.

# REFERENCES



Vasan Subramanian, “Pro MERN Stack”,  
Apress, 2017.





# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

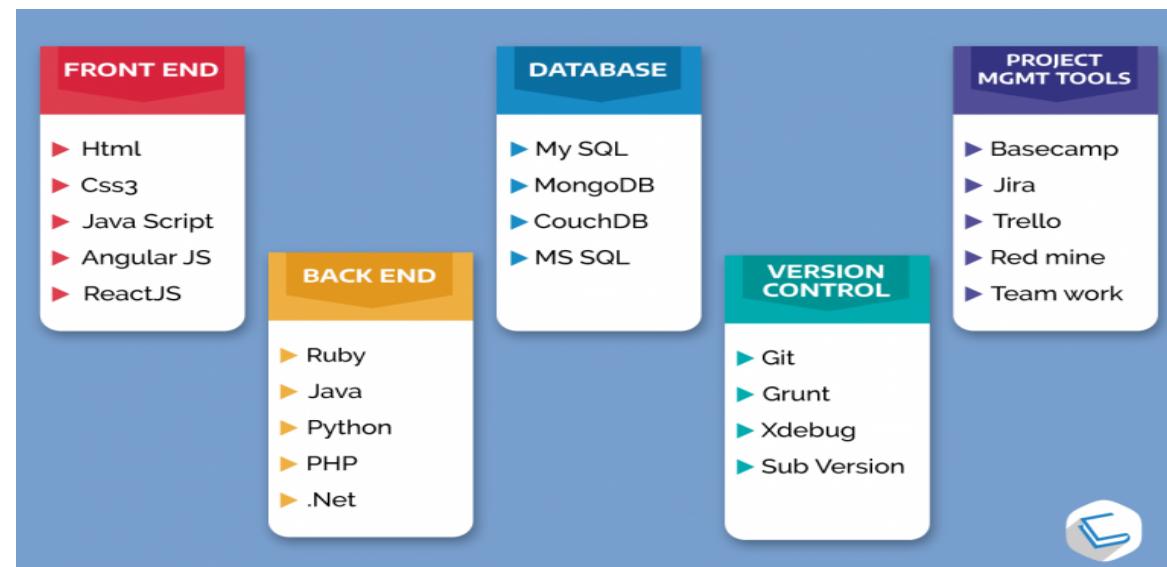
DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



# INT436 FULL STACK WEB APPLICATION DEVELOPMENT



- **Server-less Hello world**
- Server setup
  - nvm
  - Node.js
  - Project
  - npm
  - Express
- Build-Time JSX Compilation
  - Separate Script File
  - Transform
  - Automate; React Library
- ES2015

# Server-less Hello World

- We'll use React to render a simple page and use Node.js and Express to serve that page from a web server.
- Let's write a simple piece of code in a single HTML file that uses React to display a simple page on the browser.
- Open up your favorite editor and create an HTML file with a head and body

# Server-less Hello World(Contd..)

index.html: Server-less Hello World

```
<!DOCTYPE HTML>
<html>

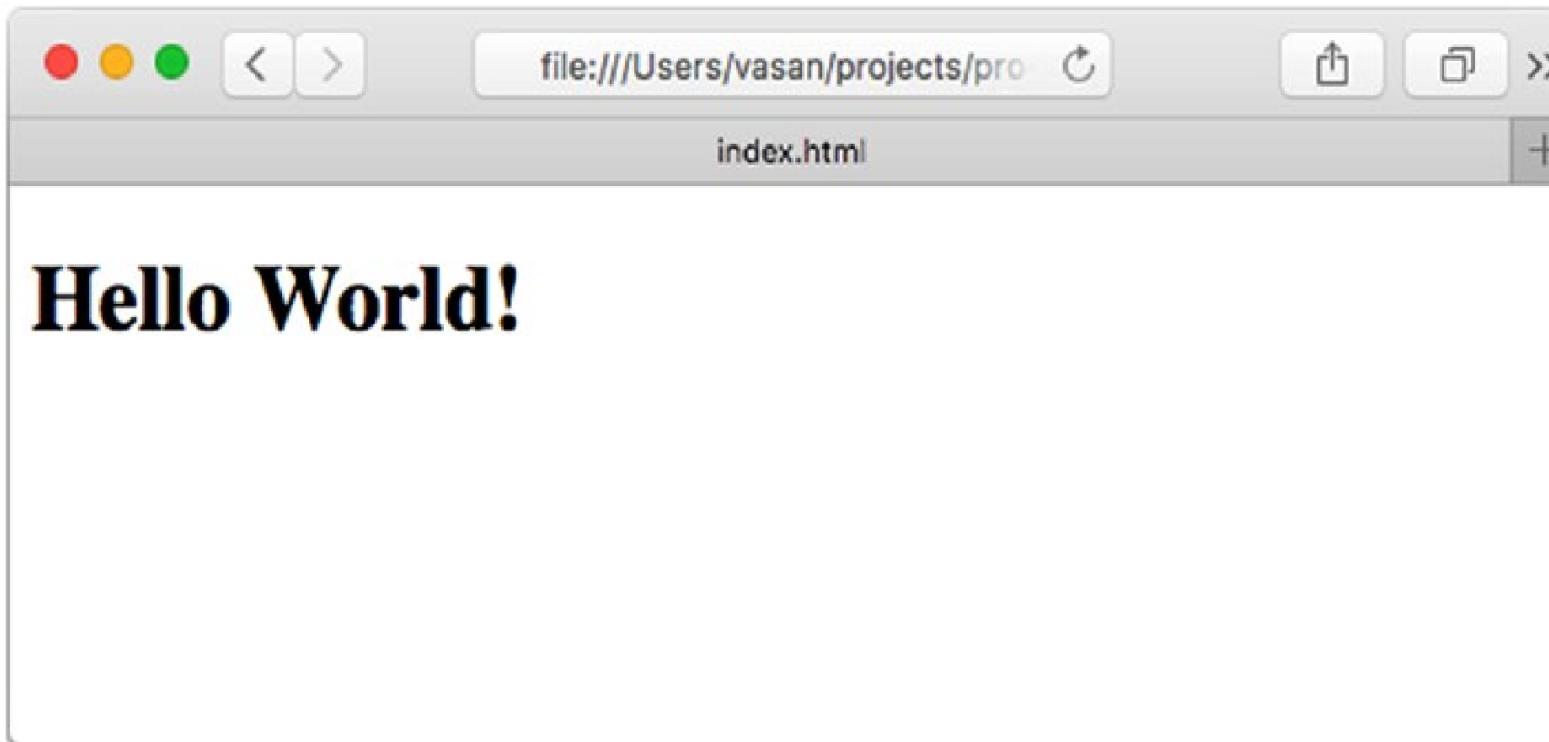
<head>
  <meta charset="UTF-8" />
  <title>Pro MERN Stack</title>
  <script src=
    "https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js">
  </script>
  <script src=
    "https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js">
  </script>
  <script src=
    "https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js">
  </script>
</head>
```

# Server-less Hello world(Contd..)

```
<body>
  <div id="contents"></div><!-- this is where our component will appear --&gt;
  &lt;script type="text/babel"&gt;
    var contentNode = document.getElementById('contents');
    var component = &lt;h1&gt;Hello World!&lt;/h1&gt;; // A simple JSX component
    ReactDOM.render(component, contentNode); // Render the component inside
  the content Node
  &lt;/script&gt;
&lt;/body&gt;

&lt;/html&gt;</pre>
```

# Server-less Hello world(Contd..)



*Hello World written in React*

# Server-less Hello World(Contd..)

- UTF-8
  - Unicode Transformation Format
- `<head>` – is a **container for metadata**
  - Placed between head and body tag
- The `<div>` tag defines a division or a section in an HTML document and used as a **container for HTML elements**
- `cdnjs.cloudflare.com` is an ultra-fast, reliable, globally available **Content Delivery Network(CDN)** for open-source libraries.

# Server-less Hello world(Contd..)

```
ReactDOM.render(element, container[, callback])
```

- **element:** The React element or component to render.
- **container:** The DOM element where the React content should be rendered.
- **callback** (optional): A function to execute once the rendering is complete.

# Server-less Hello World(Contd..)

- `ReactDOM.render(component, contentNode);`
  - asks the `ReactDOM` library to render the component within the content node
- In this case, the content node is the one with the ...

```
var component = <h1>Hello World!</h1>;
```

...
- The **component is not a string**; it's not enclosed in quotes; **It's not even valid JavaScript**.
- It is, instead, a special HTML-like language called **JSX**(**J**ava**S**cript **X**ML)
  - **JSX converts HTML tags into react elements in React's virtual DOM.**

# Server-less Hello World(Contd..)

- After transformation, this is what the code that is generated will look like:

```
...  
var component = React.createElement('h1', null, 'Hello World!');  
...
```

- This essentially creates a React <h1>element

```
React.createElement(  
  type,  
  [props],  
  [...children]  
)
```

## Parameters

- type**: A string representing the HTML tag name (e.g., 'div', 'span') or a React component.
- props**: An object containing the properties (props) for the element.
- children**: The child elements or content nested inside the element. These can be strings, numbers, React elements, or arrays of these types.

# Server-less Hello world(Contd..)

- How and when did the code get `const React.createElement`

```
...  
<script src=  
    "https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js">  
</script>  
...
```

- The babel library is a **browser-based JSX transformer**.
- **Note:** We will **stop using the browser-based transformer**

# Server-less Hello World(Contd..)

- The browser-based JSX compiler looks for all inline scripts of type “text/babel”

```
...  
  <script type="text/babel">  
    var contentNode = document.getElementById('contents');  
  ...
```

- Used to convert backwards compatible version of JavaScript in current and older browsers or environments.

- ECMAScript(ES)

- The newer version of JavaScript that was introduced in 2015 or 2016 (ES6)
- is a standard for scripting languages, including JavaScript, JScript, and ActionScript.
- to ensure the interoperability of web pages

# Server-less Hello world(Contd..)

- The other two scripts are the **core React libraries** that handle react component creation and rendering.

```
<script src=
  "https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js">
</script>
<script src=
  "https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js">
```

- Server-less Hello World
- **Server setup**
  - nvm
  - Node.js
  - Project
  - npm
  - Express
- Build-Time JSX Compilation
  - Separate Script File
  - Transform
  - Automate; React Library
- ES2015

# Server Setup

- The server-less setup allowed you to get familiarized with React **without any installations of a server.**
  - it's **good neither for development nor for production**
- Loading the scripts from a content delivery network (**CDN**)

# nvm

- This is the **Node Version Manager (NVM)** that makes installation and switching between **multiple versions of Node.js** easy
  - Follow the installation instruction
    - <https://github.com/creationix/nvm>
- 1) To **install or update nvm**, you should run the install script

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.4/install.sh | bash
```

[Notepad-Server setup instruction](#)

# Server.js

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

const app = express();
app.use(express.static('static'));

app.listen(3000, function () {
  console.log('App started on port 3000');
});
```

# Server.js

```
...  
const express = require('express');  
...
```

## Syntax

javascript

```
const module = require('module-name');
```

# Server.js

```
...  
const express = require('express');  
...
```

- ***require*** is a JavaScript keyword specific to Node.js, and it is used to **import other modules**
  - to include external modules that exist in separate files
- We loaded up the **module called express** and saved the module exports, in the constant named **express**.
- Node.js allows the thing to be a **function, an object, or whatever can fit into a variable**
- In the case of **Express**, the module **exports** a function that can be used to **instantiate an application**.

# Server.js (Contd..)

- Node.js supports ES2015 to a large extent

...

```
const app = express();
app.use(express.static('static'));
```

...

- This instantiates the application and then mounts a **middleware**.
- The middleware generator takes the parameter **static** to indicate that this is the **directory where all the static files reside**.
- The ***express.static*** generated **middleware function** is looking for **index.html** in the static directory.

- The **app.listen()** method in Express is used to **start the server and make it listen for incoming**

# Server.js (Contd..)

```
app.listen(port, [hostname], [backlog], [callback])
```

- **port**: The port number on which the server should listen.
  - **hostname** (optional): The hostname on which the server should listen. Defaults to `'localhost'`.
  - **backlog** (optional): The maximum number of pending connections. Defaults to the operating system's default.
  - **callback** (optional): A function to execute once the server is bound and listening.
- **\$ npm start**
  - **Output**
    - <http://localhost:3000>
    - (OR) <http://localhost:3000/index.html>

- Server-less Hello world
- Server setup
  - nvm
  - Node.js
  - Project
  - npm
  - Express
- **Build-Time JSX Compilation**
  - Separate Script File
  - Transform
  - Automate; React Library
- ES2015

# Build-Time JSX Compilation









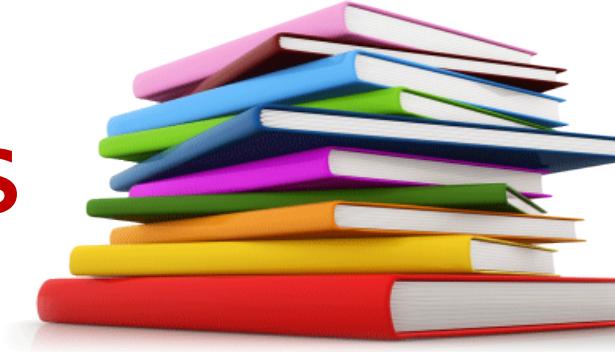








# REFERENCES



Vasan Subramanian, “Pro MERN Stack”,  
Apress, 2017.





# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

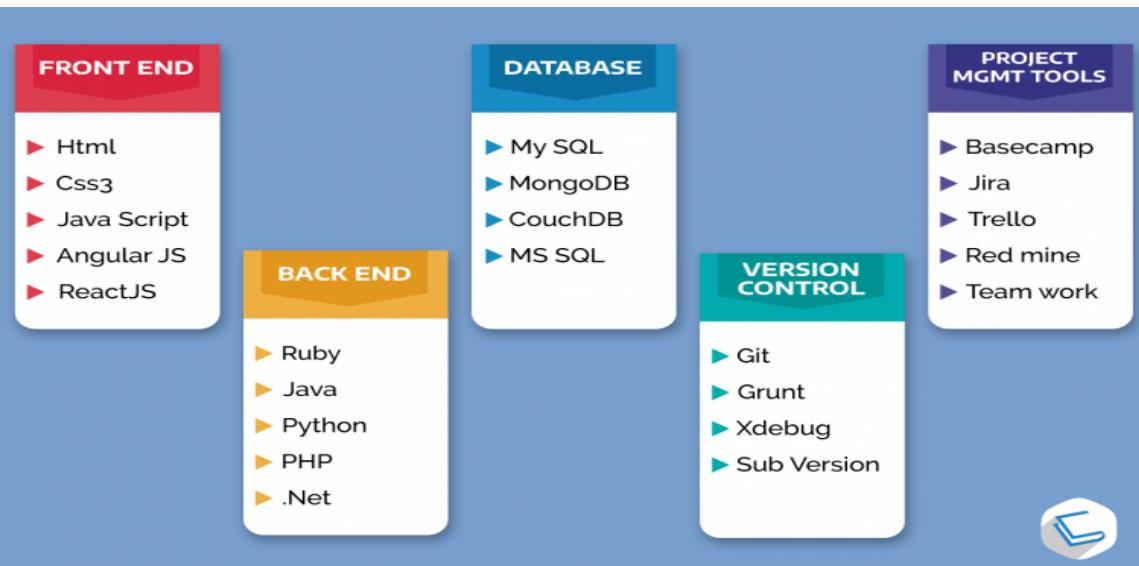
(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



# INT436

## FULL STACK WEB APPLICATION DEVELOPMENT





Ø Components

Ø what?

Ø Types

Ø Composing  
Components

Ø Life Cycle of  
Components

# What is React Component?



<https://www.simplilearn.com/tutorials/reactjs-tutorial/reactjs-components>

# What is React Component?

- *Components* are building blocks of any application created with React.
- Every application in react can be considered as a collection of components.
- A single app most often consists of many components.
- A component is a piece of the user interface.
- Components allow us split the UI into independent and reusable pieces.
- They accept arbitrary inputs (*called “props” - properties*) and return React elements describing what should appear on the screen.

# React Component

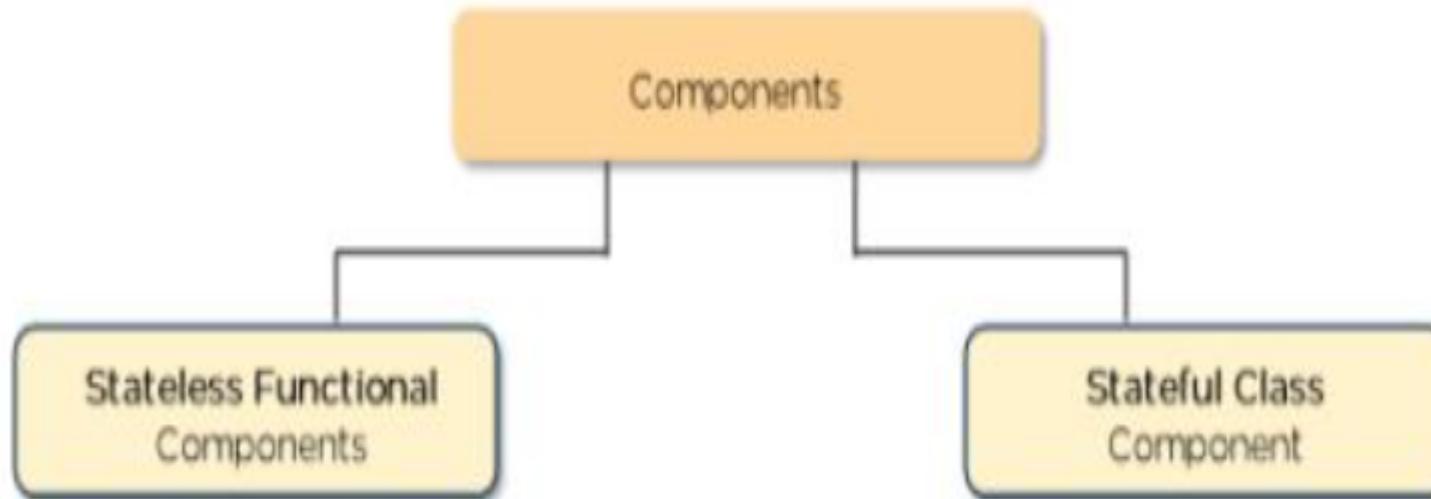
- **Re-usability:**
  - We can reuse a component used in one area of the application in another area.
  - This speeds up development and helps avoid cluttering of code.
- **Nested components:**
  - A component can contain within itself, several more components.
  - This helps in creating more complex design and interaction elements.

# React Component (Contd..)

- **Render method:**
  - In its minimal form, a component must define a render method specifying how it renders to the DOM.
- **Passing Props (Properties):**
  - A component can also receive props.
  - These are properties that its parent passes to specify particular values.

# React Component - Types

- There are two types of components in React



- **Stateless (Until React 16.8):** Before the introduction of React Hooks, function components were considered "stateless," meaning they couldn't manage their own state or lifecycle methods.

# Functional Component

- Functional components can be considered as simple JavaScript functions

Function components are simpler and more concise. They are essentially JavaScript functions that accept `props` as an argument and return React elements.

- Syntax**

```
function Componentname()  
{  
return <h1>welcome Text!</h1>;  
}
```

[FunctionalComp1.js](#)

- Component name can be any user-defined name and **should start with a capital letter**
- React treats components **starting with lowercase letters as DOM tags**
- Functional components only - when our component **does not need to interact with other components**.
- D:\EBOOKS\Full stack web application development\INT436\_July 2024\VSC\functioncomp*

# Class Component

- The functional components have no idea about other components present in the application.
- **Class components can communicate with other components.**
- Data can be transferred to other components using class components.
- Class components can be used whenever state management is required in react.
- **Creating a class component requires us to create javascript classes.**

# Class Component

- Syntax

```
class Componentname extends React.Component
{
  render()
  {
    return <h1>welcome Text!</h1>;
  }
}
```

[classComp1.js](#)

- To define a React component class, you need to extend `React.Component`
- The **only method** you *must* define in a `React.Component` subclass is called `render()`

# Class Component

- Example

```
class Car extends React.Component
{
  render()
{
  return <h2>Hi, I am a Car!</h2>;
}
}
```

Now your React application has a component called car, which returns a `<h2>` element

# Class Component

- Add constructor() function – Component
- Initialize components properties
- In React, component properties should be kept in an object called - state

[ClassComp2.js](#)

```
class Car extends React.Component {  
  constructor() {  
    super();  
    this.state = {color: "red"};  
  }  
  render() {  
    return <h2>I am a  
{this.state.color} car!</h2>;  
  }  
}
```

# Props

- Add component properties – props
- **Props are like function arguments**, and you send them into the component as attributes.
- Use an attribute to pass a color to the Car component, and use it in the render() function

[\*classComp3.js\*](#)

```
class Car extends React.Component {  
  render() {  
    return <h2>I am a  
    {this.props.color} Car!</h2>;  
  }  
}  
  
var mount =  
document.getElementById('app');  
ReactDOM.render(<Car color="green"/>,  
mount);
```

# Props in the Constructor

- If your component has a constructor function,
  - the props should always be passed to the **constructor** and also to the **React.Component** via the **super()** method.

```
class Car extends React.Component {  
  constructor(props) {  
    super(props);  
  }  
  render() {  
    return <h2>I am a  
{this.props.model}</h2>;  
  }  
}  
  
var mount =  
document.getElementById('app');  
ReactDOM.render(<Car model="Mustang"  
/>mount);
```

[classComp4.js](#)

# Composing Components

- It's possible to build a component that uses other user-defined components as well.
- This is called **component composition**, and it is one of the most powerful features of React.

```
class Car extends React.Component {  
  render() {  
    return <h1>Hi, I am a Car!</h1>;  
  }  
}  
  
class Bike extends React.Component {  
  render() {  
    return <h1>Hi, I am a Bike!</h1>;  
  }  
}
```

# Composing Components

```
class Vehicle extends React.Component {  
  render() {  
    return (  
      <div>  
        <h1>vehicle List</h1>;  
        <Car />  
        <hr />  
        <Bike />  
      </div>  
    );  
  }  
}  
  
var mount =  
document.getElementById('app');  
ReactDOM.render(<Vehicle />, mount);
```

[ComposingComp.js](#)

# Lifecycle of Components

- Each component in React has a lifecycle
- The three phases are: **Mounting**, **Updating**, and **Unmounting**.
- **Mounting**
- **Mounting** means putting elements into the DOM.
- React has four built-in methods that gets called, in this order, when mounting a component:
  - constructor()
  - getDerivedStateFromProps()
  - render()
  - componentDidMount()
- The render() method is required and will always be called, the others are optional and will be called if you define them.

# LifeCycle of Components

- **Unmounting.**
- The `componentWillUnmount()` method allows us to execute the React code **when the component gets destroyed** or unmounted from the DOM (Document Object Model).
- This method is called during the Unmounting phase of the React Life-cycle i.e before the component gets unmounted.
- All the ***cleanups*** such as invalidating timers, canceling network requests should be coded in the `componentWillUnmount()` method block.

# Lifecycle of Components

- Updating
- The next phase in the lifecycle is when a component is updated.
- A component is updated whenever there is a change in the component's state or props.
- React has five built-in methods that gets called, in this order, when a component is updated:
  - `getDerivedStateFromProps()`
  - `shouldComponentUpdate()`
  - `render()`
  - `getSnapshotBeforeUpdate()`
  - `componentDidUpdate()`

# References

- <https://www.simplilearn.com/tutorials/reactjs-tutorial/reactjs-components>
- [https://www.w3schools.com/react/react\\_class.asp](https://www.w3schools.com/react/react_class.asp)
- <https://www.geeksforgeeks.org/reactjs-componentwillunmount-method/>





# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

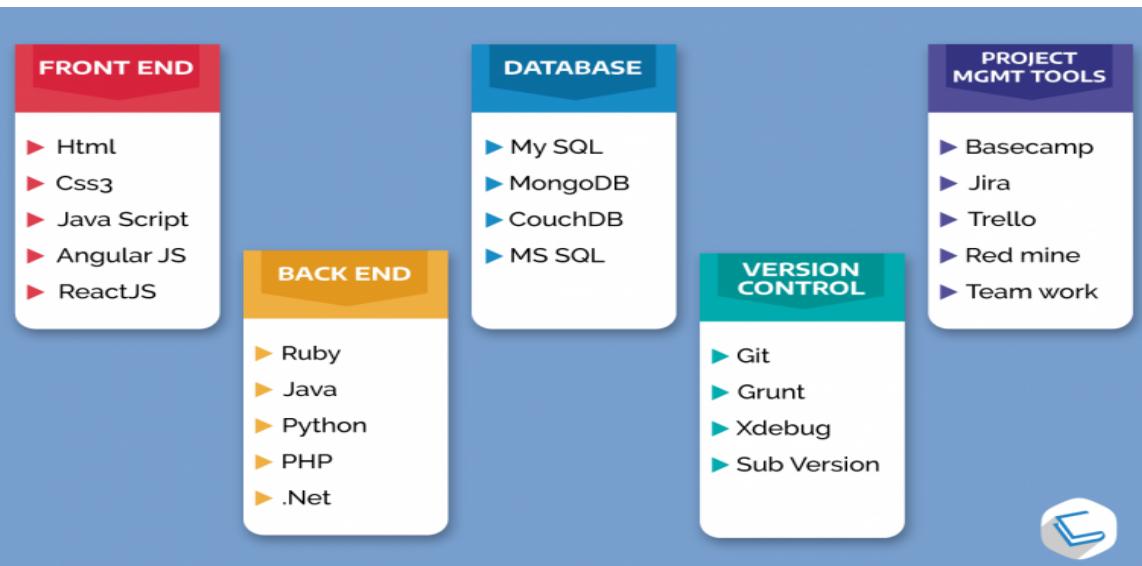
(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



# INT436

## FULL STACK WEB APPLICATION DEVELOPMENT





- Ø Passing Data
- Ø Using Properties
- Ø Using Children
- Ø Dynamic Composition
- Ø React Fragments

# Using Properties

- Need - to pass data from a parent component to a child component and make it render differently on different instances
- You can pass data from a parent to a child component in different ways.
- One way to do this is using properties.
- Any data passed in from the parent can be accessed in the child component through a special variable, *this.props*.

# Recap

```
<body>  
<div id="app"></div> <!-- this is where our component  
will appear -->  
  
<script type="text/babel">  
  
class Car extends React.Component {  
  render() {  
    return <h1>Hi, I am a {this.props.color}  
Car!</h1>;  
  }  
}  
  
var mount = document.getElementById('app');  
ReactDOM.render(<Car color="red" />, mount);  
</script>
```

[classComp3.js](#)

# Props in a function component

- In a function component,
  - components receive **props** exactly like an ordinary function argument.
- A function component will receive the props object with properties you described in the component call.

src > JS App.js > default

```
1 import Greetings from './Greetings';
2
3 function App() {
4   return (
5     <div className="App">
6       <Greetings name="ABCDE" age={22} occupation="Student" />
7     </div>
8   );
9 }
10 export default App;
```

*Greetings.js*

```
src > JS Greetings.js > Greetings
1 import React from 'react'
2
3 function Greetings(props) {
4   return (
5     <div>
6       <p> Hello! I'm {props.name}, a {props.age} years old {props.occupation}.
7       Pleased to meet you! </p>
8     </div>
9   );
10 }
11
12 export default Greetings;
```

# Props in a function component

- Aside from passing multiple props at once, in this example, you also see the age prop is a *number data type*.
- This demonstrates that you can pass any type of data available in JavaScript—such as number, Boolean, or object—into props.
- This is how props enable you to send data using the top-down approach, wherein a component at a higher level can send data to a component below it.

# Code reuse with props

- The use of props allows you to reuse React code.
- In the case of our example, you can reuse the same Greeting component for many different people:

```
function App() {
  return (
    <div>
      <Greeting name="Jack" age={27} occupation="Software
Developer" />
      <Greeting name="Jill" age={24} occupation="Frontend
Developer" />
    </div>
  );
}

function Greeting(props) {
  return (
    <p>
      Hello! I'm {props.name}, a {props.age} years old
{props.occupation}.
      Pleased to meet you!
    </p>
  );
}
```

Greetings.js

# Limitation – props

- Since props are read-only and must not be changed manually throughout the lifespan of a React application
- Using only props in your React app
  - doesn't really make it a dynamic app that can respond to user interactions and render accordingly.
- In order to do that, you need to use state.

# Output

?

- By mistake you forget to pass a required prop into the component that needs it.

```
function App() {  
  return <Greeting name="ABCDE" />;  
}
```

```
function Greeting(props) {  
  return (  
    <p>  
      Hello! I'm {props.name}, a {props.age} years old  
      {props.occupation}.  
      Pleased to meet you!  
    </p>  
  );  
}
```

Greetings.js

- while `props.age` and `props.occupation` are undefined in the `Greeting` component,
- React will simply ignore the expression to call on their value and render the rest of the text.
- It doesn't trigger any error, but you know you can't let this kind of thing go unaddressed.

# propType S

- This is where propTypes comes to help.
- PropTypes is a special component property that can be used to validate the props you have in a component.
- It's a separate, optional npm package, so you need to install it first before using it.
- `npm install --save prop-types`

```
Greeting.propTypes = {  
  name: PropTypes.string.isRequired, // must be a  
string and defined  
  age: PropTypes.number.isRequired, // must be a  
number and defined  
  occupation: PropTypes.string.isRequired // must be a  
string and defined  
};
```

Propsdemo.js

- with the propTypes property declared,
  - the Greeting component will throw a warning to the console
  - when its props aren't passing propTypes validation.

# defaultProps

- You can also define default values for props in cases where props are not being passed into the component on call by using another special property called `defaultProps`.

```
Greeting.defaultProps = {  
  name: "SASTRA",  
  age: 35,  
  occupation: "Teacher"  
};
```

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

# Using Children

- There is another way to pass data to other **components**, using the contents of the HTML like node of the component.
- In the child component,
  - this can be accessed using a special property of `this.props` called `this.props.children`.
- React **children** is also a react prop which is declared in react by default.
- This prop contains the content which is written in between the **opening (`<>`)** and **closing (`</>`)** tags where the component was called.

# Using Children - Example

- Consider an example in which we are using a component 'Card' inside the App.js file of the React Application.

```
import React from "react";
import Card from "./Card.jsx";

function App() {
  return (
    <div className="App">
      <Card>
        This is the content for the react children which
        can be accessed from
        the Card component (Card.jsx)
      </Card>
    </div>
  );
}

}
```

# Using Children - Example

```
import React from "react";

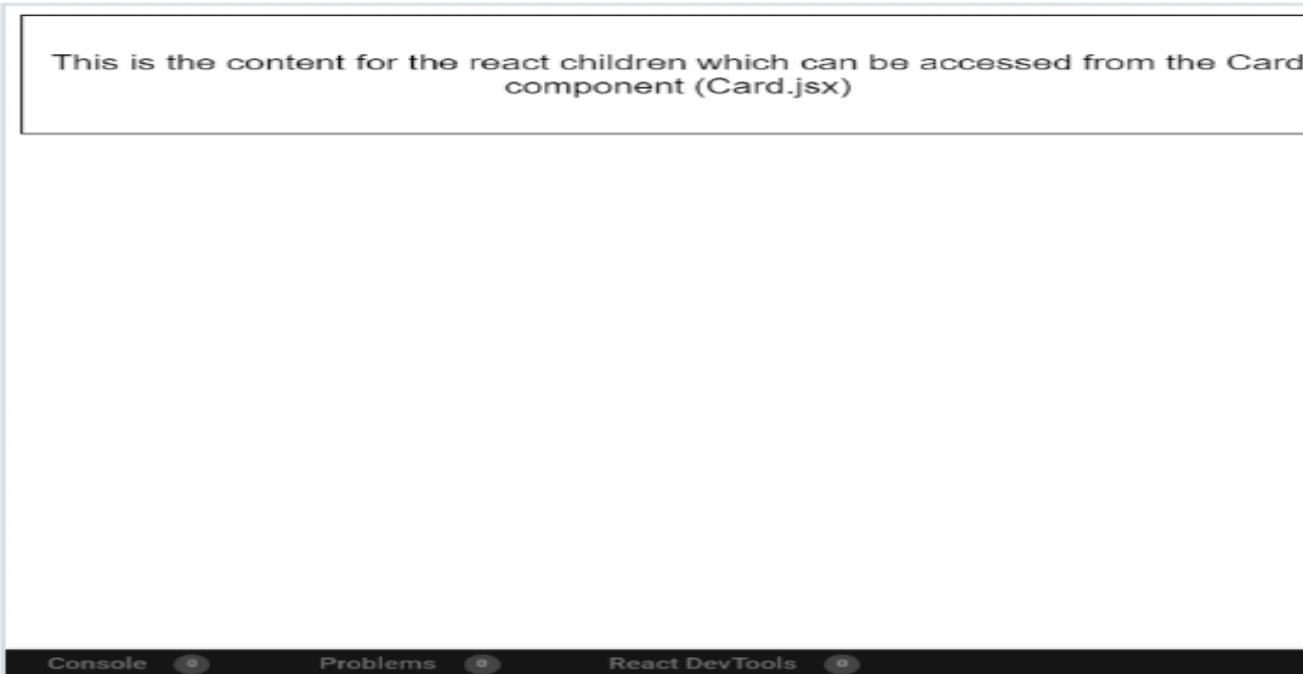
function Card(props)
{
  return (
    <div className="paragraph">
      <p>{props.children}</p>
    </div>
  );
}
```

[Card.js](#)

we can access children by using  
props.children

# Output

- This is what our App.js page looks like.
- we can see the content that is written inside Card tags in App.js has reached the Card.jsx component through the prop 'children'.



This is the content for the react children which can be accessed from the Card component (Card.jsx)

```
Console Problems React DevTools
```

# Dynamic Composition

- we'll replace the `components` content with a programmatically generated set.
- we'll use a simple `JavaScript array` in memory to store the list of contents to be displayed.
- Later chapters,
  - we'll get more sophisticated by getting the data from the server, and then from a database.

# Dynamic Composition

- Rendering Multiple Elements

- `const reptiles = ["alligator", "snake", "lizard"];`

- Return a `<li>` element for each item in the array.

```
function ReptileListItems() {  
  const reptiles = ["alligator",  
"snake", "lizard"];  
  
  return reptiles.map((reptile) =>  
    <li>{reptile}</li>);  
}
```

## Output

- alligator
- snake
- lizard

# Review Question

```
<html>
<script
src="https://unpkg.com/@babel/standalone/babel.js"></script>
<script
src="https://unpkg.com/react/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom/umd/react-dom.development.js"></script>
<div id="root"/>
<script type="text/babel">
  function Example() {
    return (
      <h1>SASTRA University</h1>
      <h2>School of Computing</h2>
    );
  }
  ReactDOM.render(<Example />,
document.getElementById('root'));
</script>
```

# Solution

```
<html>
<script
src="https://unpkg.com/@babel/standalone/babel.js"></script>
<script
src="https://unpkg.com/react/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom/umd/react-dom.development.js"></script>
<div id="root"/>
<script type="text/babel">
  function Example() {
    return (
      <div>
        <h1>SASTRA University</h1>
        <h2>School of Computing</h2>
      </div>
    );
  }
  ReactDOM.render(<Example />, INT436 - FSWAD
document.getElementById('root'));
  
```

# div - issue

```
function Column () {  
    return (  
        <div>  
            <td>Hello  
world!</td>  
            <td>Hello  
SASTRA!</td>  
        </div>  
    );  
}
```

```
function Table () {  
    return (  
        <table>  
            <tr>  
                <Column />  
            </tr>  
        </table>  
    );  
}
```

```
<table>  
    <tr>  
        <div>  
            <td>Hello  
world!</td>  
            <td>Hello  
SASTRA!</td>  
        </div>  
    </tr>  
</table>
```

If we wrap the `<td>` elements in a `<div>`, this will add a `<div>` element in middle of `<tr>` and `<td>` and break the parent-child relationship.

# React Fragments

- React Fragments allow you to wrap multiple elements without adding an extra DOM element.
- Replace the `<div>` warpper with the Fragment element in the `<Column />` component

```
function Column () {  
  return (  
    <React.Fragment>  
      <td>Hello  
world!</td>  
      <td>Hello  
SASTRA!</td>  
    </React.Fragment>  
  );  
}
```

```
function Column () {  
  return (  
    <>  
      <td>Hello  
world!</td>  
      <td>Hello  
SASTRA!</td>  
    </>  
  );  
}
```

# References

- <https://blog.logrocket.com/the-beginners-guide-to-mastering-react-props-3f6f01fd7099/>
- <https://sebhastian.com/react-props-vs-state/>
- <https://www.scaler.com/topics/react/react-children/>
- <https://www.freecodecamp.org/news/javascript-map-how-to-use-the-js-map-function-array-method/>
- <https://www.freecodecamp.org/news/arrow-function-javascript-tutorial-how-to-declare-a-js-function-with-the-new-es6-syntax/>

Thank  
you



# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



# INT436

## FULL STACK WEB APPLICATION DEVELOPMENT





Ø React State

Ø Creating the state  
Object

Ø Using the state  
Object

Ø Changing the state  
Object

Ø setTimeout()

# Review Question

```
class Statedemo extends Component
{
    render()
    {
    }
}
```

```
class Statedemo extends Component
{
    render()
    {
        return null;
    }
}
```

*Statedemo1.js*

ERROR

[eslint]  
src\Statedemo.js  
Line 4:5: Your render method should have a return  
statement react/react-render-return

D:\EBOOKS\Full stack web application

development\INT436\_July

2024\vscode\U2\_demo1\src\Statedemo1.js

INT436\_FSWAD

# React State

- Until now, you've only seen **static components**.
- To make components that respond to user input and other events, React uses a **data structure called state** in the component.
- A **state** is an object that holds data and information related to a React component.
- It can be used to store, manage, and update data within the application, which in turn allows for **dynamic changes** to user interfaces.
- For example, if a **button needs to change its text based on user input**, then this would be done by updating the state with new values.

# React State

- React treats the component as a **simple state machine**.
- **whenever the state changes**,
  - it triggers a **rerender** of the component and the view automatically changes.
- **The way to inform React of a state change is by using the `setState()` method.**
- This method takes in an object, and the top-level properties are **merged** into the existing state.
- within the component, you can access the properties via the **`this.state` variable**.
- The **initialization of the state is done in the constructor**

# Creating the state Object

- The state object is initialized in the **constructor**

```
class Car extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      brand: "Ford",  
      color: "red",  
    };  
  }  
  render() {  
    return (  
      <div>  
        <h1>My Car</h1>  
      </div>  
    );  
  }  
}
```

# Using the state Object

- Refer to the state object anywhere in the component by using the **this.state.propertyname** syntax.

```
class Car extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      brand: "Ford",  
      color: "red",  
    };  
  }  
}
```

*Car.js*

```
render() {  
  return (  
    <div>  
      <h1>My  
      {this.state.brand}</h1>  
      <p>  
        It is a  
        {this.state.color} color  
        car.  
      </p>  
    </div>  
  );  
}  
INT436 – FSWAD
```

Demo

# Changing the state object

*Counter.js*

- To change a value in the state object, use the `this.setState()` method.
- When a value in the **state object changes**, the **component will re-render**, meaning that the output will change according to the new value(s).

# Do Not Modify State Directly

- `this.state.comment = 'Hello'; //wrong`
- `this.setState({comment: 'Hello'})`;
- The only place where you can assign `this.state` is the constructor.

# State Updates May Be Asynchronous

- React may batch **multiple setState() calls** into a single update for performance.
- Because **this.props** and **this.state** may be updated **asynchronously**, you should not rely on their values for calculating the next state.
- For example, this code may fail to update the counter:

*Counter2.js*

```
this.setState({  
  counter: this.state.counter +  
  this.props.increment,  
});
```

# State Updates May Be Asynchronous

- To fix it, use a second form of `setState()` that accepts a function rather than an object.
- That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument:

```
this.setState(function(state,  
props) {  
  return {  
    counter: state.counter +  
    props.increment  
  };  
});
```

# State Updates are Merged

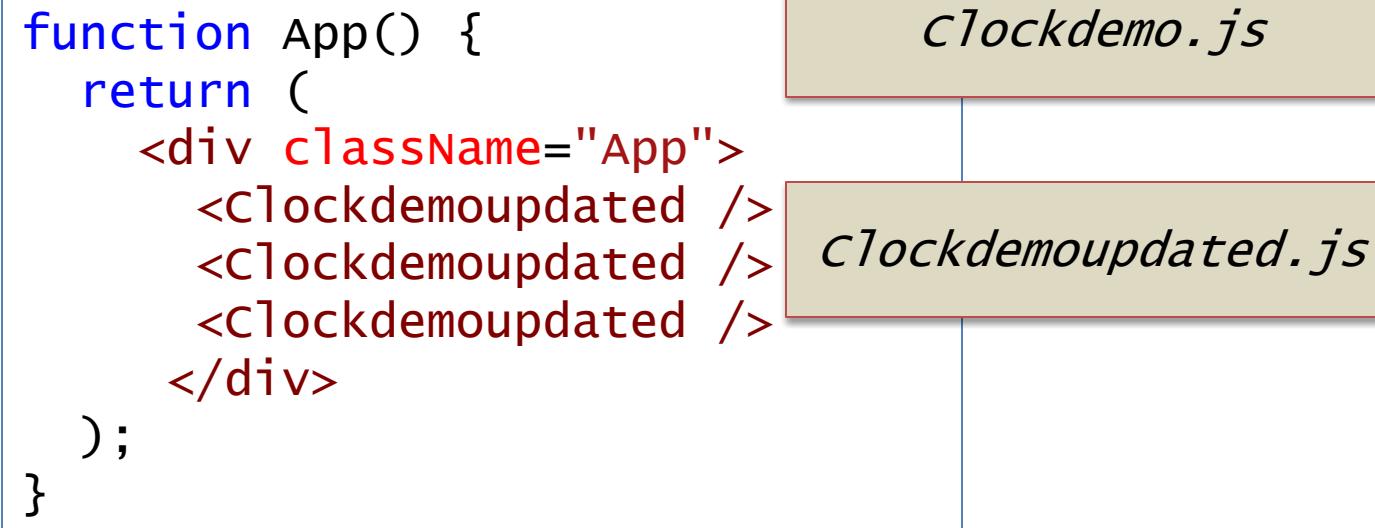
- When you call `setState()`
  - React merges the object you provide into the current state.
- In the example (next slide)
  - we're updating the variable **dogNeedsVaccination** independently of the other state variables.
- The merging is shallow,
  - so `this.setState({ dogNeedsVaccination: true })`
  - leaves the other variables intact,
  - replacing only the value of `dogNeedsVaccination`.

# State Updates are Merged

```
state = {  
    dogId: 991,  
    dogName: 'Pluto',  
    dogAge: 8,  
    dogIsMicroChipped: true,  
    dogLastVaccineDate: 1538784000,  
    dogNeedsVaccination: false,  
}  
  
this.setState({  
    dogNeedsVaccination: true,  
});
```

# Changing the state object

- To show that **all components are truly isolated**
- We can create an App component that renders three `<clock>`s:



to

- The **setTimeout()** method sets a timer which executes a function or specified piece of code once the timer expires.

## Syntax:

- setTimeout(functionRef)
- setTimeout(functionRef, delay)
- setTimeout(functionRef, delay, param1)

*Timeoutdemo.js*

## Parameters:

- **functionRef** - A function to be executed after the timer expires.
- **delay** - The time, in milliseconds that the timer should wait before the specified function or code is executed.
- If this parameter is omitted, a value of 0 is used, meaning execute "immediately", or more accurately, the next event cycle.

# setTimeout()

- param<sub>1</sub>, ..., param<sub>N</sub> Optional
  - Additional arguments which are passed through to the function specified by functionRef.
- **Return value**
- The returned **timeoutID** is a positive integer value which identifies the timer created by the call to **setTimeout()**.
- This value can be passed to **clearTimeout()** to cancel the timeout.

# setTimeout()

- `setTimeout(() => {`
- `console.log("Delayed for 1 second.");`
- `}, "1 second");`

# setTimeout

to

- If setTimeout() is called with delay value that's not a number, **implicit type coercion** is silently done on the value to convert it to a number.
- `setTimeout(() => {  
 console.log("Delayed for 1 second.");  
}, "1000");`
- `setTimeout(() => {  
 console.log("Delayed for 1 second.");  
}, "1 second");`
- Don't use strings for the delay value but instead always use numbers:

# setTimeout()

```
setTimeout(() => {  
    console.log("this is the first  
message");  
, 5000);
```

```
setTimeout(() => {  
    console.log("this is the second  
message");  
, 3000);
```

```
setTimeout(() => {  
    console.log("this is the third  
message");  
, 1000);
```

## Output:

this is the third  
message  
this is the second  
message  
this is the first  
message

Note: setTimeout() is  
asynchronous; three  
different timerID  
will be created

# setTimeout()

- The `setTimeout()` is executed only once.
- If you need repeated executions, use `setInterval()` instead.
- Use the `clearTimeout()` method to prevent the function from starting.
- To clear a timeout, use the id returned from `setTimeout()`.

*clocksetInterval.js*

- <https://legacy.reactjs.org/docs/state-and-lifecycle.html>
- <https://www.codingame.com/playgrounds/8747/react-lifecycle-methods-render-and-componentdidmount>
- <https://developer.mozilla.org/en-US/docs/Web/API/setTimeout>
- [https://www.w3schools.com/jsref/met\\_win\\_settimeout.asp](https://www.w3schools.com/jsref/met_win_settimeout.asp)
- <https://www.mygreatlearning.com/reactjs/tutorials/reactjs-state>
- <https://www.educative.io/answers/what-is-setstate-in-react>





# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

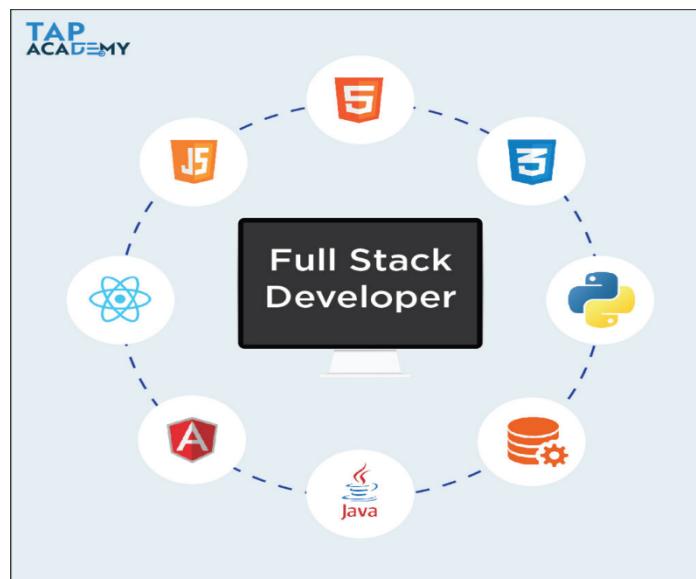
(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



# INT436

## FULL STACK WEB APPLICATION DEVELOPMENT





- Ø Event Handling
- Ø Difference Between HTML And React Event Handling
- Ø Synthetic Event
- Ø Adding event handlers

# Event Handling in React

- Handling events in React is crucial for building interactive and dynamic user interfaces.
- React components are designed to respond to user input,
  - such as clicks, keyboard input, and form submissions,
  - by updating their state or triggering changes in other components.
- By handling events properly, developers can create robust and responsive applications that provide a smooth user experience.

# Understanding Events

## in React

- Events in React refer to actions that happen in the user interface, such as a mouse click, a keyboard key press, or a form submission.
- In React,
  - events are handled by defining event handlers,*
  - which are functions that are executed in response to a specific event.
- Event handling in React is guided by a few basic principles that align with its component-based architecture.
- These principles include:
  - Synthetic event system
  - Naming conventions
  - Passing event handlers as props
  - Inline function and component methods

# synthetic Event

- How React deals with events first?
- React listens to every event at the document level, after receiving an event from the browser, **React wraps this event with a wrapper** that has the same interface as the native browser event
- So what is the need for this wrapper?
- Think of a situation where the **exact same event has different names across different browsers.**
- Imagine an event that fires when the user winks,
  - this event in chrome called A
  - In Safari called B,
  - in such case, we will need to make **different implementations** for each browser

# Syntetic Event

- what this wrapper does is
    - *registering all the different names for the same event effect,*
    - **winking** in our case,
    - with **only one name**,
    - so in a case when we want to listen to our **winking effect** instead of being listening to A for chrome and B for Safari we just use **onwink**, which is the wrapper react creates around the real event.
  - So whenever we are triggering an event in a React component,
  - we are not actually dealing with the real DOM event,
  - instead, *we are dealing with React's custom event type, a synthetic event.*

# Understanding Events in React

- **Naming conventions**
  - revolve around a set of consistent naming that developers use for **identifying events** and the **handler functions** at a glance.
  - Every event uses a *camelCase* naming convention, and the handler function they run is *prefixed with "handle"*, followed by the event name.
  - For example, an **onClick** event running a **handleClick** function.

# Understanding Events in React

- **Passing event handlers as props**

- Event handlers are the **functions** that run when the event is fired.
- They're usually **defined** before the **render**, just above the return statement.
- On many occasions, they are also **passed as props** to components.
- This aligns with **React component-based architecture**, allowing event logic to be embedded within the components using them.

# Understanding Events in React

- **Inline function and component methods**
  - In React components,
    - events typically run inline functions or
    - standalone functions within the component when fired.
  - with `this`, you can utilize hooks like `useState` for state and `useCallback` for memoizing handler functions.
  - This helps manage state changes and optimize performance.

# Types of Events in React

- React supports a variety of event types, including:
  - Clipboard Events
  - Composition Events
  - Keyboard Events
  - Focus Events
  - Form Events
  - Mouse Events
  - Pointer Events
  - Selection Events
  - Touch Events
- Each of these events in **React** has a set of associated **event handlers**.
- For instance, the **onClick** event handler is used to handle click events, the **onChange** event handler is used to handle change events, and so on.

# What is an event handler?

- An event handler is a function responsible for handling events.
- Using event handlers, we can define custom behavior that should occur when an event is initiated.
- In React, event handlers are attached to specific components using event props that are named after DOM elements.

# Adding event handlers

Create an App component that renders a button when clicked, does nothing.

```
export default function App() {  
  return (  
    <button>  
      I don't do anything  
    </button>  
  );  
}
```

# what is an event handler?

```
import React from 'react';

function Button() {
  const handleclick = () => {
    alert('Hello world!');  };

  return ( <button onclick={handleclick}>Click
me</button> );
}
```

*Eventdemo.js*

# Props with event handlers

- As event handlers are **functions** that are declared inside a React component, they have **access to the props associated with the component**.

```
function AlertButton({message}) {  
  return (  
    <button onclick={() => alert(message)}>  
      Click Me  
    </button>  
  );  
}  
  
export default function App() {  
  return (  
    <AlertButton message = "Hello World"/>  
  );  
}
```

*AlertButton.js*

# Difference Between HTML And React Event Handling

- Handling events with React elements is very similar to handling events on DOM elements.
- There are some syntax differences:
  - React uses **camelCase** for event names, while HTML uses lowercase.
  - Instead of **passing a string as an event handler**, we pass a **function** in React.

*Texteventdemo.js*

*Keyeventdemo.js*

*MouseEventDemo.js*

```
<button  
onClick="activateLasers()">  
    Activate Lasers  
</button>
```

14/08/2024

11:21:08 am

```
<button  
onClick={activateLasers}>  
    Activate Lasers  
</button>
```

INT436 – FSWAD

48

4

# Difference Between HTML And React Event Handling

- Additionally, much like in HTML, we cannot **return false** to override default behavior; instead, we must use **preventDefault**
- In HTML
- `<form onsubmit="console.log('clicked'); return false">`
- `<button type="submit">Submit</button>`
- `</form>`

# Difference Between HTML And React Event Handling

- In React, this could instead be:

```
function Form() {  
  function handleSubmit(e) {  
    e.preventDefault();  
    console.log('You clicked submit.');//  
  }  
  
  return (  
    <form onSubmit={handleSubmit}>  
      <button type="submit">Submit</button>  
    </form>  
  );  
}
```

# Best Practices for Efficient Event Handling in React

- Avoid Using Anonymous Arrow Functions Inside Events
  - It looks convenient to use arrow functions directly in events, like `onClick={() => console.log('button clicked')}`.
  - The downside to this is that it can lead to performance issues because a new function is created on every render.
  - Always define the handler function to run when the event is fired outside the render method to avoid those performance issues.

# BEST PRACTICES FOR Efficient Event Handling in React

- Prevent Default Behavior where Necessary
  - Use `event.preventDefault()` in your event handlers when you need to stop the browser from performing default actions, like submitting a form.
- Clean Up Event Listeners
  - If you set up your event listeners in `useEffect`, always return a `cleanup` function to remove the event listener.
  - otherwise, it'll cause memory leaks.
- Memoize Events with `useCallback` hook
  - For components that re-render often, memoizing the handlers in it with the `useCallback` hook can prevent unnecessary re-renders.
  - This is useful when passing events as props to child components that might re-render unnecessarily.

# References

- <https://www.scaler.com/topics/react/event-handling-in-react/>
- <https://www.educative.io/answers/event-handler-in-reactjs>
- <https://www.mygreatlearning.com/react-js/tutorials/reactjs-state>
- <https://dev.to/nagwan/react-synthetic-events-34e5>

# Reference S

- <https://www.dhiwise.com/post/reacting-to-user-actions-a-deep-dive-into-handling-events-in-react>
- <https://www.almabetter.com/bytes/tutorials/reactjs/event-handling-in-reactjs>
- <https://www.freecodecamp.org/news/how-to-handle-events-in-react-19/>





# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act, 1956)

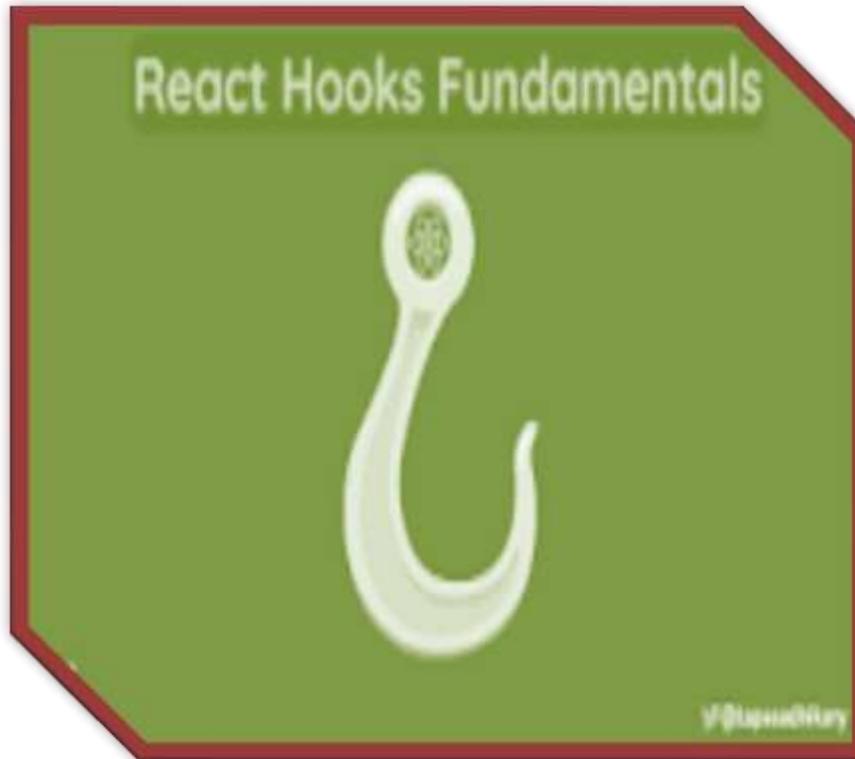
THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



## INT436

# FULL STACK WEB APPLICATION DEVELOPMENT





# Review Question

```
class Car extends React.Component
{
  render()
{
  return <h2>Hi, I am a Car!</h2>;
}
}
```

```
import { PureComponent } from 'react';

class Car extends PureComponent
{
  render()
{
  return <h2>Hi, I am a Car!</h2>;
}
}
```

# PureComponent

- To skip re-rendering a class component for same props and state, extend **PureComponent** instead of **Component**
- **PureComponent** is a subclass of **Component** and supports all the **Component APIs**.
- React normally re-renders a component whenever its parent re-renders.
- As an optimization, you can **create a component** that React will not re-render when its parent re-renders so long as its new props and state are the same as the old props and state.
- Class components can opt into this behavior by extending **PureComponent**.

# Introduction

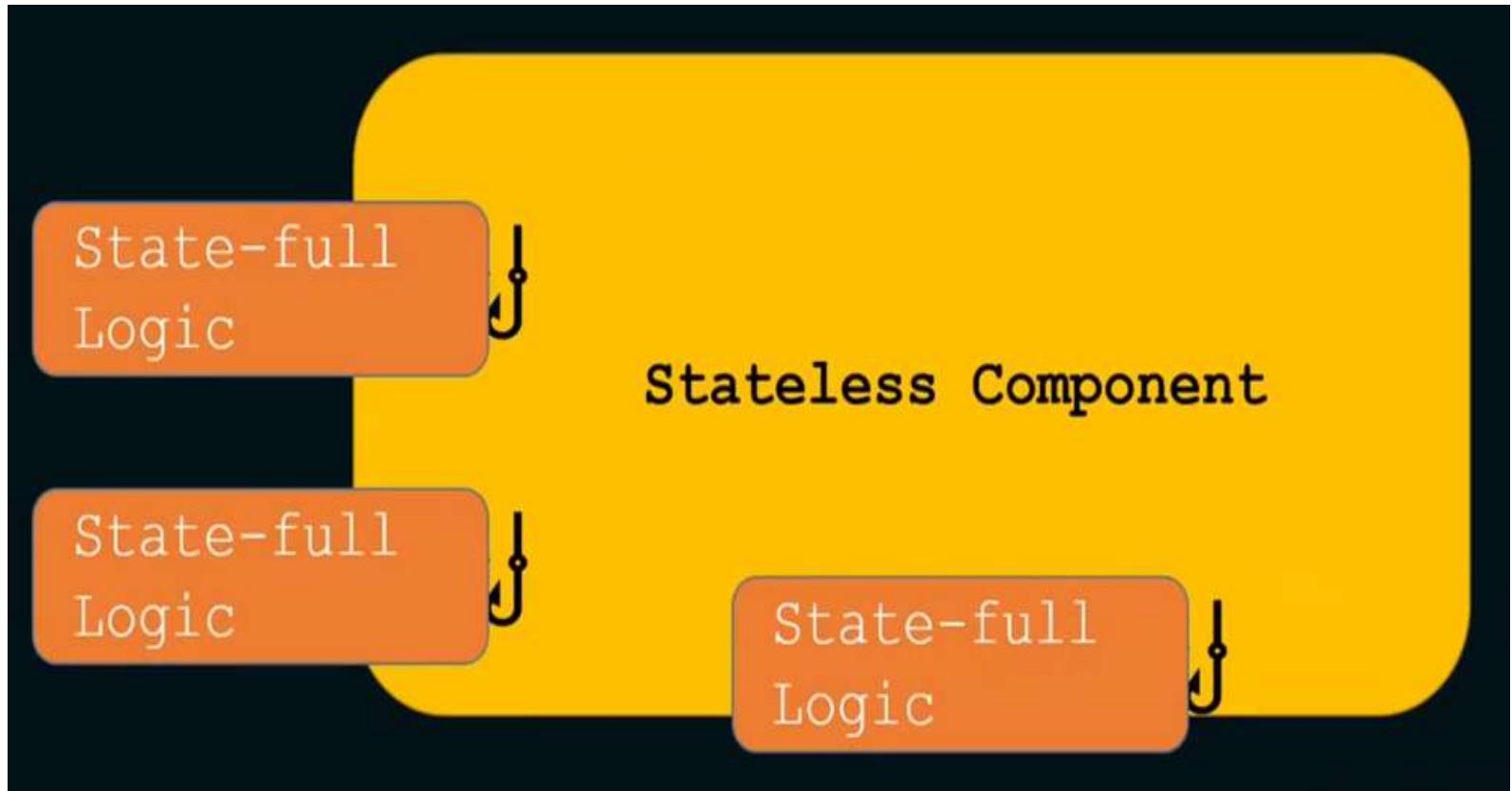
- Hooks are new React APIs added to React 16.8
- Before Hooks, **React functional and class components** performed distinct functions.
- Functional components were only used for **presentation purposes**—to render data to the UI.
- They could **only receive and render props from parent components**, which were usually class components.
- Functional components **did not keep track of an internal state** and did not know the component lifecycle.
- Thus, they were referred to as “**dumb components**.”

# Introduction

- Class components, on the other hand, track a component's internal state and enable you to perform operations during each phase by using lifecycle methods.
- For example, you can fetch data from an external API once a component mounts, update the state due to user interactivity, and unsubscribe from a store once a component unmounts.
- All of this is possible because a class component keeps track of its internal state and lifecycle.
- Consequently, class components were—and still are—referred to as “smart components.”

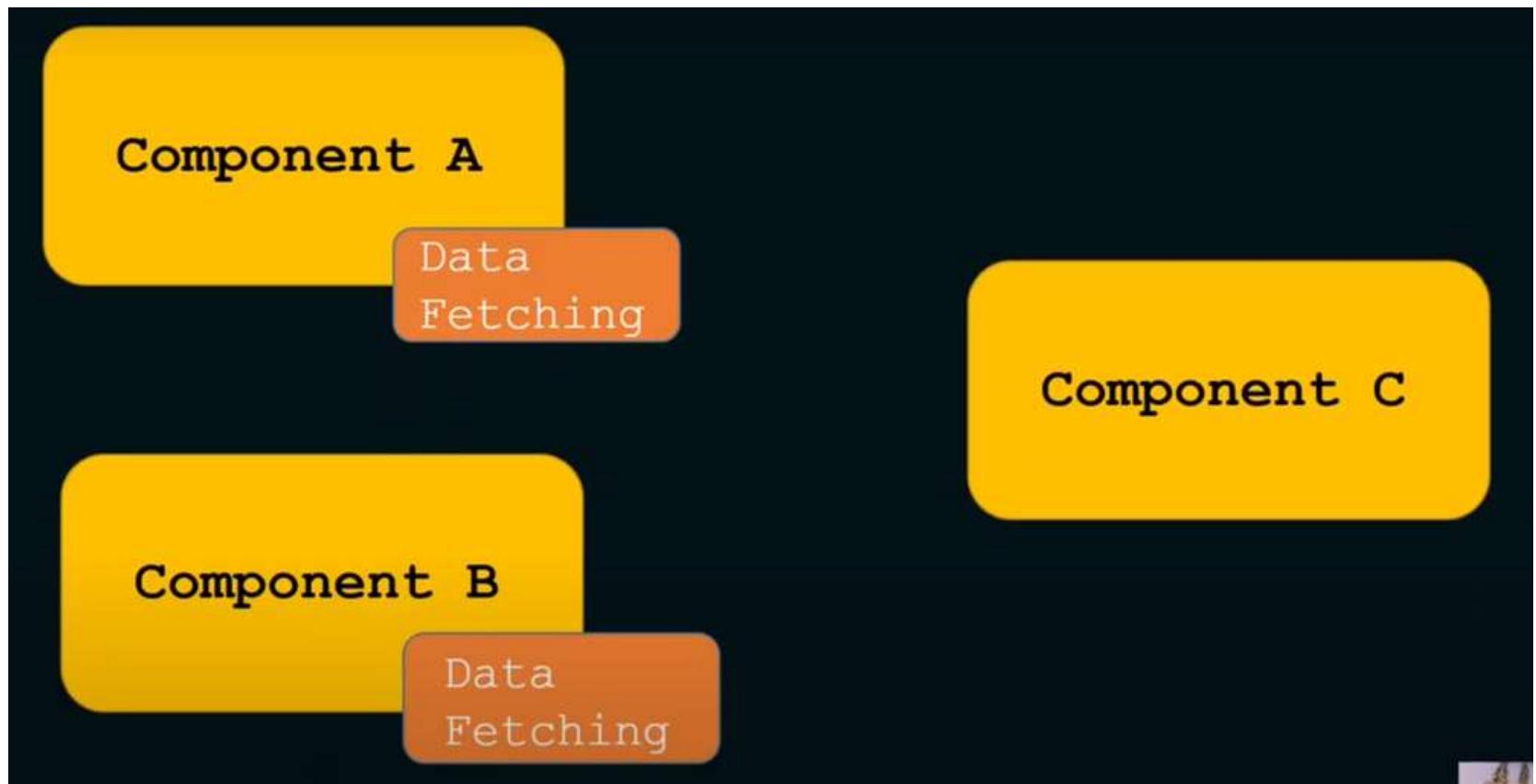
# Stateful vs. Stateless Components

- Components in React can be stateful or stateless.
  - A **stateful component** declares and manages local state in it.
  - A **stateless component** is a pure function that doesn't have a local state and side-effects to manage.



- The question is, what is this stateful logic?
  - It can be anything that needs to declare and manage a state variable locally.
- For example,
  - the logic to fetch data and manage the data in a local variable is stateful.
- We may also want to reuse the fetching logic in multiple components.

# Stateful Logic - Reuse



# What Exactly Are React Hooks?

- React Hooks are simple JavaScript functions that we can use to isolate the reusable part from a component.
- Hooks can be stateful and can manage side-effects.
  - **useState Hook**
  - **useEffect Hook**
  - **useRef Hook**
  - **useCallback Hook**
  - **useMemo Hook**
  - **useContext Hook**
  - **useReducer Hook**

# Rules of Hooks

- Only call Hooks at the top level.
  - Don't call Hooks inside loops, conditions, or nested functions.
- Only call Hooks from React function components.
  - Don't call Hooks from regular JavaScript functions.

# What is useState() hook?

- In React, **the state is data or properties you can use in our application.**
- States are mutable, meaning their value can change, and for that, the **useState() hook** is used to handle and manage your states.
- The **useState()** hook allows you to create, track and update a state in functional components.

# Declaring a state to React with useState()

- To use the useState() hook,
  - First import it from React,
  - or you will have to append it like React.useState() anytime you have to create a state.
- The useState() hook takes in the initial value of the state variable as an argument.
- This value could be of any data type, such as string, number, object, array, and lots more.

# Declaring a state to React with useState()

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

const App = () => {
    const number = useState(0);
    const string = useState('')
    const object = React.useState({})
    const array = React.useState([])

    return (
        // ...
    );
}
```

# Declaring a state to React with useState()

Initial value can be used directly within your project using curly braces alongside the state variable.

```
const App = () => {  
  const [count, setCount] = useState({ number: 5 });  
  return (  
    <div>  
      <p>{count.number}</p>  
    </div>  
  );  
};
```

# Declaring a state to React with useState()

The useState() hook does not return just a variable  
but it also returns a second value, a function that you can use to update the state value.

```
const [count, setCount] = useState(0);
```

current state

function to update state

Initial value

# Update a state with useState()

- The **second value is a function** you can use to update or set your state.
- This can be called anything, but it is best practice to use the variable name **with a prefix of set**.
- how this works?
- If you have a **number whose default value is set to 0**, and then you want this number to increase when a button is clicked.
- This means that when the **button is clicked**, you want your state to update by adding one to whatsoever number:

# Update a state with useState()

```
const App = () => {  
  const [count, setCount] = useState(0);  
  
  const incrementCount = () => {  
    setCount(count + 1);  
  };  
  
  return (  
    <div>  
      <p>{count}</p>  
      <button type="button" onClick={incrementCount}> Increment Count  
      </button>  
    </div>  
  );  
};
```

count stands for the state variable,  
setCount is the function used to update the state,  
while 0 is the initial value.

*UseStateDemo.js*

The state value is passed alongside a button with an onClick() event.

In the click event, an increment function is passed – use the setCount function to increment the count state.

# Using multiple state variables

- For example, if you want to create a state for user data such as **name, age, and hobby**.

```
const App = () => {
  const [name, setName] = useState("Anand");
  const [age, setAge] = useState(20);
  const [hobby, setHobby] = useState("reading");

  return (
    // ...
  );
};
```

Creating individual states for data that you can combine into one state won't be advisable when working with real-life data.

# Using multiple state variables

- You can combine all these into an **object** and then access them within our JSX via the **dot operator**.

```
const App = () => {
  const [userDetails, setUserDetails] = useState({
    name: 'Anand',
    age: 20,
    hobby: 'Reading',
  });

  return (
    <div>
      <h1>{userDetails.name}</h1>
      <p>
        {userDetails.age} || {userDetails.hobby}
      </p>
    </div>
  );
};
```

*UseStateDemo2.js*

# Without dot operator

```
const App = () => {
  const [{name, age, hobby}, setUserDetails] = useState({
    name: 'Anand',
    age: 20,
    hobby: 'Reading',
  });

  return (
    <div>
      <h1>{name}</h1>
      <p>
        {age} || {hobby}
      </p>
    </div>
  );
};
```

*UseStateDemo3.js*

# Using an object as a state variable

```
const App = () => {
  const [userDetails, setUserDetails] = useState({
    userName: {
      firstName: 'John',
      lastName: 'Doe',
    },
    age: 20,
    hobby: 'Reading',
  });
}

const changeName = () => {
  setUserDetails({
    ...userDetails,
    userName: {
      ...userDetails.userName,
      firstName: 'Jane',
    },
  });
};
```

# Using an object as a state variable

```
return (  
  <div>  
    <h1>  
      Hello {userDetails.userName.firstName}  
      {userDetails.userName.lastName},  
    </h1>  
    <p>  
      {userDetails.age} || {userDetails.hobby}  
    </p>  
    <button onClick={changeName}>Change Name</button>  
  </div>  
);  
};
```

# Initializing State as a Function

- You can initialize a state with a function that returns a value.

```
const expensiveComputation = () => {
  let number = 50;
  let newNumber = (50 % 10) * 10 - number;
  return newNumber;
};

const App = () => {
  const [calc, setCalc] = useState(() => expensiveComputation());
  return (
    <div>
      <p>{calc}</p>
    </div>
  );
};
```

# References

- <https://codersociety.com/blog/articles/react-hooks>
- <https://hygraph.com/blog/usestate-react>
- <https://react.dev/reference/react/useState>
- <https://blog.logrocket.com/guide-usestate-react/>
- <https://daveceddia.com/intro-to-hooks/>
- <https://www.freecodecamp.org/news/usestate-hook-3-different-examples/>





# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



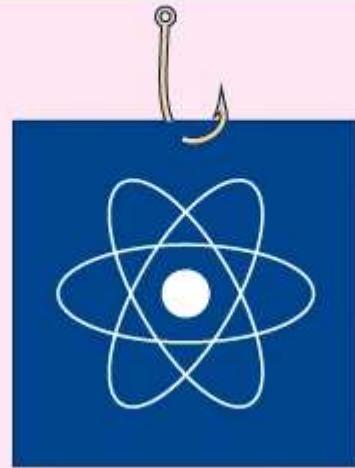
# INT436

## FULL STACK WEB APPLICATION DEVELOPMENT





## useEffect hook



# Need & Usage

What is the difference between useState and useEffect?

- **useState** is a hook used to manage state in functional components.
- **useEffect** is the hook that manages the side-effects in functional components.
- The **useEffect** hook accepts a function, which it will run after each render of the function component by default.
- The **useEffect** hook can be used to simulate the **componentDidMount()**, **componentDidUpdate()**, and **componentWillUnmount()** lifecycle methods in function components.

# Side-effects – Pure Function

- In computer science, a side effect is a result of an **impure function**.
- A pure function is one whose return value is always the same when given the same arguments, and that doesn't do anything that lasts past the running of the function except return a value.
- In simple terms, **pure functions do not have an internal state**.
- Therefore, all operations performed in pure functions are not **affected by their state**.
- As a result, the **same input parameters will give the same deterministic output** regardless of how many times you run the function.

# Pure Function - Example

```
function add(a,b) {  
    return a + b  
}  
  
console.log(add(4,5))
```

- This example contains a simple `add()` function, which gives 9 as the output.
- It is a very predictable output, and it does not depend on any external code.
- This makes the `add()` function a pure function.

# Impure Function

- An **impure function** is a function that contains one or more side effects.
- It mutates data outside of its lexical scope and does not predictably produce the same output for the same input.
- Side effects can occur in your program when it uses an external code block inside a function.
- **Math.random()**
  - Math.random() is an **impure function**
  - since it modifies the internal state of the Math object and provides different results with each call.
  - Math.random() can contain side effects.
    - Math.random(); (Output: 0.4450692005082965)
    - Math.random(); (Output: 0.7533405303023756)
    - Math.random(); (Output: 0.4011148700956255)

# Impure Function

- In the example, the variable preNumber is used inside the addValue() function.
- This behavior can result in the following side effects.
- **Side effect 1: Dependency issue**
  - The addValue() method depends on the preNumber variable.
  - If preNumber is not defined or not available, the method will throw an error.
- **Side effects 2: External code modification**
  - When the addValue() function executes, it changes the state of the preNumber variable.
  - It shows that the addValue() method has a side effect of modifying external code.

```
var preNumber =2;  
  
function addValue(newNumber){  
    return preNumber += newNumber;  
}
```

# Side-effects

- A functional React component uses props and/or state to calculate the output.
- If the component makes calculations that don't target the output value, then these calculations are named side-effects.
- Anything that a function does that has an effect outside of the function, other than producing a return value, is a side effect.
- Side effects in a browser-based application can include:
  - Modifying global variables.
  - Making a network request.
  - Changing the DOM.
  - Writing to a database or a file.
  - Modifying an argument.

# Side-effects

- Keep component rendering and the side-effect logic are **independent**.
- Performing side-effects directly in the body of the component is a **mistake**, because the body **computes the component's output or calls hooks**.

```
function Greet({ name }) {  
  
  const message = `Hello, ${name}!`; // Calculates output  
  
  // Bad!  
  
  document.title = `Greetings to ${name}`; // Side-effect!  
  
  return <div>{message}</div>; // Calculates output  
}
```

# Comparison



## PURE VS. IMPURE FUNCTIONS IN JAVASCRIPT

### FEATURES

### PURE FUNCTION

### IMPURE FUNCTION

Input and output

Same output for same input

Different output for same input

Side effects



Return value



Behavior of the state

No change

State of the program, system will change

Testing accuracy

Easy for testing

Testing is difficult due to side effects

# useEffect()

- How to decouple rendering from the side-effect?
- useEffect() — the hook that runs side-effects independently of rendering.

```
import { useEffect } from 'react';

function Greet({ name }) {
  const message = `Hello, ${name}!`; // Calculates output
  useEffect(() => {
    // Good!
    document.title = `Greetings to ${name}`; // Side-effect!
  }, [name]);
  return <div>{message}</div>; // Calculates output
}
```

# useEffect() arguments

- **useEffect()** hook accepts 2 arguments.

*UseEffectDemo1.js*

**useEffect(callback[, dependencies]);**

- **callback** is a function that contains the side-effect logic.
- **callback** is executed right after the DOM update.
- **dependencies** is an optional array of dependencies.
- **useEffect()** executes **callback** only if the **dependencies** have changed between renderings.

# Side-effects

```
useEffect(() => {  
  
  document.title = `Greetings to ${name}`;  
}, [name])
```

- The **document title update is the side-effect** because it doesn't directly calculate the component output.
- That's why the **document title update is placed in a callback** and supplied to `useEffect()`.
- Also, you don't want the **document title update to run every time Greet component renders**.
- You only want it to happen when the name prop changes — that's the reason you **supplied *name as a dependency*** to `useEffect(callback, [name])`.

# Dependencies argument

- **dependencies argument** of `useEffect(callback, dependencies)` lets you control when the side-effect runs.
- If dependencies are:
  - Not provided: the side-effect runs after every rendering.

```
import { useEffect } from 'react';

function MyComponent() {
  useEffect(() => {
    // Runs after EVERY rendering
  });
}
```

# Dependencies argument

- If dependencies are:
  - An empty array []: the side-effect *runs once after the initial rendering.*

```
import { useEffect } from 'react';

function MyComponent() {
  useEffect(() => {
    // Runs ONCE after initial rendering
  }, []);
}
```

# Dependencies argument

- If dependencies are:
  - Has props or state values [prop1, prop2, ..., state1, state2]:
  - the side-effect runs once after initial rendering and then only when any dependency value changes.

```
import { useEffect, useState } from 'react';
function MyComponent({ prop }) {
  const [state, setState] = useState('');
  useEffect(() => {
    // Runs ONCE after initial rendering
    // and after every rendering ONLY IF `prop` or `state`
    // changes
  }, [prop, state]);
}
```

# Component lifecycle

- The **dependencies argument** of the `useEffect()` lets you **catch certain component lifecycle events.**
- when the component has been mounted or a specific prop or state value has changed.
- **Component did mount**
- Use an empty dependencies array to invoke a side-effect once after component mounting.

```
import { useEffect } from 'react';

function Greet({ name }) {
  const message = `Hello, ${name}!`;

  useEffect(() => {
    // Runs once, after mounting
    document.title = 'Greetings page';
  }, []);
}

return <div>{message}</div>;
}
```

*UseEffectDemo2.js*

```
// First render
<Greet name="Abcd" /> // Side-effect RUNS

// Second render, name prop changes
<Greet name="Xyz" /> // Side-effect DOES NOT RUN
// Third render, name prop changes
<Greet name="Mnop"/> // Side-effect DOES NOT RUN
```

# Dependencies argument

- **Component did update**
- Each time the side-effect uses props or state values, you must indicate these values as dependencies.

```
import { useEffect } from 'react';

function Greet({ name }) {
  const message = `Hello, ${name}`;

  useEffect(() => {
    document.title = `Greetings to ${name}`;
  }, [name]);

  return <div>{message}</div>;
}
```

// First render  
// Side-effect RUNS  
<Greet name="Abcd" />

// Second render, name prop changes  
// Side-effect RUNS  
<Greet name="Xyz" />

// Third render, name prop doesn't change  
// Side-effect DOES NOT RUN  
<Greet name="Xyz" />

// Fourth render, name prop changes  
// Side-effect RUNS  
<Greet name="Klmno" />

# Side-effect cleanup

- In the `useEffect` hook in React,
    - the **cleanup function** is a function that is executed before the hook **is** executed again or the component is unmounted.
  - This function can be used to perform tasks such as
    - **canceling network requests,**
    - **removing event listeners,** or
    - **cleaning up any other resources that were created by the hook.**

# Side-effect cleanup

- To define a cleanup function for the `useEffect` react hook, you can return the function from the hook itself.
- In the example, **the cleanup function is defined by returning a function** from the `useEffect` react hook.
- This function will be executed before **the hook is executed again or the component is unmounted**.
- Inside the cleanup function, you can perform **any necessary cleanup tasks**, such as canceling network requests or removing event listeners.

```
useEffect(() => {  
  // Do something  
  
  // Define the cleanup function  
  return () => {  
    // Clean up  
  };  
}, [dependency1, dependency2, ...])
```

# Side-effect cleanup

- It is important to note that the cleanup function will only be executed
  - if the hook has dependencies and
  - one of those dependencies changes.
- If the hook
  - does not have any dependencies or
  - if an empty array is passed as the dependencies, then the cleanup function will not be executed.

# Side-effect cleanup

Cleanup works the following way.

A) After initial rendering,

- `useEffect()` invokes the callback with the side-effect.
- cleanup function is not invoked.

B) On later renderings,

- before invoking the next side-effect callback,
- `useEffect()` invokes the cleanup function from the previous side-effect execution (to clean up everything after the previous side-effect),
- then invokes the current side-effect.

C) Finally,

- after unmounting the component
- `useEffect()` invokes the cleanup function from the latest side-effect.

# References

- <https://dmitripavlutin.com/react-useeffect-explanation/>
- <https://www.scaler.com/topics/react/useeffect/>
- <https://daveceddia.com/useeffect-hook-examples/>
- <https://www.educative.io/blog/react-useeffect-hook>
- <https://react.dev/reference/react/useEffect>





# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

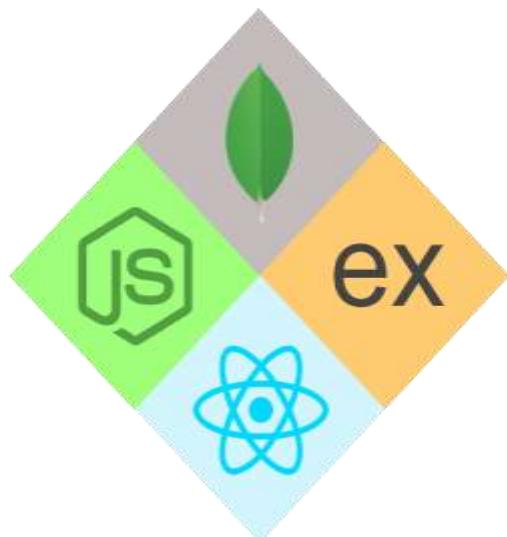
(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



## INT436

# FULL STACK WEB APPLICATION DEVELOPMENT





# React REST API

# What is REST API?

- REST is an acronym for **Representational State Transfer**
- REST is an architectural style for **building web services** that interact via an HTTP protocol.
- Its principles were formulated in 2000 by computer scientist **Roy Fielding**



# REST API concepts

- The **key elements** of the REST API paradigm are
  - **a client or software** that runs on a user's computer or smartphone and initiates communication
  - **a server** that offers an API as a means of access to its data or features and
  - **a resource**, which is any piece of content that the server can provide to the client (for example, a video or a text file).
- To get access to a resource, **the client sends an HTTP request**.
- In return, **the server generates an HTTP response** with encoded data on the resource.
- **Both types of REST messages are self-descriptive**, meaning they contain information on **how to interpret and process them**.

# The Six Rules of REST APIs

The following are **some of the principles** of the REST architectural style:

1. Client-Server Separation
2. Uniform Interface
3. Stateless
4. Layered System
5. Cacheable
6. Code on Demand (Optional)

# REST API methods and request structure

- Any **REST request** includes four essential parts:
  - an **HTTP method**, an **endpoint**, **headers**, and a **body**.
- An **HTTP method** **describes what is to be done with a resource**.
- There are **four basic methods** also named **CRUD operations**:
  - **POST** to Create a resource,
  - **GET** to Retrieve a resource,
  - **PUT** to Update a resource, and
  - **DELETE** to Delete a resource.

# REST API methods and request structure

- **An endpoint** contains a Uniform Resource Identifier (URI) indicating where and how to find the resource on the Internet.
  - The most common type of ***URI is a Unique Resource Location (URL)***, serving as a complete web address.
- **Headers** store information relevant to both the client and server.
  - Mainly, **headers provide authentication** data — such as an API key, the name or IP address of the computer where the server is installed, and the information about the **response format**.
- **A body** is used to convey additional information to the server.
  - For instance, it may be a piece of data you want to add or replace.

# REST API Request

**Endpoint** →

`https://apiurl.com/review/new`

**HTTP Method** →

`POST`

**HTTP Headers** →

`content-type: application/json`  
`accept: application/json`  
`authorization: Basic abase64string`

**Body** →

```
{  
  "review" : {  
    "title" : "Great article!",  
    "description" : "So easy to follow.",  
    "rating" : 5  
  }  
}
```



`https://www.sitepoint.com/rest-api/`

# REST response structure

- **In response,**
  - the server **does not send** the sought-for resource itself
  - **but its representation** — a machine-readable description of its current state.
  - The same resource can be represented in different formats, but **the most popular ones are XML and JSON.**
- Whenever relevant,
  - a server includes in **the response hyperlinks or hypermedia** that links to other related resources.

# REST API Response with Hypermedia

HTTP/1.1 200 OK  
Content-Type: text/html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Home Page</title>
  </head>
  <body>
    <div>Hello World!</div>
    <a href= "http://www.recurse.com"> Check out the Recurse Center! </a>
    
  </body>
</html>
```

tells the client to make a GET request to  
<http://www.example.com/awesome-pic.jpg> to display the user's image

tells the client to make a GET request to  
<http://www.recurse.com> if the user clicks on the link

# SOAP vs REST

	SOAP	REST
Stands for	Simple Object Access Protocol	Representational State Transfer
What is it?	SOAP is a protocol for communication between applications	REST is an architecture style for designing communication interfaces.
Design	SOAP API exposes the operation.	REST API exposes the data.
Transport Protocol	SOAP is independent and can work with any transport protocol.	REST works only with HTTPS.
Data format	SOAP supports only XML data exchange.	REST supports XML, JSON, plain text, HTML.
Performance	SOAP messages are larger, which makes communication slower.	REST has faster performance due to smaller messages and caching support.
Scalability	SOAP is difficult to scale. The server maintains state by storing all previous messages exchanged with a client.	REST is easy to scale. It's stateless, so every message is processed independently of previous messages.
Security	SOAP supports encryption with additional overheads.	REST supports encryption without affecting performance.
Use case	SOAP is useful in legacy applications and private APIs.	REST is useful in modern applications and public APIs.

# 3 Ways to Make React API Calls

# 3 Ways to Make React API Calls

- In React, we can make the API call in the following ways:
  - XMLHttpRequest
  - Fetch API
  - Axios

# 1. XMLHttpRequest

- In JavaScript,
  - the XMLHttpRequest object is an API for sending HTTP requests from a web page to a server.
- It's a low-level API
  - because it only provides a basic mechanism for making HTTP requests and
  - leaves it up to the developer
    - to parse the response,
    - handle errors and
    - manage the request's state.

# Example

```
import React, { useState } from 'react';
function Example() {
  const [data, setData] = useState(null);

  function handleClick() {
    const xhr = new XMLHttpRequest();
    xhr.open('GET', 'https://api.example.com/data');
    xhr.onload = function() {
      if (xhr.status === 200) {
        setData(JSON.parse(xhr.responseText));
      }
    };
    xhr.send();
  }

  return (
    <div>
      <button onClick={handleClick}>Get Data</button>
      {data ? <div>{JSON.stringify(data)}</div> : <div>Loading...</div>}
    </div>
  );
}
```

## 2. Fetch API

- Fetch API is a modern, **promise-based API** for making HTTP requests in JavaScript.
- It provides a simple and flexible interface for making GET, POST, PUT and DELETE requests and handling the response from the server.

## 2. Fetch API - Example

```
import React, { useEffect, useState } from 'react';

const UserComponent = () => {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetchData();
  }, []);

  const fetchData = async () => {
    try {
      const USER_API = 'https://jsonplaceholder.typicode.com/users';
      const response = await fetch(USER_API);
      const jsonData = await response.json();
      setData(jsonData);
    } catch (error) {
      console.log('Error:', error);
    }
  };
}
```

# XMLHttpRequest vs. Fetch API

- Simpler syntax:
  - Fetch API uses a more concise syntax that is easier to read and write.
  - With Fetch API, **you can make a request and handle the response using a single function call**
  - while with XMLHttpRequest, you need to **create an instance of the XMLHttpRequest object, set properties and register event listeners.**
- Automatic type conversion:
  - Fetch API provides **automatic type conversion for request and response data**, making it easier to work with things like JSON data, for example.
  - With XMLHttpRequest, we **must parse** the response data manually,

### 3. AXIOS

- Axios is a popular **JavaScript library** for making HTTP requests.
- Axios makes it easy to send **asynchronous HTTP requests** to REST endpoints and perform CRUD operations (create, read, update and delete), as well as handle the responses.
- To use Axios in your React application you first need to install it.
- **npm install axios**
- Then, you'll need to import Axios into your React component to use it.
- **import axios from 'axios';**

# Example

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [posts, setPosts] = useState([]);
  useEffect(() => {
    axios.get('https://jsonplaceholder.typicode.com/posts')
      .then(response => {
        setPosts(response.data);
      })
      .catch(error => {
        console.error(error);
      });
  }, []);
}

return (
  <ul>
  {posts.map(post => (
    <li key={post.id}>{post.title}</li>
  )));
  </ul>
); }
```

*RESTAPIDemo3.js*

# Fetch API vs. Axios

- **Abort requests:**
  - Axios **allows you to cancel a request** by using the `CancelToken` feature.
  - With `fetch`, you **can't cancel a request** once it has started.
- **Improved functionality:**
  - Axios has additional functionality not available in `fetch`
    - such as the ability to make **GET and POST requests with a single line of code**
    - as well as the ability to make requests with a **specified timeout**.

# References

- <https://www.altexsoft.com/blog/rest-api-design>
- <https://builtin.com/software-engineering-perspectives/react-api>
- <https://websparrow.org/react/how-to-call-a-rest-api-in-react>
- <https://stackabuse.com/building-a-rest-api-with-node-and-express/>
- <https://aws.amazon.com/compare/the-difference-between-soap-rest>
- <https://blog.hubspot.com/website/what-is-rest-api>

# References

- <https://medium.com/@nutanbhogendrasharma/step-by-step-consume-rest-api-in-react-application-48388f6c4d9c>
- <https://mocki.io/call-a-rest-api-in-react>
- <https://hevodata.com/learn/react-rest-apis/>
- <https://www.positronx.io/how-to-use-fetch-api-to-get-data-in-react-with-rest-api/>
- <https://www.sitepoint.com/rest-api/>





# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



## INT436

# FULL STACK WEB APPLICATION DEVELOPMENT





# EXPRESS JS

- Express is a minimal and flexible Node.js web application framework that provides a robust set of features for building web and mobile applications.
- Web Applications:
  - Express is often used to build full-stack web applications where the *server-side logic* is managed by Express.
- RESTful APIs:
  - Express is commonly used to build RESTful APIs that serve data to client-side applications or other services.
- Single-Page Applications (SPAs):
  - Express can serve as the backend for SPAs, managing routes and handling data requests.
- Microservices: Express is also a popular choice for building microservices due to its lightweight nature and flexibility.

# Key Features of Express

- **Routing:**
  - Express provides a powerful **routing mechanism** to define various routes of your application.
  - You can specify what should happen when a user visits a particular path (URL) using methods like get, post, put, delete, etc.
- **Middleware:**
  - Middleware functions are functions that **have access to the request object (req), the response object (res), and the next middleware function** in the application's request-response cycle.
  - They **can execute code**, modify the request and response objects, end the request-response cycle, and call the next middleware function.
  - Middleware is used for tasks such as **parsing JSON** or form data, logging requests, handling errors, and more.

# Key Features of Express

- Serving Static Files:
  - Express can **serve** static files, such as **images**, **CSS files**, and **JavaScript files**, using the `express.static` middleware.
- Request and Response Objects:
  - Express provides the `req` (request) and `res` (response) objects **to handle HTTP requests** and responses.
  - You can retrieve data sent by the client, such as **query parameters**, **form data**, **headers**, etc., and send responses back, such as HTML pages, **JSON** data, or files.
- RESTful APIs:
  - Express is commonly used to build RESTful APIs. **It simplifies the process** of setting up routes, handling different HTTP methods, and managing request data.

# Example

```
const express = require('express');
const app = express();
app.get('/', (req, res) => {
  res.send('Hello World!');
});
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

- `const express = require('express');`
- Initialize an Express application.
- The `express()` function creates an instance of an Express application and stores it in the app variable.
- This app instance is used to define the routes and middleware for handling HTTP requests.

# Example

- `app.get('/hello', (req, res) => { res.send('Hello World'); });`
- This line defines a **route handler** for the **HTTP GET request** at the **/hello** path.`app.get('/hello', ...);`
- The 'get' method of the 'app' object is used to **define a route** that **listens for GET requests at the '/hello' path.**
- When a request is made to this path, the specified **callback function** is executed.

# Example

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

- This is **the callback function** that is executed when the /hello route is accessed.
- It takes **two arguments**:
- **req**: The request object, which **contains information about the HTTP request** (e.g., request headers, query parameters).
- **res**: The response object, which is **used to send a response back to the client**.

# Example

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

- This line **sends a plain text response** with the content "**Hello World**" to **the client** that made the GET request.

# Example

```
app.listen(3000, () => {  
    console.log('Server is running on port 3000');  
});
```

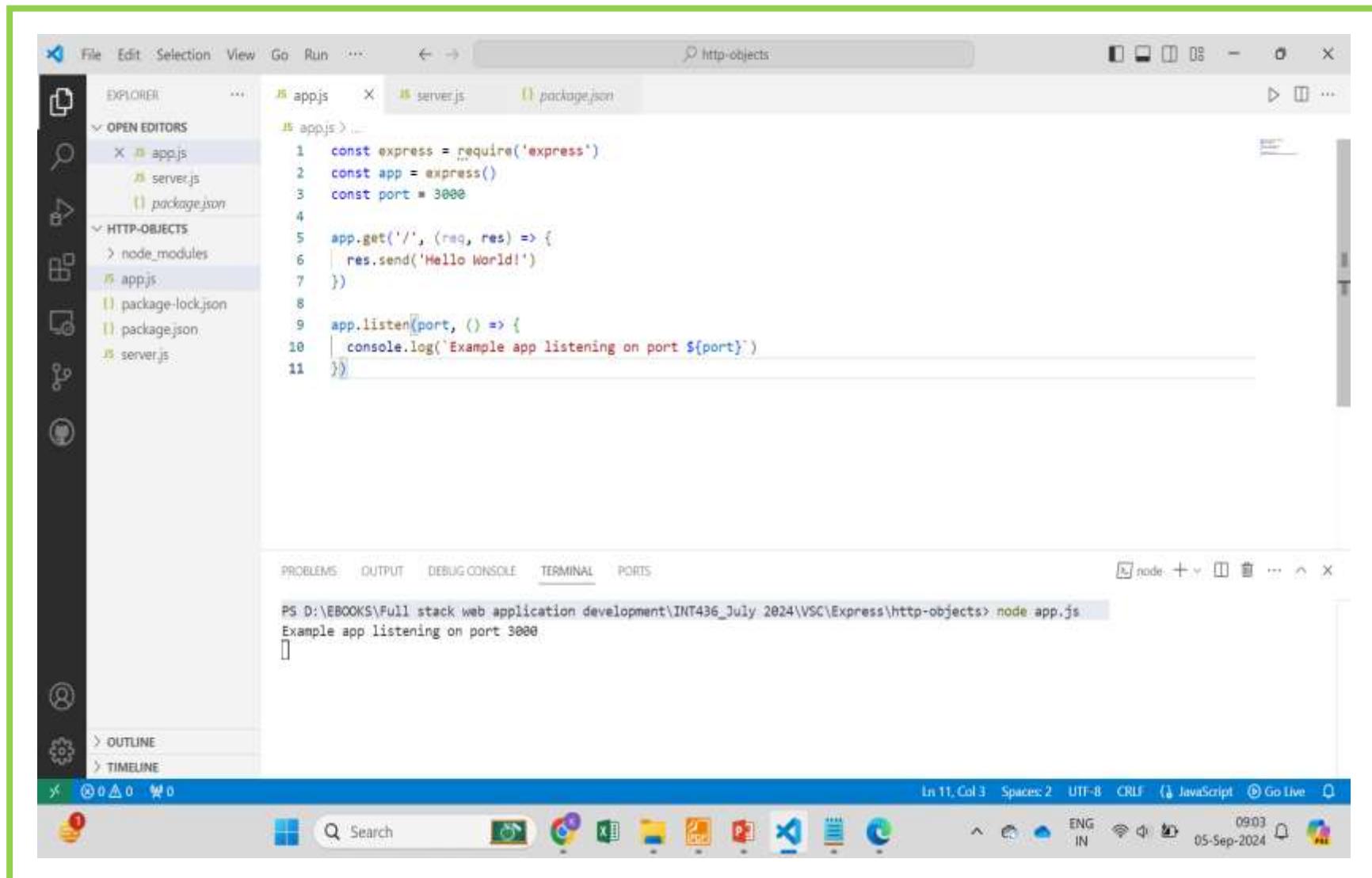
- This piece of code is typically found at the end of an Express application **script to initiate the server** after all routes and middleware have been defined.
- This listen method **instructs the server to start listening for incoming requests** on a specific port number, which in this case is 3000.
- Port 3000 is a commonly used **port for development purposes**, but you can specify any other port number that your system allows.
- The server **can only listen to one port at a time** unless you start multiple instances on different ports.

# Example

```
app.listen(3000, () => {  
    console.log('Server is running on port 3000');  
});
```

- The **callback function is executed once** the server successfully starts listening on the specified port.
- Inside this callback, a message 'Server is running on port 3000' is logged to the console.
- This is useful for developers to confirm that **the server has started successfully** and is ready to handle requests.

# Sample screen



The screenshot shows a Microsoft Visual Studio Code (VS Code) interface with a green border. The left sidebar contains icons for Explorer, Search, Open Editors, and Outline. The main editor area displays an `app.js` file with the following code:

```
const express = require('express')
const app = express()
const port = 3000

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

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

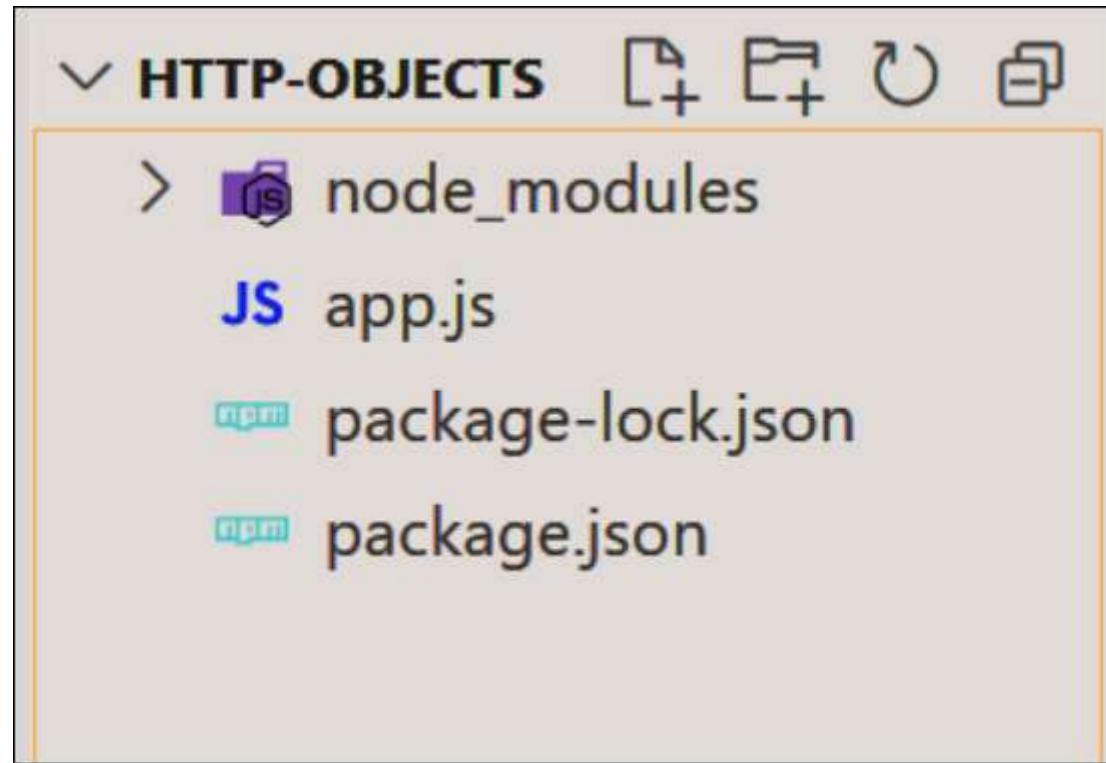
The terminal at the bottom shows the output of running the application:

```
PS D:\EBOOKS\Full stack web application development\INT436_July 2024\VSC\Express\http-objects> node app.js
Example app listening on port 3000
```

The status bar at the bottom right shows the date and time: 05-Sep-2024 09:03.

# Steps to use req and res objects in Express

## Project Structure

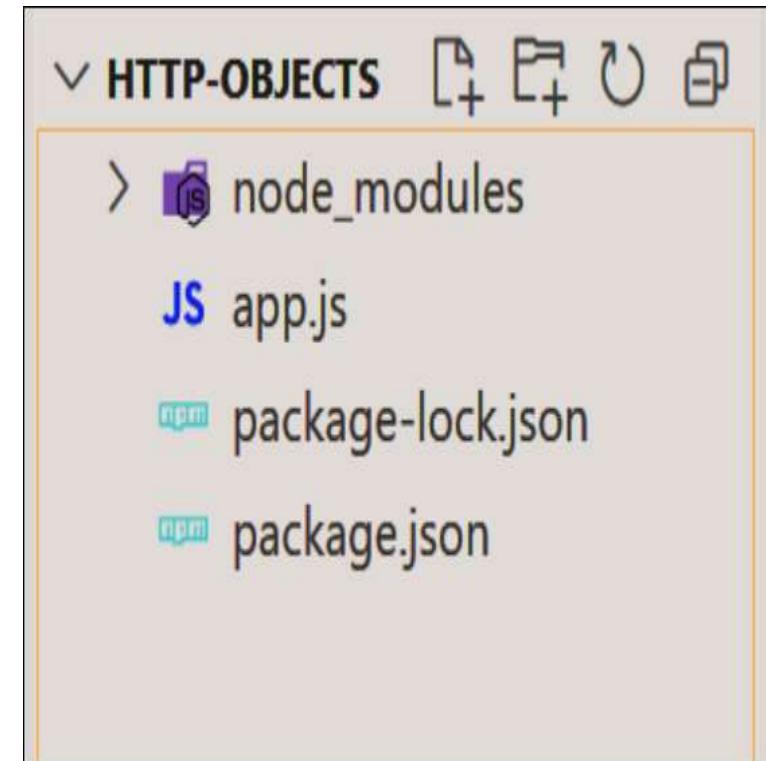


# Steps to use req and res objects in Express

- **Step 1:**
  - In the first step, we will create the new folder as an http-objects by using the below command in the VScode terminal.
- `mkdir http-objects`
- `cd http-objects`
- **Step 2:**
  - After creating the folder, initialize the NPM using the below command. Using this the package.json file will be created.
- `npm init -y`

# Steps to use req and res objects in Express

- **Step 3:**
  - Now, we will install the express dependency for our project using the below command.
- **npm i express**
- **Step 4:**
  - Now create the below Project Structure of our project which includes the file as app.js.



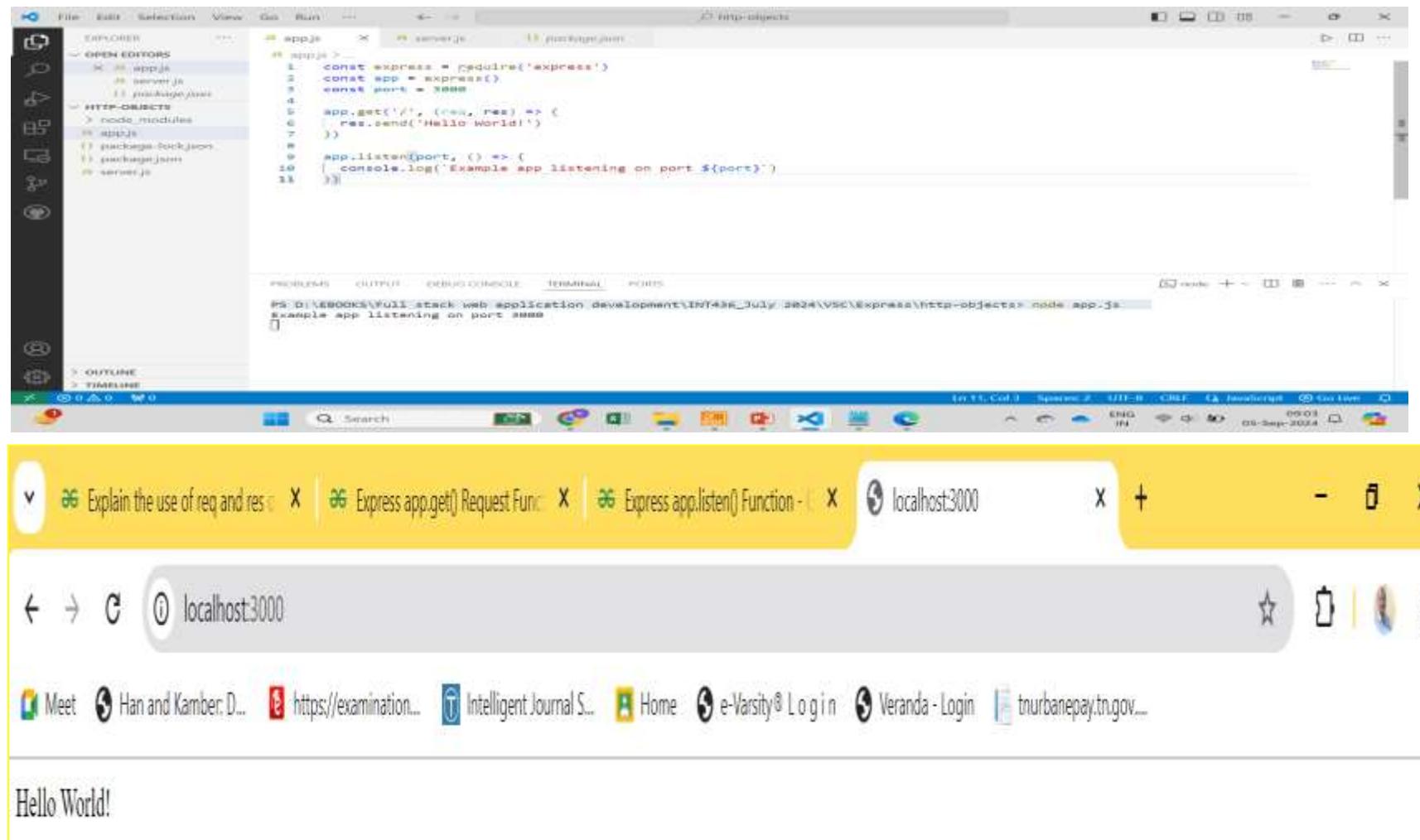
# Steps to use req and res objects in Express

- The updated dependencies in package.json file will look like:

```
"dependencies": {  
  "express": "^4.19.2"  
}  
}
```

- Write the following code in app.js file (Slide. No:6)
- Start the server by using the below command:
  - node app.js

# Output



The screenshot shows a development environment and a browser window. In the top half, a code editor displays an `app.js` file with the following code:

```
const express = require('express')
const app = express()
const port = 3000

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

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

The terminal below shows the command run and its output:

```
PS D:\EBOOKS\Full_stack_web_application_development\INT436_July_2024\VSCode\Express\http-objects> node app.js
Example app listening on port 3000
```

In the bottom half, a browser window is open at `localhost:3000`, displaying the text "Hello World!".

# References

- [https://www.geeksforgeeks.org/express-js-app-get-request-function/?ref=ml\\_lbp](https://www.geeksforgeeks.org/express-js-app-get-request-function/?ref=ml_lbp)





# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



# INT436

## FULL STACK WEB APPLICATION DEVELOPMENT





# EXPRESS JS

- Router
  - takes a client request
  - matches it against **any routes** that are present, and
  - executes the **handler** function that is associated with that route.
- The handler function is expected to generate the **appropriate response**.
  
- Request Matching
  - When a request is received, the first thing that happens is request matching.  

```
app.get('/hello', (req, res) => {
  res.send('Hello World');
```
  - The **request's method is matched with the route method** and also the request URL with the path spec ('/hello' in the above example).
  - If a request matches this spec, then the handler is called.

# Routing

- **Route Parameters**
  - **Route parameters** are named segments in the path specification that match **a part of the URL**.
  - If a match occurs, the value in that part of the URL is supplied as a variable in the request `object.app.get('/customers/:customerId', ...)`
  - The URL `/customers/1234` will match the above route specification, and so will `/customers/4567`.
  - In either case, the customer ID will be captured and supplied to the handler function as part of the request in `req.params`, with the name of the parameter as the key.
  - Thus, `req.params.customerId` will have the value `1234` or `4567` for each of these URLs, respectively.

- Route Lookup
  - Multiple routes can be set up to match different URLs and patterns.
  - The router does not try to find a best match; instead, it tries to match all routes in the order in which they are installed.
  - The first match is used.
  - So, if two routes are possible matches to a request, it will use the first defined one.
  - It is up to you to define routes in the order of priority.
  - So, when you add patterns rather than very specific paths, you should be careful to add the more generic pattern after the specific paths.
    - For example, if you want to match everything that goes under /api/, that is, a pattern like /api/\*, you should add this route only after all the specific routes that handle paths such as /api/issues.

# Handler Function

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

- Handler Function
  - Once a route is matched, the **handler function** is called, which in the above example was an **anonymous function supplied** to the route setup function.
  - The parameters passed to the handler are a **request object** and a **response object**.

# Request /Response Objects

- The objects 'req' and 'res' are responsible for handling the **HTTP requests** and responses in the web applications.
- By using **the req object**,
  - we can access the various components of the **incoming HTTP request** which consists of data, headers, parameters, etc.
- By using **the res object**,
  - we **can send the HTTP responses** to the client which includes the resource, data, status codes, and headers.
  - we **can send the error responses** which is important for handling the errors and providing feedback to the client.

# Request Object

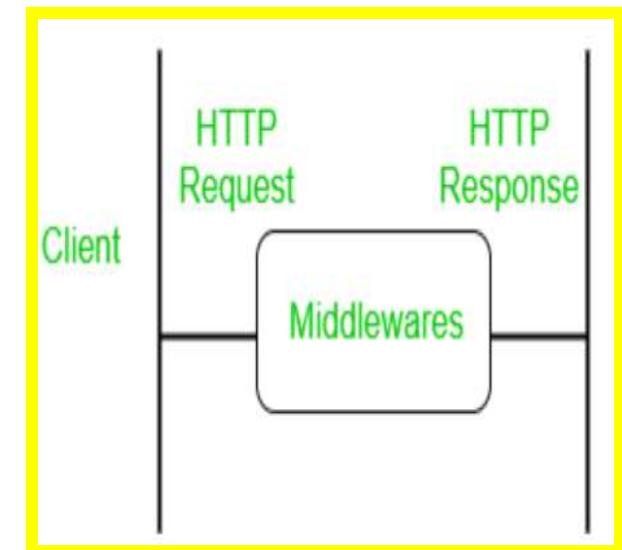
- **Methods**
- ‘req.query’: Access query string parameters
- ‘req.params’: Access route parameters
- ‘req.body’: Access the body of the request (for POST/PUT)
- ‘req.headers’: Access request headers
  
- **Properties**
- ‘req.method’: HTTP method (e.g., GET, POST)
- ‘req.url’: URL of the request
- ‘req.cookies’: Cookies sent by the client
- ‘req.ip’: IP address of the client

# Response Object

- Methods
  - ‘res.send()’: Send a response body
  - ‘res.json()’: Send a JSON response
  - ‘res.status()’: Set the status code of the response
  - ‘res.sendFile()’: Send a file as the response
- Properties
  - ‘res.headersSent’: Boolean indicating if the headers have been sent
  - ‘res.locals’: Object for passing data to the views
  - ‘res.app’: Access the Express application object

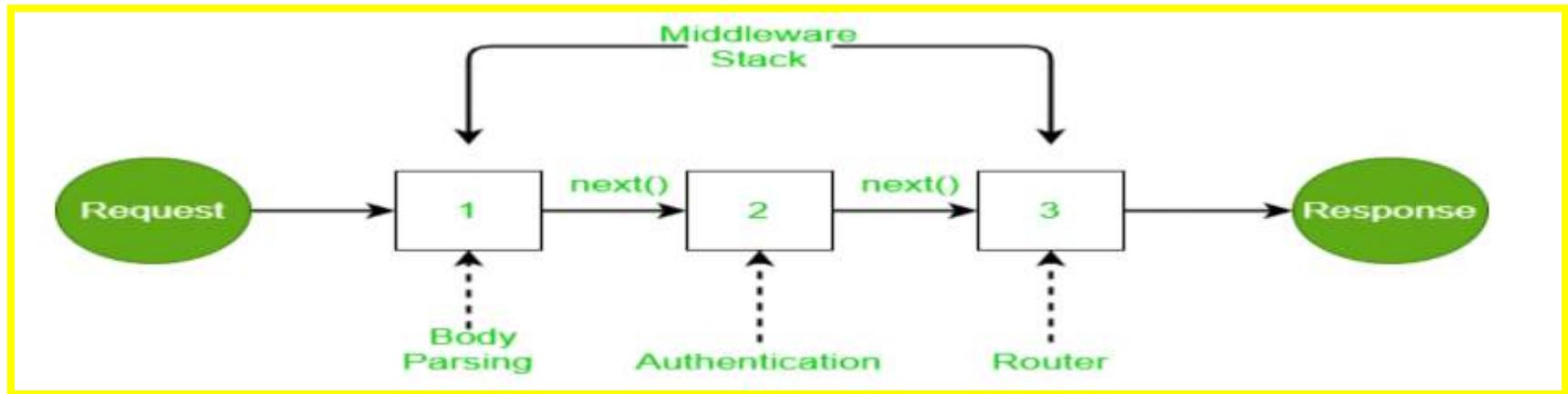
# Middleware

- Express is a web framework that has minimal functionality of its own.
- An Express application is essentially a series of middleware function calls.
- In fact, the Router itself is nothing but a middleware function.
- Middleware functions are those that have access to the request object (`req`), the response object (`res`), and the next middleware function in the application's request-response cycle.
- The next middleware function is commonly denoted by a variable named `next`.



- Types of Middleware
  - Express JS offers different types of middleware and you should choose the middleware on the basis of functionality required.
- Application-level middleware:
  - Bound to the entire application using `app.use()` or `app.METHOD()` and executes for all routes.
- Router-level middleware:
  - Associated with specific routes using `router.use()` or `router.METHOD()` and executes for routes defined within that router.
- Error-handling middleware:
  - Handles errors during the request-response cycle. Defined with four parameters (`err, req, res, next`).
- Built-in middleware:
  - Provided by Express (e.g., `express.static`, `express.json`, etc.).
- Third-party middleware:
  - Developed by **external packages** (e.g., `body-parser`, `morgan`, etc.).

# Middleware Chaining



- Middleware **can be chained** from one to another, Hence creating a chain of functions that are executed in order.
- The **last function sends the response back to the browser**.
- So, before sending the response back to the browser the different middleware processes the request.
- The **next() function** in the express is responsible for calling the **next middleware function** if there is one.

# Steps to Implement Middleware in Express

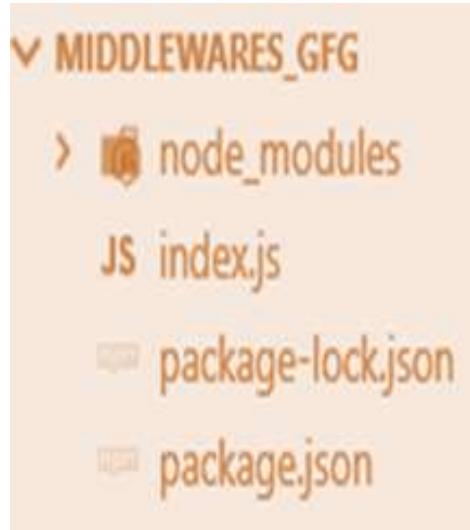
- **Step 1:**
  - Go to your project directory and enter the following command to create a NodeJs project.
  - Make sure that NodeJs is installed in your machine.
- **npm init -y**
- **Step 2:**
  - Install two dependencies using the following command.
- **npm install express nodemon**

# Steps to Implement Middleware in Express

- Step 3:
  - In the scripts section of the package.json file, add the following code line.
- "start": "nodemon index.js",
- Step 4:
  - create an index.js file in the directory.
  - Make sure that it is not inside any subdirectories of the directory you are working in.
- The updated dependencies in package.json file will look like:

```
"dependencies": {  
  "express": "^4.19.2",  
  "nodemon": "^3.1.4"  
}
```

# Steps to Implement Middleware in Express



Step to run the application:

Run the code by entering the following command on the terminal.

npm start

**Step 5:** Now we will set up our **express app** and send a **response** to our **server**.

Here is the code for the **index.js** file.

```
const express = require("express");
const app = express();

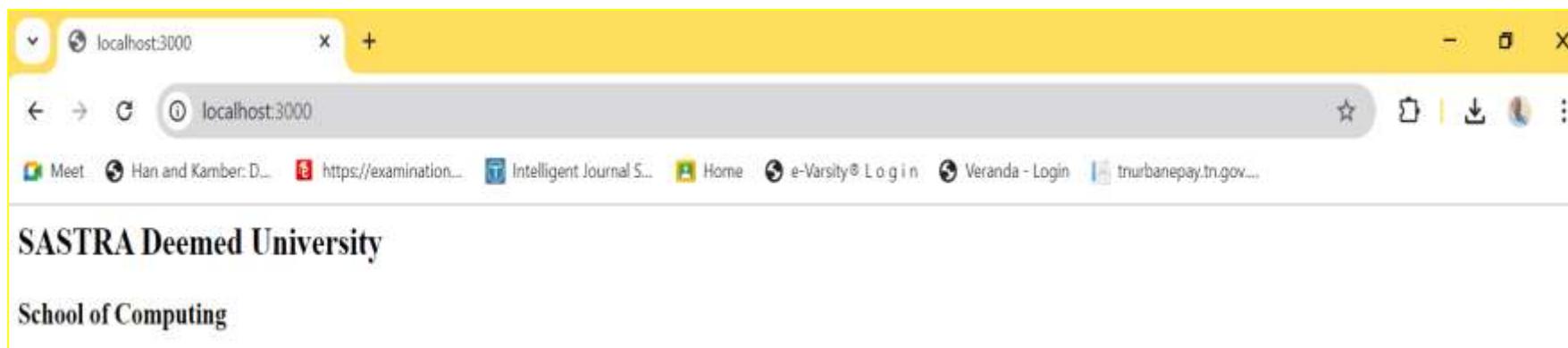
const port = process.env.port || 3000;
app.get("/", (req, res) => {
  res.send(`<div>
    <h2>SASTRA Deemed University</h2>
    <h3>School of Computing</h3>
  </div>`);
});
app.listen(port, () => {
  console.log(`Listening to port ${port}`);
});
```

# Output

```
PS F:\FullStack_VSC\middleware> npm start

  > middleware@1.0.0 start
  > nodemon index.js

[nodemon] 3.1.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
Listening to port 3000
```





# Revised Code

```
app.get(
  "/",
  (req, res, next) => {
    console.log("hello");
    next();
},
(req, res) => {
  res.send(
    `<div>
      <h2>ASTRA Deemed University</h2>
      <h3>School of Computing</h3>
    </div>`
  );
}
);
app.listen(port, () => {
  console.log(`Listening to port ${port}`);
});
```

# Output

```
PS F:\FullStack_VSC\middleware> npm start

> middleware@1.0.0 start
> nodemon index.js

[nodemon] 3.1.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
Listening to port 3000
hello
```



# Automatic Server Restart

- In order **to have the changes take effect**, you need to **restart** the server.
- You can stop the server using **Ctrl-C**, and start it **again** using **npm start**.
- But, if you are going to make many changes and test them often, this soon becomes a hassle.
- To automatically restart the server on changes, let's **install** and use the **package nodemon**.
- You may also use the package **forever** is another package that can be used to achieve the same goal.
- Typically, **forever** is used to restart the server on crashes rather than watch for changes to files.
- The best practice is to use **nodemon** during development (where you watch for changes) and **forever** on production (where you restart on crashes).

# Automatic Server Restart

- **npm install express nodemon**
- Since it is a local install, you need to use a script in package.json that tells npm to use nodemon instead of running the server directly.
- You can either modify the start command or add a new command.
- Here are the changes in package.json:

```
"scripts": {  
  "test": "echo \\\"Error: no test specified\\\" && exit 1",  
  "start": "nodemon index.js"  
}
```

# body-parser

- Adding a new Data
- **Create API – adding new Data**
- The resource URI for this API needs to be /api/student and the method needs to be a POST.
- The request body will contain the new issue object to be created.
  
- Express does not have an in-built parser that can parse request bodies to convert them into objects.
- So, we first need to install an npm package that helps us do that.
  
- `$ npm install body-parser –save`
  
- `const bodyParser = require('body-parser');`

- Need to test the newly created API.
- To test it, you can just type `http://localhost:3000/api/issues` in the browser's URL.
- You can use the command line utility curl, which will come in handy while testing HTTP methods other than GET (because you can only simulate a GET by typing in the URL).
- It may also be useful to install browser extensions or apps such as getpostman ([www.getpostman.com/](http://www.getpostman.com/)), a full-fledged API tester.
- The extension jsonview is also useful to see JSON output formatted nicely in the browser.

# References

- <https://www.geeksforgeeks.org/explain-the-use-of-req-and-res-objects-in-express-js/>
- <https://www.geeksforgeeks.org/middleware-in-express-js/?ref=lbp>





# SASTRA

ENGINEERING • MANAGEMENT • LAW • SCIENCES • HUMANITIES • EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



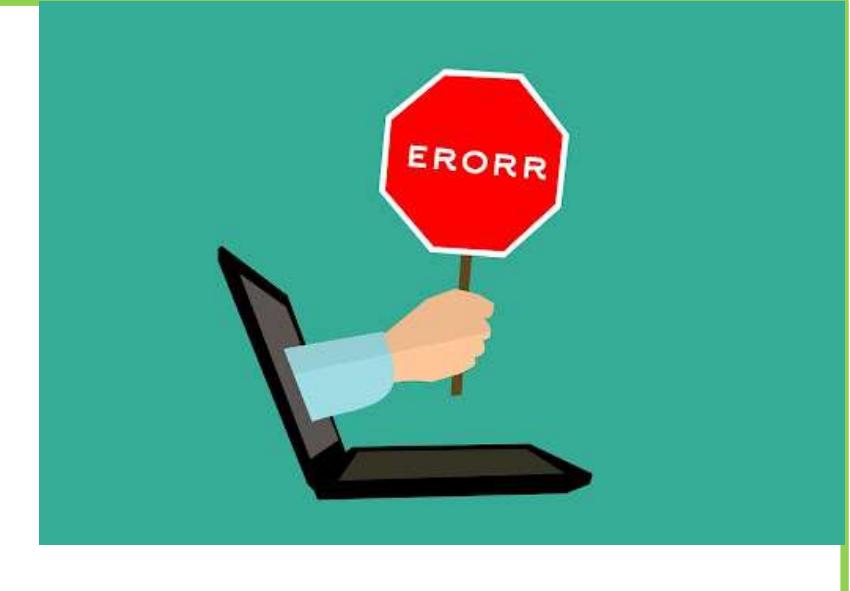
## INT436

# FULL STACK WEB APPLICATION DEVELOPMENT



Error handlers in Express js

- Error handling introduction
- HTTP Error codes
- Synchronous code error
- Asynchronous code error
- Ways to Perform Error Handling



- **Error Handling** refers to how Express catches and processes errors that occur both synchronously and asynchronously. Express comes with a **default error handler**
- Express catches all errors that occur while running route handlers and middleware functions.
- Middleware functions that have access to the request and response objects, as well as the next middleware function in the application's request-response cycle.

- ✓ *1xx informational response* – the request was received, continuing process
- ✓ *2xx successful* – the request was successfully received, understood, and accepted
- ✓ *3xx redirection* – further action needs to be taken in order to complete the request
- ✓ *4xx client error* – the request contains bad syntax or cannot be fulfilled
- ✓ *5xx server error* – the server failed to fulfil an apparently valid request

# Example HTTP codes

- **100 - continue**
- **101 – switching protocols**
- **200 –ok**
- **201 – created**
- **202 – accepted**
- **300 – multiple choices (redirection)**
- **307 – temporary redirect**
- **400 – Bad request**
- **403 – forbidden**
- **404 – not found**
- **500 Internal Server Error**
- **502 Bad Gateway**
- **503 Service Unavailable**

# Synchronous code error

- Errors that occur in synchronous code inside route handlers and middleware require no extra work. **If synchronous code throws an error, then Express will catch and process it.**
- For example

```
app.get('/', (req, res) => {
  throw new Error('BROKEN') // Express will catch this on its own.
})
```

# Asynchronous code error

For **errors returned from asynchronous functions** invoked by route handlers and middleware, you must **pass them to the next() function**, where Express will catch and process them. For example:

```
app.get('/', (req, res, next) => {
  fs.readFile('/file-does-not-exist', (err, data) => {
    if (err) {
      next(err) // Pass errors to Express.
    } else {
      res.send(data)
    }
  })
})
```

# Custom error handler

- Express has built-in error handling middleware, such as the `app.use(function(err, req, res, next) {})` function, which can be used to handle errors that are thrown or passed to the next() function.
- It is to handle own error handling middleware functions
- Error-handling middleware should be placed at the end of the **middleware stack**. so that it can catch any errors that are not handled by the other middleware.

# Reason for using error handling

- ✓ To prevent application crashes
- ✓ To provide meaningful error messages
- ✓ To improve debugging
- ✓ To comply with standards and regulations

# Ways to Perform Error Handling

- ✓ 1. Middleware function
- ✓ 2. Try-Catch statements
- ✓ 3. Error logging
- ✓ 4. HTTP status codes

# Middleware function

It is a Built-in support for error-handling middleware in express.js

Syntax:

```
app.use(function(error, request, response, next) {  
    // Handle the error  
    response.status(500).send('Internal Server Error');  
});
```

# Example

```
• const express = require('express');
• const app = express();
• // Custom error handling middleware
• const errorHandler = (err, req, res, next) => {
    console.error(err.stack);
    res.status(500).send('Something went wrong!');
};
• //error-handling middleware will catch any errors thrown
• //in the previous middleware or routes,
• //and handle them according to the logic defined
• //in the errorHandler function.
• app.use((req, res, next) => {
    throw new Error('Something broke!');
});
• );
```

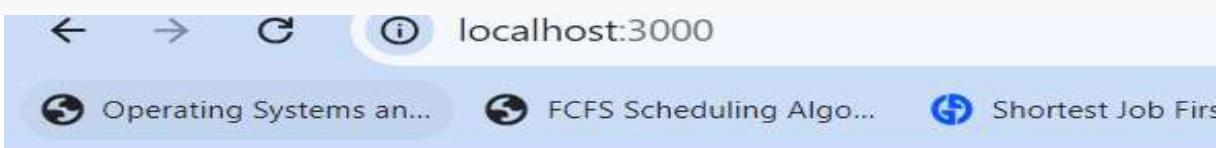
# Example

- // Use the custom error handling middleware
- app.use(errorHandler);
- app.listen(3000, () => {
- console.log('App is listening on port 3000');
- });

App is listening on port 3000

Error: Something broke!

```
at F:\fullstack\restfulapi\index.js:14:11
    at Layer.handle [as handle_request] (F:\fullstack\restfulapi\node_modules\express\lib\router\layer.js:95:5)
    at trim_prefix (F:\fullstack\restfulapi\node_modules\express\lib\router\index.js:328:13)
```



Something went wrong!

# Try-Catch statements

- Use try-catch statements to handle errors that occur **within specific blocks of code**.
- Errors are handled in **controlled manner**.

## Syntax:

```
app.get('/', function (req, res) {  
  try {  
    // Code that might throw an error  
  } catch (error) {  
    // Handle the error  
    res.status(500).send('Internal Server Error');  
  }  
});
```

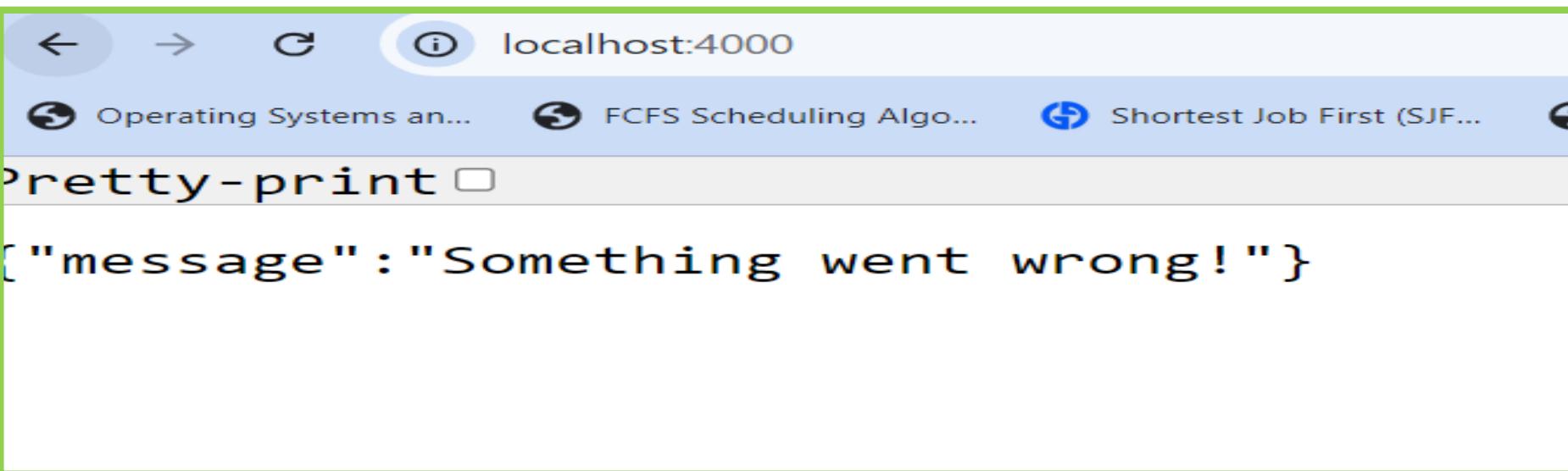
```
• const express = require('express');
• const app = express();
• app.get('/', (req, res, next) => {
•   try {
•     // Some code that might throw an error
•     const data = someFunctionThatMightThrowError();
•     res.status(200).json(
•       { message: 'Data retrieved successfully', data });
•   } catch (error) {
•     next(error);
•   }
• });

});
```

- // Custom error handling middleware
- app.use((err, req, res, next) => {
- console.error(err.stack);
- res.status(500).json(  
•         { message: 'Something went wrong!' });
- });
- app.listen(4000, () => {  
•     console.log('App is listening on port 4000');
- });
- if an error is thrown, it will be caught by the corresponding catch block, which logs the error stack and sends a response with a status code of 500 and the message "Something went wrong!".
- try-catch statement can be used multiple times in different middleware functions or routes

# Output

```
PS F:\fullstack\restfulapi> node index1.js
App is listening on port 4000
ReferenceError: someFunctionThatMightThrowError is not
defined
    at F:\fullstack\restfulapi\index1.js:6:22
        at Layer.handle [as handle_request] (F:\fullstack\r
estfulapi\node_modules\express\lib\router\layer.js:95:5
```



A screenshot of a web browser window. The address bar shows "localhost:4000". Below the address bar, there are three tabs: "Operating Systems an...", "FCFS Scheduling Algo...", and "Shortest Job First (SJF...)".

The main content area displays a JSON object:

```
{"message": "Something went wrong!"}
```

# Error logging

- You can set up error logging so that any errors that occur during the execution of the application are logged to a file or a database for later analysis.

## Syntax:

```
app.get('/', function (req, res) {  
  try {  
    throw new Error('Something went wrong');  
  } catch (error) {  
    console.error(error);  
  }  
});
```

# Error codes:

- Error codes in Node.js are **symbolic values** that represent specific types of errors that can occur during the execution of a program.
- It usually represented as strings and are used to help identify and categorize different types of errors.
- For example
- Node.js fs module uses error codes such as ‘ENOENT’ (no such file or directory) or ‘EACCES’ (permission denied) to represent specific types of file system errors.

```
1 const fs = require('fs');
2
3 fs.readFile('nonexistent-file.txt', (error, data) => {
4   if (error) {
5     console.error(error.code);
6   } else {
7     console.log(data.toString());
8   }
9 });


```

[PROBLEMS](#)[OUTPUT](#)[DEBUG CONSOLE](#)[TERMINAL](#)[PORTS](#)[POSTMAN CONSOL](#)

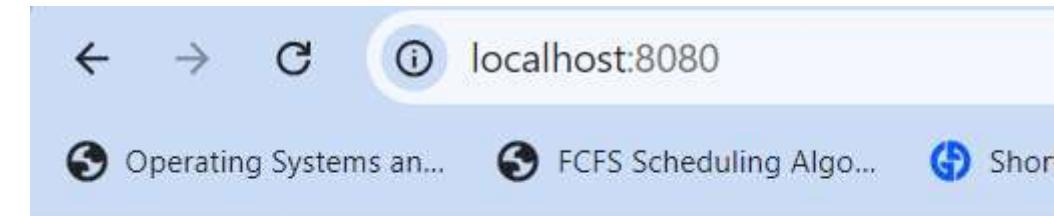
```
PS F:\fullstack\restfulapi> node index2.js
ENOENT
```

# HTTP status codes

- Use HTTP status codes to indicate the type of error that occurred. For example, a status code of 400 (Bad Request) can indicate a validation error, while a status code of 500 (Internal Server Error) can indicate a server-side error.

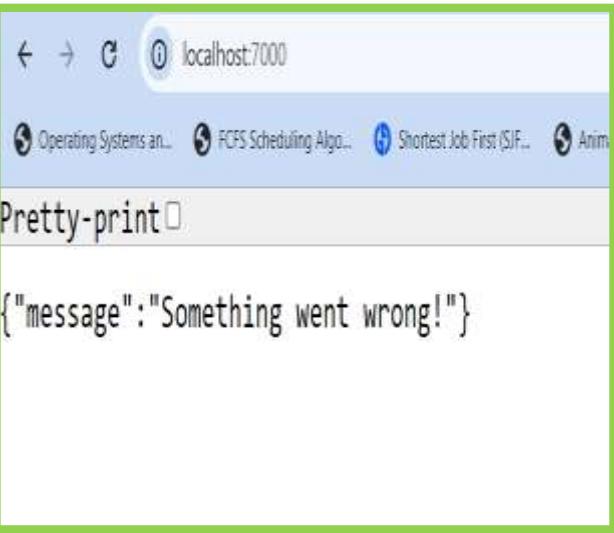
```
const server = http.createServer((request, response) => {
  response.statusCode = 200;
  response.end('OK');
});

server.listen(8080);
```



# Using next()

- You can use the `next()` function to pass errors to the next middleware function in the chain.



A screenshot of a web browser window. The address bar shows 'localhost:7000'. The page content is a JSON object: {"message": "Something went wrong!"}. The entire screenshot is highlighted with a green rectangular border.

```
{"message": "Something went wrong!"}
```

```
const express = require('express');
const app = express();
app.get('/', (req, res, next) => {
    // Some code that might throw an error
    const data = someFunctionThatMightThrowError();
    if (!data) {
        return next(new Error('Error retrieving data'));
    }
    res.status(200).json(
        { message: 'Data retrieved successfully', data });
})
// Custom error handling middleware
app.use((err, req, res, next) => {
    console.error(err.stack);
    res.status(500).json(
        { message: 'Something went wrong!' });
});
app.listen(7000, () => {
    console.log('App is listening on port 7000');
});
```

```
PS F:\fullstack\restfulapi> node index4.js
App is listening on port 7000
ReferenceError: someFunctionThatMightThrowError is not defined
  at F:\fullstack\restfulapi\index4.js:6:18
  at Layer.handle [as handle_request] (F:\fullstack\restfulapi\node_modules\express\lib\router\layer.js:95:5)
    at next (F:\fullstack\restfulapi\node_modules\express\lib\router\route.js:149:13)
  at Route.dispatch (F:\fullstack\restfulapi\node_modules\express\lib\router\route.js:119:3)
```

- ✓ **Increased complexity** - add complexity to your application to anticipate potential errors
- ✓ **Overhead** - additional logic and processing to capture and respond to errors . This can impact the performance of your application.
- ✓ **False alarms** - If error handling is not implemented correctly, it can lead to false alarms
- ✓ **Security risks** - Improper error handling can create security risks. it can reveal sensitive information or provide attackers with clues about potential vulnerabilities in your application.

- <https://www.geeksforgeeks.org/explain-error-handling-in-express-js-using-an-example/>
- <https://expressjs.com/en/guide/error-handling.html>
- [https://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_status\\_codes](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes)

THANK YOU



# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

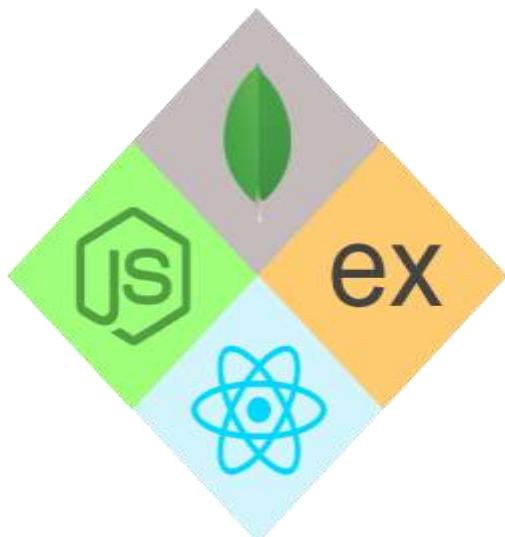
(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



## INT436

# FULL STACK WEB APPLICATION DEVELOPMENT





- Routing in React
- React router
- Dynamic routing
- Programmatic navigation
- No matching route



# What is routing?

- Routing in react is the mechanism by which we **navigate through a website** or web-application **by entering a URL or clicking on an element.**
- Routing can be manually implemented using **useState** and **JSX for conditioning.**
- But being inefficient for large-scale applications **like e-commerce.**
- Routing can be **server-side routing** or **client-side routing.**
- Server-side routing results in
  - every request being a full page refresh
  - loading unnecessary and repeated data every time
  - causing a delay in loading.

# What is routing?

- With client-side routing,
  - it could be the **rendering** of a new component
  - where routing is handled internally by JavaScript.
  - The component is rendered without the server being reloaded.
- React makes use of an **external library** to handle routing
- **Browser Router or React Router API** is a most popular library for routing in react to navigate among different components in a React Application keeping the UI in alignment with URL.

# Various Packages in React Router Library

- There are 3 different packages for React Routing.
  - **react-router**:
    - It is the heart of react-router and provides **core functionalities** for routing in **web applications**
  - **react-router-native**:
    - It is used for routing in **mobile applications**
  - **react-router-dom**:
    - It is used to implement **dynamic routing** in **web applications**.

# Different Types of Routers in React Router?

- React Router provides three different kinds of routers.
  - Memory Router
  - Browser Router
  - Hash Router

```
import { MemoryRouter as Router } from 'react-router-dom';
```

- The memory router keeps **the URL changes** in **memory** not in the user browsers.
- It keeps the **history of the URL** in **memory** and it does not read or write to the address bar so the user can not use **the browser's back button** as well as **the forward button**.
- It doesn't change the URL in your browser.
- It is very **useful for testing** and non-browser environments like React Native.



A screenshot of a web browser window titled "localhost:3000". The address bar shows "localhost:3000". Below the address bar is a navigation bar with icons for back, forward, refresh, and search. The main content area displays a list of links:

- Apps
- What is Google Clo...
- jQuery hits
- GraphQL Tutorial #...
- 9.7. itertools — Fun...

Below this is a sidebar menu with the following items:

- Home
- About Us
- Contact 

At the bottom of the page, a banner reads: "GeeksforGeeks is a computer science portal for geeks!"

```
import { BrowserRouter as Router } from 'react-router-dom';
```

- It uses [HTML 5 history API](#) (i.e. pushState, replaceState, and popState API) to keep [your UI in sync with the URL](#).
- [It routes as a normal URL in the browser](#) and assumes that the server is handling all the request URL



A screenshot of a web browser window. The address bar shows "localhost:3000/contact". The page content includes a sidebar menu with three items: "Home", "About Us", and "Contact Us". Below the sidebar, there is a section titled "You can find us here:" followed by the text "GeeksforGeeks" and "5th & 6th Floor, Royal Kapsons, A- 118, Sector- 136, Noida, Uttar Pradesh (201305)". The browser interface also shows other tabs and icons at the top.

- Home
- About Us
- Contact Us

You can find us here:  
GeeksforGeeks  
5th & 6th Floor, Royal Kapsons, A- 118,  
Sector- 136, Noida, Uttar Pradesh (201305)

```
import { HashRouter as Router } from 'react-router-dom';
```

- Hash router uses **client-side hash routing**.
- It uses the **hash portion of the URL** to keep your UI in sync with the URL.
- The hash portion of the URL won't be handled by the server, the **server will always send the index.html** for every request and ignore the hash value.
- It is used to **support legacy browsers** which usually don't support **HTML pushState API**.



A screenshot of a web browser window. The address bar shows "localhost:3000/#/about". Below the address bar, there are several tabs: "Apps", "What is Google Clo...", "jQuery hits", "GraphQL Tutorial #...", and "9.7. itertools — Fun...". The main content area displays a navigation menu with three items: "Home", "About Us", and "Contact Us". A cursor is hovering over the "Contact Us" link. At the bottom of the page, the text "GeeksforGeeks is a computer science portal for geeks!" is displayed.

# How to Install React Router?

- React makes use of an external library to handle routing.
- First install it
- **Step 1:** Run the following commands in terminal
- `npm install react-router-dom@6`
- or
- `yarn add react-router-dom@6`

# How to Install React Router?

- After the completion of npm
- A message is received on the terminal varying with your system architecture.

```
+ react-router-dom@6
```

added 11 packages from 6 contributors  
and audited 1981 packages in 24.897s

114 packages are looking for funding  
run `npm fund` for details

found 0 vulnerabilities

added 3 packages, and audited 1513 packages in 6s

245 packages are looking for funding  
run `npm fund` for details

6 high severity vulnerabilities

To address all issues (including breaking changes), run:  
npm audit fix --force

Run `npm audit` for details.

# Source Code and Snippets

- After the installation of react-router-dom we can ensure that the package is successfully installed or not by checking the **package.json** file to see the installed react-router-dom module and its version.

```
{  
  "name" : "reactApp",  
  "version" : "1.0.0"  
  "description" : "It is.a react app"  
  "dependencies" : {  
    "react" : "^17.0.2",  
    "react-dom" : "^17.0.2",  
    "react-icons" : "^4.3.1",  
    "react-router" : "^6.2.1",  
    "react-router-dom" : "^6.2.1"  
  },
```

```
{  
  "name": "unit3demo",  
  "version": "0.1.0",  
  "dependencies": {  
    "@testing-library/jest-dom": "^5.17.0",  
    "@testing-library/react": "^13.4.0",  
    "@testing-library/user-event": "^13.5.0",  
    "react": "^18.2.0",  
    "react-dom": "^18.2.0",  
    "react-router-dom": "^6.15.0",  
    "react-scripts": "5.0.1",  
    "web-vitals": "^2.1.4"  
  }  
}
```

# How to Setup React Router?

- To configure React router, **navigate to the index.js file**, which is the root file, **and import BrowserRouter** from the react-router-dom package that we installed, wrapping it around our App component as follows:

```
// index.js
import React from 'react';
import ReactDOM from 'react-dom/client';
import { BrowserRouter } from 'react-router-dom';
import App from './App';
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>
);
```

# How to Configure Routes In React?

- We have now successfully installed and imported React router into our project;
  - the next step is to use React router to implement routing.
- The first step is to **configure all of our routes** (that is, all the pages/components to which we want to navigate).
- Now, we can make **directories for the components** or the pages we want to render.
- Either you **can make separate folders or can have one folder** with all components.

# How to Configure Routes In React?

- We would first create those components, in our case two pages:
  - the [Home page](#) and
  - the [About Page](#).
- `mkdir src/components/Home`
- `mkdir src/components/About`
- Now we will create a component inside each directory we created above.
- Here we will first create a [Home.js file](#) for the Home directory.
- `src/components/Home/Home.js`

# How to Configure Routes In React?

- Then add the basic component rendering code for it.

src/components/Home/Home.js

```
import React from "react";
function Home() {
  return (
    <div>
      <h1> This is the home page </h1>
    </div>
  );
}
export default Home;
```

src/components/About/About.js

```
import React from "react";
function About() {
  return (
    <div>
      <h1> This is the about page </h1>
    </div>
  );
}
export default About;
```

# How to Configure Routes In React?

- Now come to the **main App.js file** which is the core of implementing all we have defined and declared till now by defining routes for each **component** and where and which **component** they will render when the path matches with the base URL entered or clicked by the user.

```
import {Routes , Route } from "react-router-dom"
import Home from "./components/Home/Home"
import About from "./components/About/About"
function App(){
  return (
    <div className="App">
      <Routes>
        <Route path="/" element={<Home/>} />
        <Route path="/about" element={<About/>} />
      </Routes>
    </div>
  )
}
export default App
```

# Routes Tag

- In the above code (Previous slide) that we imported Routes and Route components from react-router-dom and then used them to declare the routes we want.
- All Routes are wrapped in the Routes tag, and these Routes have two major properties.
- **path:**
  - As the name implies, this identifies the path we want users to take to reach the set component.
- **element:**
  - This contains the component that we want the set path to load.
  - This is simple to understand, but remember to import any components we are using here, or else an error will occur.

# How to Access Configured Routes with Links?

- How we could manually access these routes via the URL?
- How we can create links within our application so that users can easily navigate?
- This is accomplished with the **Link tag**, just as it is with the **<a> tag in HTML**.

```
// App.js
import NavBar from './Components/NavBar';
const App = () => {
  return (
    <>
      <Routes>
        // ...
      </Routes>
    </>
  );
};
export default App;
```

# How to Access Configured Routes with Links?

```
import {Link} from 'react-router-dom';

const NavBar = () => {
  return (
    <nav>
      <ul>
        <li>
          <Link to="/">Home</Link>
        </li>
        <li>
          <Link to="/about">About</Link>
        </li>
        <li>
          <Link to="/products">Products</Link>
        </li>
      </ul>
    </nav>
  );
};

export default NavBar;
```

- First import Link from react-router-dom
- Then added the **to** property based on the path we specified while configuring our routes.
- This is how simple it is to add links to our React application
- Allowing us to access configured routes.

# How to Implement Active Links?

- When adding links to a navbar,
  - we want users **to be able to identify the exact link** they are currently navigating to
  - by giving it a **different color** than other nav links,
  - adding extra **styles like underline**, and so on.
- It is easier to handle and implement the **active link** feature in React by using the **NavLink component** rather than the Link component.

# How to Implement Active Links?

```
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="/products">Products</NavLink>
        </li>
      </ul>
    </nav>
  \\\);
};

export default NavBar;
```

# Review Question

- What is the Difference Between React Router and React Router DOM?

- These two seem identical.
- react-router is the core npm package for routing.
- but react-router-dom is the superset of react-router providing additional components like BrowserRouter, NavLink and other components, it helps in routing for web applications.

# References

- <https://hygraph.com/blog/routing-in-react>
- <https://www.knowledgehut.com/blog/web-development/routing-in-reactjs>
- <https://www.freecodecamp.org/news/a-complete-beginners-guide-to-react-router/include-router-hooks/>
- <https://learnwithparam.com/blog/different-types-of-router-in-react-router/>
- <https://medium.com/front-end-weekly/choosing-a-router-in-react-apps-85ae72fe9d78>
- <https://www.geeksforgeeks.org/reactjs-types-of-routers/>





# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

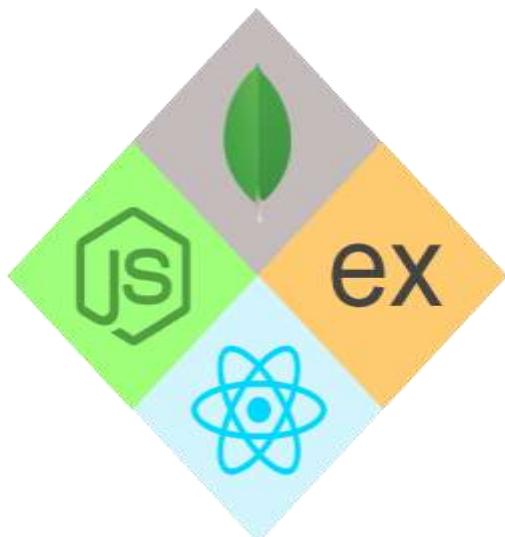
(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



## INT436

# FULL STACK WEB APPLICATION DEVELOPMENT





➤ Dynamic routing



# Review Question

- What is the Difference Between React Router and React Router DOM?

- These two seem identical.
  - `react-router` is the core npm package for routing.
  - But `react-router-dom` is the superset of `react-router` providing additional components like `BrowserRouter`, `NavLink` and other components, it helps in routing for web applications.

# How to fix No Routes Found Error?

# How to fix No Routes Found Error?

- When routing,
  - a situation may cause a user to access an unconfigured route
  - or a route that does not exist.
- When this occurs,
  - React does not display anything on the screen except a warning with the message "**No routes matched location.**"
- This can be fixed by **configuring a new route to return a specific component** when a user navigates to an unconfigured route.

# How to fix No Routes Found Error?

```
// App.js
import { Routes, Route } from 'react-router-dom';
import NoMatch from './Components/NoMatch';

const App = () => {
  return (
    <>
      <Routes>
        // ...
        <Route path="*" element={<NoMatch />} />
      </Routes>
    </>
  );
};

export default App;
```

- we created a route with the **path \*** to get all non-configured paths and assign them to the attached component.

- We created a component called **NoMatch.js**, but you can name yours whatever you want to **display 404, page not found**, on the screen, so users know they are on the wrong page.
- We can also **add a button** that takes the **user to another page or back**, which leads us to programmatic navigation.

# Use of exact prop in react routing

# Use of exact prop

- The exact param comes into play when you have **multiple paths** that **have similar names**
- For example, imagine we had a **Users component** that **displays a list of users**.
- We also have a **CreateUser component** that is **used to create users**.
- The URL for **CreateUser** should be nested under **Users**.
- So our setup could look something like this.

```
<Routes>
  <Route path="/users" component={Users} />
  <Route path="/users/create" component={CreateUser} />
</Routes>
```

# Use of exact prop

- Now the problem here is,
  - when we go to <http://app.com/users>
    - the router will go through all of our defined routes and **return the FIRST match it finds.**
    - So in this case, it would find the **Users route first** and then return it.
- But, when we go to <http://app.com/users/create>,
  - it would **again** go through all of our defined routes and return the FIRST match it finds.
  - React router does partial matching, so /users partially matches /users/create
  - so it would **incorrectly return the Users route again!**

# Use of exact prop

- The **exact** param disables the partial matching for a route and makes sure that it only returns the route if the path is an EXACT match to the current url.
- So in this case, we should add **exact** to our Users route so that it will only match on /users:

```
<Routes>
  <Route exact path="/users" component={Users} />
  <Route path="/users/create" component={CreateUser} />
<Routes>
```

# Static and Dynamic routing

# Static and Dynamic routing

- Earlier versions of `react-router` and frameworks like Angular, Ember use static routing.
- In static routing,
  - you need to **define all the routes** in a centralized location in your application.
  - Then these routes **will be imported** to the top level of the application before it starts rendering.

# Static and Dynamic routing

- If we consider Angular as an example, **its app-routing.module.ts** file will contain all the routes and corresponding components.

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HeroesComponent } from './heroes/heroes.component';
const routes: Routes = [
  { path: 'component1', component: Component1 },
  { path: 'component2', component: Component2 },
  { path: 'component3', component: Component3 },
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Then this file is imported to `app.module.ts` before application rendering takes place.

# Dynamic routing

- Dynamic routing is a routing method used in web development where the routes of an application are determined dynamically.
- Unlike **static** routing, where each route is **predefined and fixed**, dynamic routing allows routes to be **generated on the fly** based on user input, application state, or other factors.
- Dynamic routing can significantly improve the user experience.
- By delivering the right content at the right time, it ensures that users always have access to the **most relevant and up-to-date information**.

# Static and Dynamic routing

- The main difference between static vs. dynamic routing is the time at which the routing takes place.
- In dynamic routing
  - routes are initialized dynamically when the page gets rendered.
- Dynamic routing allows rendering your React components conditionally.

# Dynamic routing

- Dynamic routing protocols are the rules or standards that govern how routers communicate with each other to disseminate information that **allows them to select routes between any two nodes** on a computer network.
- In the context of frontend development, however, the term "dynamic routing protocols" takes on a slightly different meaning.
- In frontend development,
  - dynamic routing protocols define
    - how routes are structured,
    - how route parameters are handled, and
    - how routes are matched and rendered.

# Dynamic routing

- In React Router,
  - Routes are defined using the `Router` component
  - which takes a `path` prop to specify the route's pattern and
  - a `component` prop to specify the component to be rendered when the route is matched.
- **Route parameters** can be included in the **path using a colon (:)** followed by **the parameter name**, and they can be **accessed** in the component using the `match.params` object.

```
<Route path="/user/:id" component={User} />
```

- In this example, the '`:id`' in the path is a route parameter that matches any string.
- When a route like `/user/123` is accessed, the `User` component is rendered, and the `id` parameter can be accessed as `match.params.id`.

# Dynamic routing

- Let's assume we're building a **blog application** and we want to create a dynamic route for individual blog posts.
- First, we'll add a new **<Route>** component for the blog post route.
- This route will include a route parameter **(:id)** to represent the ID of the blog post.

```
<Route path="/post/:id" component={Post} />
```

# Dynamic routing

- Next, we'll create the **Post** component.
- This component will receive the `match` prop from React Router, which includes the `params` object.
- The `params` object contains the `values of the route parameters`, so we can use `match.params.id` to access the ID of the blog post:

```
function Post({ match }) {  
  return <h2>Post ID: {match.params.id}</h2>;  
}
```

# Dynamic routing

- Finally, we'll add a few `<Link>` components to navigate to different blog posts:

```
<Link to="/post/1">Post 1</Link>
<Link to="/post/2">Post 2</Link>
<Link to="/post/3">Post 3</Link>
```

- We've implemented dynamic routing in our React application.
- Now, when you navigate to `/post/1`, `/post/2`, or `/post/3`, the `Post` component will be rendered with the corresponding post ID.

# Nested Dynamic Routes

- Nested routes are a powerful feature of React Router that allows you to **create complex routing structures** with ease.
- **A nested route is a route that's defined inside another route**, allowing you to create hierarchical relationships between your routes.
- For example,
  - let's say you're building a blog application and
  - you want to create a **separate route for the comments** of each blog post.
  - You could do this by defining a **nested route inside the Post component**

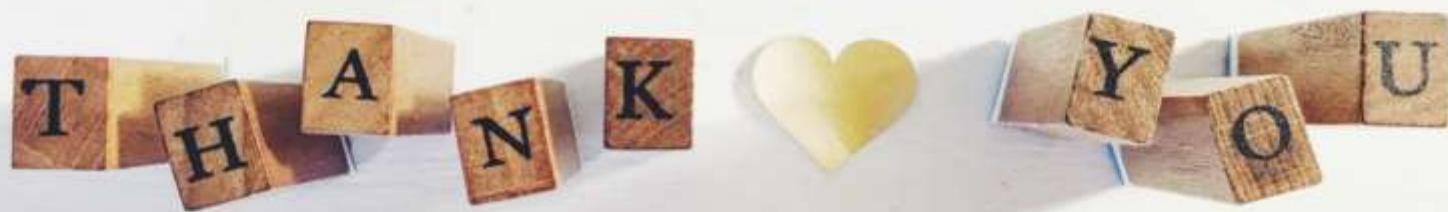
# Nested Dynamic Routes

```
function Post({ match }) {  
  return (  
    <div>  
      <h2>Post ID: {match.params.id}</h2>  
      <Route path={`${match.path}/comments`} component={Comments} />  
    </div>  
  );  
}
```

- In this example, the `Comments` component will be rendered when you navigate to a URL like `/post/1/comments`.
- The ``${match.path}/comments` expression` is used to append `/comments` to the current path, creating a nested route.

# References

- <https://dev.to/shubhamtiwari909/dynamic-routes-in-react-491g>
- <https://blog.bitsrc.io/dynamic-vs-static-routing-in-react-49730baaf3e9>
- <https://www.dhiwise.com/post/dynamic-routing-for-building-flexible-and-scalable-react-apps#:~:text=In%20frontend%20development%2C%20on%20the,or%20a%20user%20profile%20page.>





# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

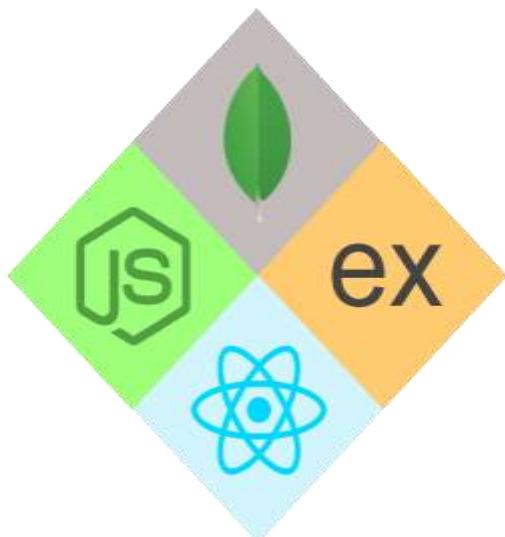
(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



## INT436

# FULL STACK WEB APPLICATION DEVELOPMENT





- Programmatic navigation
- Lazy Loading



- While the `<Link>` component is great for creating links to different routes
  - sometimes you might need to navigate programmatically
  - for example, in **response to a button click or a form submission.**
- React Router provides the **history prop** for this purpose
  - which includes **several methods** for programmatically navigating between routes.
- The most commonly used method is **history.push()**
  - which pushes a new entry onto the history stack, effectively navigating to a new route.

```
function Post({ history }) {  
  return (  
    <div>  
      <h2>Post</h2>  
      <button onClick={() => history.push('/about')}>Go to About</button>  
    </div>  
  );  
}
```

- In this example,
  - when the button is clicked,
  - the application will navigate to the **/about** route.

# React Router Hooks

- The React Router comes with a few hooks that allow us to programmatically access the Router State and navigate between components.
  - `useHistory()`
  - `useNavigate()`
  - `useLocation()`
  - `useParams()`
- To use any of these hooks,
  - we must first import them from the `react-router-dom` package
  - then specify a variable that invokes the hook

# useHistory()

- The **useHistory()** hook
  - Provides direct access to React Router's history instances
  - Enabling us to perform actions like
    - retrieving the number of entries in the history stack,
    - adding,
    - altering, or removing an entry from the stack.

- Some of the most useful methods here include:
  - `goBack()` - Go backward in history.
  - `goForward()` - Go forward in history.
  - `push()` - Add a new entry to the history stack, i.e., navigate to a new URL.
  - `replace()` - Similar to `push()` in that it replaces the current stack in the history, but unlike `push()`, **the user cannot travel back in history**, i.e. clicking the browser back button will not return to the previous state.

# useHistory()

```
import React from "react";
import { useHistory } from "react-router-dom";

const About = () => {
  let history = useHistory();

  const goHome = () => {
    history.push("/");
  };

  return (
    <div>
      <h1> About page here! </h1>
      <p> useHistory Demo </p>
      <button onClick={goHome}>Go to home page</button>
    </div>
  );
};

export default About;
```

- we import the `useHistory()` hook and create a new `goHome()` function that executes on a button click.
- It's just a wrapper for a `push()` call.

- The `useHistory()` hook has been deprecated in the latest version of React Router.
- If you're using React Router V6, you'll want to [use the `useNavigate\(\)` hook instead](#).

# useNavigate()

- If you're using the most recent version of React Router,
  - the `useHistory()` hook has been deprecated in favor of `useNavigate()`.
- The approach is nearly identical.
- The main difference is that the `useNavigate()` hook does not accept methods like `.push()` or `.replace()`.
- You just `navigate()` to a certain entry.

```
import React from "react";
import { useNavigate } from "react-router-dom";

const About = () => {
  let navigate = useNavigate();

  const goHome = () => {
    navigate("/");
  };

  return (
    <div>
      <h1>About page here!</h1>
      <button onClick={goHome}>Go to home page</button>
    </div>
  );
};

export default About;
```

```
const GoBack = () => {
  const navigate = useNavigate();
  const handleClick = () => navigate(-1);
  return (
    <button type="button" onClick={handleClick}>
      Go Back ▶
    </button>
  );
};
```

```
const GoForward = () => {
  const navigate = useNavigate();
  const handleClick = () => navigate(1);

  return (
    <button type="button" onClick={handleClick}>
      Go Forward
    </button>
  );
};
```

# How to get the necessary information about the current route?

# useLocation()

- The `useLocation()` hook provides us access to the browser's `location` object.
- Use this hook to obtain the necessary information about the current route.

```
import { useNavigate, useLocation } from "react-router-dom";
/*...*/
let location = useLocation();
console.log(location);
```

```
{
  "pathname": "/about",
  "search": "?",
  "hash": "#",
  "state": null,
  "key": "default"
}
```

# userParams()

- the `userParams()` hook
  - can be used to get the value of URL parameters.
- When called, `useParams()` provides `an object` that maps the names of URL parameters to their values in the current URL.

In our router configuration, say we've had a dynamic route.

```
<Route path="/about/:user_id">  
  <About />  
</Route>
```

And on another page, we have a link component that points to some information pertaining to User 2.

```
<Link to="/about/2">About User 2</Link>
```

# userParams()

```
import { useParams } from "react-router-dom";

const About = () => {
  const { user_id } = useParams();

  return (
    <div>
      <h1>About user {user_id}</h1>
      <p>
        useParams() demo
      </p>
    </div>
  );
};

export default About;
```

When parameters are passed - we can access the parameters via the `useParams()` hook.

# Lazy Loading With React

# Lazy Loading With React

- When we launch a React web application
  - it usually **bundles** the entire application at once
  - loading everything including the entire web app pages, images and content
  - potentially results in a **slow load time and overall poor performance**, depending on the size of the content and the Internet bandwidth at the time.
- **lazy loading allows us to render components or elements on demand**
  - Making our app more efficient and providing a better user experience.

# How to Implement Lazy Loading in React?

- To implement lazy loading in our React applications
    - use two React features - `React.lazy()` and `React.Suspense`.
  - `React.lazy()` is a function that allows us to render **dynamic imports** in the same way as regular components.
  - Using dynamic imports alongside the `React.lazy()` will enable us to import a component just before it renders on a screen.
  - An important thing to note is that `React.lazy()` accepts a function as an argument
    - that function must call the `dynamic import()` in its body.
  - `React.Suspense` enables us to specify the **fallback prop(Alternate UI)** which takes in a **placeholder content** that would be used as a loading indicator while all the lazy components get loaded.

# Lazy Loading With React

```
import React from 'react';

// Lazy loading
const AboutUs = React.lazy(() => import('./About'));

const Home = () => {
  return (
    <div className="App">
      <h1>Home Page</h1>
      <AboutUs />
    </div>
  );
};

export default Home;
```

- Code above will always throw an error saying that our “**React component suspended while rendering, but no fallback UI was specified**”.
- This can be fixed by [wrapping the component with React.Suspense](#)

# Lazy Loading With React

```
import React from 'react';
const AboutUs = React.lazy(() => import('./About'));

const Home = () => {
  return (
    <div className="App">
      <h1>Home Page</h1>
      <React.Suspense fallback=<div>Loading...</div>>
        <AboutUs />
      </React.Suspense>
    </div>
  );
};
export default Home;
```

```
import { lazy, Suspense } from 'react';
import { BrowserRouter, Routes, Route } from 'react-router-dom';

const Home = lazy(() => import('./Home'));
const Products = lazy(() => import('./Products'));

function App() {
  return (
    <BrowserRouter>
      <Suspense fallback={<div>Loading...</div>}>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/products" element={<Products />} />
        </Routes>
      </Suspense>
    </BrowserRouter>
  );
}
export default App;
```

# References

- <https://www.dhiwise.com/post/dynamic-routing-for-building-flexible-and-scalable-react-apps#:~:text=In%20frontend%20development%2C%20on%20the,or%20a%20user%20profile%20page.>
- <https://stackabuse.com/programmatically-navigate-using-react-router/>
- <https://ultimatecourses.com/blog/programmatically-navigate-react-router>
- <https://www.pluralsight.com/guides/using-react-with-the-history-api>

# References

- <https://www.aleksandrkhannisan.com/blog/react-lazy-dynamic-imports/#:~:text=Unlike%20static%20imports%2C%20React's%20dynamic,as%20the%20app%20is%20running.>
- <https://stackabuse.com/guide-to-lazy-loading-with-react/>
- <https://www.positronx.io/react-lazy-loading-router-using-react-router-dom-tutorial/>
- <https://linguinecode.com/post/code-splitting-react-router-with-react-lazy-and-react-suspense>





# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

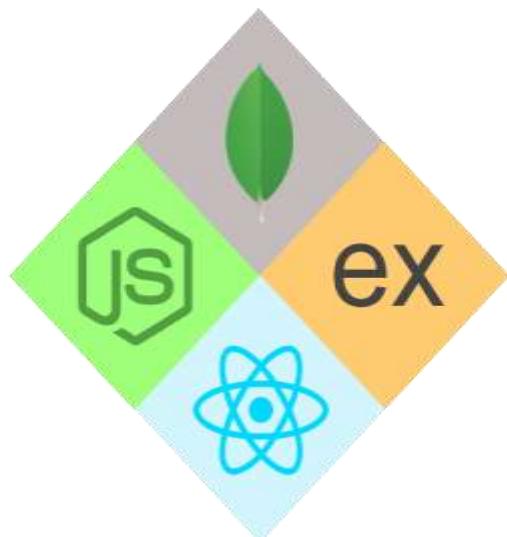
(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



## INT436

# FULL STACK WEB APPLICATION DEVELOPMENT





### New Order

Name

Address

Date  
 mm / dd / yyyy

Order Number

# Introduction

- It's nearly impossible to **create an application or a website**
  - without eventually needing a way to collect information from your users.
- This could be as simple as
  - a couple of **text inputs to get their name and email address** or
  - it could be a complex, multi-page situation requiring several distinct types of inputs, state management and more.
- **React uses forms to allow users to interact with the web page.**

# Types of Forms in React

- There are two types of forms:
- **Controlled Input**
  - A react form is considered to be controlled when a react component that is responsible for rendering is also controlling the form behavior on subsequent inputs.
  - That means whenever values in form changes, the component saves the changed value to its state.

```
changeEventHandler = event => { this.setState({  
    username: event.target.value  
}); }
```

- Value of username changes, the change event handler is called and its updated value is saved to state.
- Controlled form can be used for applying validations, disabling a button until a text field contains some text, etc.

# Types of Forms in React

- **Uncontrolled Forms**
- Uncontrolled forms are similar to HTML forms.
- This does not make use of any listener.
- Get the value of the field - for example on click of a button.
- The required value is read using a reference associated with the form elements.

```
<form onSubmit={this.submitHandler}>  
  <div>  
    <input type="text" value="valueref" ref="valueref" />  
  </div>  
</form>
```

# Adding Forms in React

- You can **add a form** in React like all other elements.

```
function MyForm() {  
  return (  
    <form>  
      <label>Enter your name:  
        <input type="text" />  
      </label>  
    </form>  
  )  
}
```

- If we run the code above, **the page will reload and the form will be sent.**
- But in React, **this is typically not what we want to happen.**
- To let React handle the form, **we want to eliminate this default behaviour.**

# Handling Forms

- Form handling refers to **how you deal with the data when it is submitted or the value changes.**
- In HTML, form data is usually **handled by the DOM.**
- In React, **form data is usually handled by the components.**
- By including event handlers in the **onChange attribute**, you can manage changes.
- The **useState Hook** can be used to keep track of each input value.

# Example

```
import { useState } from 'react';

function MyForm() {
  const [name, setName] = useState("");

  return (
    <form>
      <label>Enter your name:</label>
      <input
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
      </label>
    </form>
  )
}
```

# Submitting Forms

- You can control the submit action by adding an event handler in the **onSubmit** attribute for the <form>.

```
import { useState } from 'react';

function MyForm() {
  const [name, setName] = useState("");

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`The name you entered was: ${name}`)
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>Enter your name:</label>
      <input type="text" value={name}
        onChange={(e) => setName(e.target.value)}>
      </input>
      <input type="submit" />
    </form>
  )
}
```

# **Creating Forms in React using react-hook-form**

- Creating forms in React is a **complex task**.
- It involves handling **all the input states and their changes and validating that input** when the form gets submitted.
- But as your form gets **more complex** and you need to add various **validations**, it becomes a complicated task.
- So instead of **manually writing all of the code and handling complex forms with validation logic**, we can use the most popular React library for this, **react-hook-form**.
- It's the most popular React library for creating forms compared to **formik** and **react final form**.

## why react-hook-form is a popular choice for creating React forms.

- The **number of re-renders in the application is smaller** compared to the alternatives because it uses refs instead of state.
- The **amount of code you have to write** is less as compared to **formik**, **react-final-form** and other alternatives.
- **react-hook-form integrates well with the `yup` library for schema validation** so you can combine your own validation schemas.
- **Mounting time is shorter** compared to other alternatives.

- **Yup is a JavaScript schema builder, parser, and validator.**
- You define an object schema, which you can then use to either validate the shape of an existing value or transform an existing value to match the schema.

# How to Install react-hook-form?

- To install the react-hook-form library, execute the following command from the terminal:
- **npm install react-hook-form@7.38.0**
- OR
- **yarn add react-hook-form@7.38.0**

- The react-hook-form library provides a **useForm hook** which we can use to work with forms.
- Import the **useForm**
- **import { useForm } from 'react-hook-form';**
- You can use the **useForm hook** like this:

```
const {  
  register, handleSubmit, formState: { errors },  
} = useForm();
```

- Here,
  - **register is a function provided by the useForm hook.**
    - We can assign it to each input field so that the react-hook-form **can track the changes** for the input field value
  - **handleSubmit is the function we can call when the form is submitted**
  - **errors is a nested property in the formState object** which will contain the validation errors, if any

## How to Create a Basic Form with react-hook-form?

- We have **added a register function to each input field** by passing a unique name to each register function.
- you need to **pass a unique name to the register function** - so react-hook-form can track the changing data.
- When we submit the form,
  - **the handleSubmit function will handle the form submission.**
- It will send the user entered data to the onSubmit function where we're logging the user data to the console.
- This code is much simpler because we don't have **to add the value and onChange handler for each input field** and there is no need to manage the application state ourselves.

# How to Add Validations to the Form?

- let's add the **required field and minimum length validation** to the input fields.
- To add validation we can **pass an object to the register function** as a second parameter.
- For the email field, we're specifying required field validation.
- For the password field we're specifying the required field and minimum 6 character length validation.

```
<input  
    type="text"  
    name="email"  
    {...register("email", {  
        required: true  
    })}  
/>  
  
<input  
    type="password"  
    name="password"  
    {...register("password", {  
        required: true,  
        minLength: 6  
    })}  
/>>
```

# How to Add Validations to the Form?

- When the validation fails
  - the **errors object coming from the useForm hook will be populated with the fields for which the validation failed.**
- So we will use that **errors object to display custom error messages.**

```
{errors.email && errors.email.type === "required" && (  
  <p className="errorMsg">Email is required.</p>  
)}  
{errors.email && errors.email.type === "pattern" && (  
  <p className="errorMsg">Email is not valid.</p>
```

# How to Add Validations to the Form?

- We have added the **password field validation**
- Also, as you can see, each **input field is automatically focused** when we submit the form if there is any validation error for that input field.
- Also, the form is **not submitted as long as there is a validation error**.
- If you check the browser console, you will see that the **console.log statement is only printed if the form is valid** and there are no errors.
- So using **react-hook-form** reduced the amount of code that we had to write.
- The **validation is also responsive**, so once the field becomes valid, the error message goes away instantly.

# How to Add Validations to the Form?

- But as the number of validations for each field increases, the conditional checks and error message code will still increase.
- So we can further refactor the code to make it even simpler.

```
{...register("email", {  
    required: "Email is required.",  
    pattern: {  
        value: /^[^@ ]+@[^@ ]+\.[^@ .]{2,}$/,  
        message: "Email is not valid."  
    }  
})
```

- Here, we've directly provided the error message we want to display while adding the validation itself.
- We are displaying the error message **using the message property** available inside the errors object for each input field.

# How to Add a Multiple Validations?

- You can even provide multiple validations for the input field by adding a **validate object**.
- This is useful if you need to perform complex validations.

```
<input
  type="password"
  name="password"
  {...register("password", {
    required: true,
    validate: {
      checkLength: (value) => value.length >= 6,
      matchPattern: (value) =>
        /(?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?!.*\s)(?=.*[@#$*])/test(
          value
        )
    }
  ))}
/>
```

# How to Reset the Form Values?

- Sometimes, we need to reset/clear the data entered by the user after some action.
- For example,
  - once the form is submitted
  - we want to show the success message and
  - then clear the form data so the user should not re-submit the same data.
- In such a case,
  - we can call the **reset function** to clear the form data.

# How to Reset the Form Values?

- The reset function also accepts an optional object where you can pass the values you want the form data to reset.

```
reset({  
    username: "SASTRA",  
    email: "sastra@example.com",  
    password: "Test@123"  
});
```

- The key **username**, **email** or **password** should match with the name passed to the **register** function so the respective input field will be set to the passed value.

# References

- <https://www.educba.com/forms-in-react/>
- <https://www.freecodecamp.org/news/how-to-build-forms-in-react/>
- <https://www.telerik.com/blogs/how-to-programmatically-add-input-fields-react-forms>
- <https://www.robinwieruch.de/react-form/#form-library-react-hook-form>

# References

- <https://www.freecodecamp.org/news/how-to-create-forms-in-react-using-react-hook-form/>
- <https://blog.logrocket.com/react-hook-form-complete-guide/>
- <https://www.telerik.com/blogs/how-to-create-validate-react-form-hooks>
- <https://hygraph.com/blog/react-hook-form>

Thank  
you

# Lecture Notes: Modularization and Webpack

## Contents

<b>1 Introduction to Modularization</b>	<b>2</b>
<b>2 Introduction to Webpack</b>	<b>2</b>
<b>3 Installing Webpack</b>	<b>2</b>
<b>4 The Webpack Configuration File</b>	<b>2</b>
<b>5 Using Babel with Webpack</b>	<b>3</b>
<b>6 Transform and Bundle</b>	<b>3</b>
<b>7 Libraries Bundle</b>	<b>4</b>
<b>8 Hot Module Replacement (HMR)</b>	<b>4</b>
<b>9 Webpack-Dev-Server</b>	<b>4</b>
<b>10 Server-Side ES2015</b>	<b>5</b>
<b>11 ESLint Integration</b>	<b>5</b>
<b>12 Conclusion</b>	<b>5</b>

# 1 Introduction to Modularization

Modularization is the practice of dividing a codebase into small, reusable, and manageable components or modules. This enhances the maintainability and scalability of large applications. Each module handles a specific piece of functionality, making it easier to understand, test, and debug.

## 2 Introduction to Webpack

Webpack is a popular module bundler for JavaScript applications. It processes various assets such as JavaScript files, CSS, images, etc., and bundles them into one or more files that can be served to the browser efficiently. It supports:

- Static module bundling.
- Code splitting for lazy loading.
- Hot Module Replacement (HMR) for a better development experience.
- Advanced optimizations for production builds.

## 3 Installing Webpack

To install Webpack as a development dependency, use the following command:

```
npm install --save-dev webpack webpack-cli
```

Once installed, you can run webpack from the command line using:

```
npx webpack
```

## 4 The Webpack Configuration File

Webpack uses a configuration file (usually `webpack.config.js`) to specify entry points, output locations, and module rules. Below is an example configuration:

```
const path = require('path');

module.exports = {
  entry: './src/App.jsx',
```

```

output: {
  path: path.resolve(__dirname, 'dist'),
  filename: 'app.bundle.js'
},
module: {
  rules: [
    {
      test: /\.jsx$/,
      exclude: /node_modules/,
      use: {
        loader: 'babel-loader',
        options: {
          presets: ['@babel/preset-react', '@babel/preset-env']
        }
      }
    }
  ]
},
mode: 'development'
};

```

## 5 Using Babel with Webpack

Babel is a JavaScript transpiler that converts modern JavaScript (ES6+) and JSX code into older JavaScript versions compatible with most browsers. To use Babel with Webpack, install the necessary loaders:

```
npm install --save-dev babel-loader @babel/core @babel/preset-env @babel/preset-react
```

Then configure Webpack to use Babel by adding the loader in the `module.rules` section, as shown in the previous example.

## 6 Transform and Bundle

Webpack can transform JSX and ES2015 (ES6) into browser-compatible code, and bundle them into a single file that is served to the client. This is especially important in React applications that use JSX, which needs to be compiled before browsers can interpret it.

## 7 Libraries Bundle

Webpack allows bundling third-party libraries such as React. By separating the app code from the library code, it improves caching. For example:

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

Webpack bundles React and your application code into a single file (or separate files in case of code splitting).

## 8 Hot Module Replacement (HMR)

HMR is a feature of Webpack that allows modules to be updated in the browser without a full page reload. This significantly improves the development process by maintaining the state of the application while testing changes in real-time.

To enable HMR, update your Webpack configuration:

```
const webpack = require('webpack');

module.exports = {
  entry: ['./src/App.jsx', 'webpack-hot-middleware/client'],
  plugins: [
    new webpack.HotModuleReplacementPlugin()
  ],
  devServer: {
    hot: true,
    contentBase: path.resolve(__dirname, 'dist'),
    port: 8000
  }
};
```

## 9 Webpack-Dev-Server

Webpack-Dev-Server is a lightweight server that serves your application bundle in memory, making it faster for development purposes. It supports features like live reloading and HMR. To install:

```
npm install --save-dev webpack-dev-server
```

You can then start the server using:

```
npx webpack-dev-server --mode development
```

## 10 Server-Side ES2015

Using Babel with Webpack, you can also use ES2015 features in your server-side code. This allows unifying the development experience across both the client and server sides.

For example, to transform server-side code, use Babel with the following command:

```
npx babel src --out-dir dist --presets=@babel/preset-env
```

## 11 ESLint Integration

ESLint is a linting tool that helps identify problematic patterns or code that doesn't follow specific style guidelines. Integrating ESLint with Webpack ensures your code is consistent and free of errors. First, install ESLint and its dependencies:

```
npm install --save-dev eslint eslint-loader
```

Next, configure ESLint within Webpack:

```
module: {
  rules: [
    {
      test: /\.jsx?$/,
      exclude: /node_modules/,
      use: ['babel-loader', 'eslint-loader']
    }
  ]
}
```

## 12 Conclusion

Modularization and Webpack are powerful tools for managing modern JavaScript applications. By breaking down the code into manageable modules and leveraging Webpack for bundling and transformation, developers can ensure scalable, efficient applications. Webpack's features such as HMR, Babel integration, and ESLint help streamline the development process and enhance code quality.

## **React JS Implementing modularization with webpack**

Modularization in React.js with Webpack involves organizing your code into modules to enhance maintainability, scalability, and readability. Webpack is a popular module bundler that allows you to bundle JavaScript files and other assets (CSS, images, etc.) into a single or multiple optimized files.

### **Steps to Implement Modularization in React.js with Webpack**

- 1. Initialize a React Project**
- 2. Install Webpack and Necessary Dependencies**
- 3. Configure Webpack**
- 4. Set Up Babel for JSX and ES6**
- 5. Organize Your Code into Modules**

#### **1. Initialize a React Project**

First, create a basic directory structure and initialize a new Node.js project.

```
mkdir react-webpack-modular  
cd react-webpack-modular  
npm init -y
```

This will create a `package.json` file.

#### **2. Install Webpack and Necessary Dependencies**

Next, install Webpack, Babel, and other essential dependencies:

```
npm install webpack webpack-cli webpack-dev-server --save-dev  
npm install babel-loader @babel/core @babel/preset-env @babel/preset-react --save-dev  
npm install react react-dom
```

Explanation:

- `webpack`, `webpack-cli`: The Webpack bundler and its command-line interface.
- `webpack-dev-server`: Provides a development server with live reloading.
- `babel-loader`, `@babel/core`, `@babel/preset-env`, `@babel/preset-react`: Babel dependencies to transpile modern JavaScript and JSX.
- `react`, `react-dom`: React libraries.

#### **3. Configure Webpack**

Create a `webpack.config.js` file in the root directory:

Now, configure Webpack in `webpack.config.js`:

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js',
  },
  module: {
    rules: [
      {
        test: /\.js|jsx$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
        },
      },
    ],
  },
  resolve: {
    extensions: ['.js', '.jsx'],
  },
  devServer: {
    static: {
      directory: path.join(__dirname, 'dist'), // Replace contentBase with static.directory
    },
    compress: true,
    port: 9000,
  },
  mode: 'development',
};
```

## 4. Set Up Babel for JSX and ES6

Create a `.babelrc` file in the root directory to configure Babel:

In `.babelrc`, add the following configuration:

```
{
  "presets": [
    "@babel/preset-env",
    "@babel/preset-react"
  ]
}
```

- `@babel/preset-env`: Transpiles modern JavaScript (ES6+).
- `@babel/preset-react`: Transpiles JSX syntax used by React.

## 5. Organize Your Code into Modules

Organize your project structure to follow modular principles:

```
mkdir src
```

```
create src/index.js      src/App.js
```

Now, define your React components in modules. For example, `App.js` will hold the main React component:

**src/App.js:**

```
import React from 'react';
import Header from './components/Header'; // Import Header module
import Footer from './components/Footer'; // Import Footer module

const App = () => (
  <div>
    <Header />
    <h1>Welcome to Modularized React App</h1>
    <Footer />
  </div>
);

export default App;
```

**src/index.js:**

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App'; // Import the main App component

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

**Next, create a folder for components and define the `Header` and `Footer` components:**

```
mkdir src/components
```

```
src/components/Header.js
src/components/Footer.js
```

**src/components/Header.js:**

```
import React from 'react';

const Header = () => <header><h2>Header Component</h2></header>;

export default Header;
```

### src/components/Footer.js:

```
import React from 'react';

const Footer = () => <footer><h2>Footer Component</h2></footer>;

export default Footer;
```

## 6. Create an HTML File

Create an `index.html` file in the `dist` directory that will load the bundled JavaScript:

```
mkdir dist
```

### dist/index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>React App Modularization</title>
</head>
<body>
  <div id="root"></div>
  <script src="bundle.js"></script> <!-- Webpack will inject bundle.js here -->
</body>
</html>
```

## 7. Run the Development Server

To start the Webpack development server, add the following script to `package.json` under `"scripts"`:

```
"scripts": {
  "start": "webpack serve --open"
}
```

Now, run the development server:

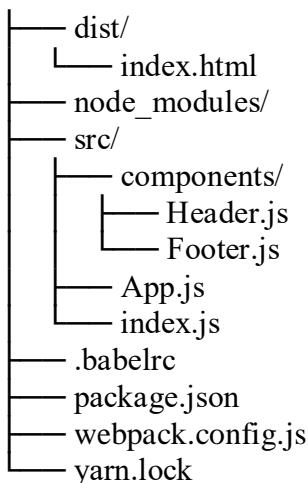
```
npm start
```

This will open the app in the browser at `http://localhost:9000`, and the development server will rebuild the app on every change.

## Modular Project Structure

After modularization, your project structure should look like this:

```
react-webpack-modular/
```



## Summary

- **Webpack** is used to bundle your React code and manage dependencies.
- **Babel** is configured to transpile modern JavaScript and JSX to older JavaScript versions.
- Code is modularized by splitting components into separate files and folders, improving maintainability and reusability.
- **Webpack Dev Server** provides a local server with live reloading.
- For production, Webpack optimizes the bundle with minification and other optimizations.

This setup gives you a flexible and efficient way to build and maintain modular React applications using Webpack. Would you like to dive deeper into optimizing the Webpack configuration for production or more advanced features?

---

## Error:

```
node_modules/open/index.js:6 import fs, { constants as fsConstants } from 'node:fs/promises';
^^^^^^^^^ SyntaxError: The requested module 'node:fs/promises' does not provide an export
named 'constants'
```

The error you're encountering happens because you're trying to import `constants` from the '`node:fs/promises`' module, which doesn't export `constants`. The correct module to import `constants` from is '`fs`', not '`fs/promises`'.

## How to Fix the Error

You should separate the imports and correctly import `fsConstants` from '`fs`' instead of '`fs/promises`'.

Here's how to fix it:

**nodemodules => open => index.js**

remove the following

```
import fs, {constants as fsConstants} from 'node:fs/promises';
```

Add the following import statement:

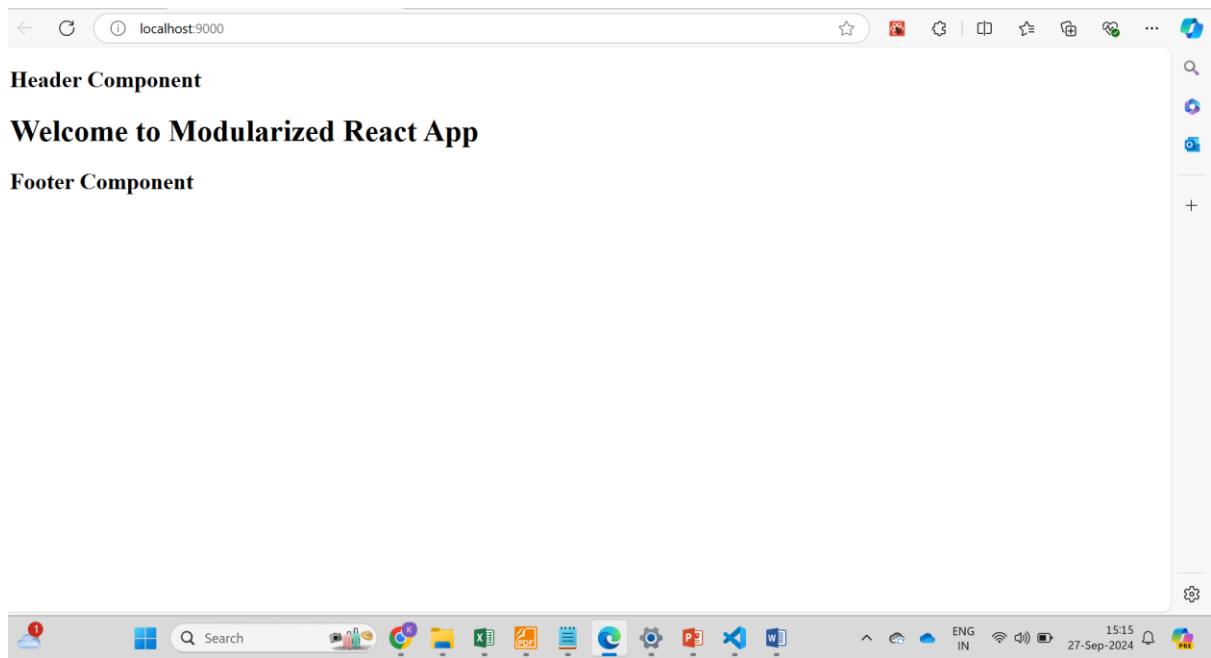
```
import fs from 'node:fs/promises';
import { constants as fsConstants } from 'fs';
```

## Explanation:

- `fs/promises`: This provides the promise-based API for file system operations (like `fs.readFile`).
- `fs.constants`: This is the correct location to import `constants` (such as `O_RDONLY`, `O_WRONLY`).

---

Check the output at <http://localhost:9000/> (in web browser)



# React.js with Babel and Webpack

Babel is responsible for converting the ES5 and ES6 code to browser understandable code, basically backward compatibility

Babel packages

1. **babel-core**: the main engine of babel plugin for its dependents to work.
2. **babel-preset-env**: This is the ES5, ES6 supporting part
3. **babel-preset-react**: Babel can be used in any framework that needs the latest JS syntax support, in our case, it's "React", hence this preset.
4. **babel-loader**: This as a bridge of communication between Webpack and Babel  
Webpack is the most widely used and an accepted module bundler and task runner throughout React.js community.

packages :

1. **webpack**: The main webpack plugin as an engine for its dependents.
2. **webpack-cli**: To access some webpack commands through CLI like starting dev server, creating production build, etc.
3. **webpack-dev-server**: A minimal server for client-side development purpose only.
4. **html-webpack-plugin**: Will help in creating HTML templates for our application.

WEBPACK program to greet user with header and footer

Create a folder

```
mkdir wpack
```

```
cd wpack
```

## **Step 1: \*\*Initialize npm\*\***

Initialize a new npm project:

### **Initialize package.json**

```
npm init -y
```

## **Step 2: \*\*Install Dependencies\*\***

Install React, ReactDOM, and Webpack with the necessary loaders and plugins:

```
npm install react react-dom
```

```
npm install webpack webpack-cli webpack-dev-server babel-loader @babel/core  
@babel/preset-env @babel/preset-react html-webpack-plugin --save-dev
```

## **Step 3: Create Project Structure**

Here's a modularized structure:

```
...  
wpack/  
    └── src/  
        ├── components/  
        │   ├── Header.js  
        │   ├── Footer.js  
        │   └── MainContent.js  
        ├── utils/  
        │   └── greetings.js  
        ├── index.js  
        └── App.js  
    └── public/  
        └── index.html  
    └── webpack.config.js  
└── package.json
```

## **Step 4: Create React Components**

### **1. \*\*src/components/Header.js\*\***

```
import React from 'react';  
  
const Header = () => {  
  return (  
    <header>
```

```
<h1>Welcome to My React App</h1>
</header>
);
};
export default Header
```

## 2. \*\*src/components/Footer.js\*\*

```
import React from 'react';

const Footer = () => {
return (
<footer>
<p>© 2024 My React App</p>
</footer>
);
};
export default Footer
```

## 3. \*\*src/components/MainContent.js\*\*

```
import React from 'react';

import { greetUser } from '../utils/greetings';
const MainContent = () => {
return (
<div>
<h2>{greetUser("User")}</h2>
<p>This is the main content of the app.</p>
</div>
);
};

export default MainContent;
```

## 4. \*\*src/utils/greetings.js\*\*

```
export const greetUser = (name) => {

  return `Hello, ${name}!`;
};
```

## 5. \*\*src/App.js\*\*

```
import React from 'react';

import Header from './components/Header';
import Footer from './components/Footer';
import MainContent from './components/MainContent';
```

```

const App = () => {
  return (
    <div>
      <Header />
      <MainContent />
      <Footer />
    </div>
  );
};

export default App;

```

## 6. \*\*src/index.js\*\*

```

import React from 'react';

import { createRoot } from 'react-dom/client';
import App from './App';

const domNode = document.getElementById('root');
const root = createRoot(domNode);

root.render(<App />);

```

## Step 5: Create HTML Template

### 1. \*\*public/index.html\*\*

```

<!-- public/index.html -->

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>React with Webpack</title>
</head>
<body>
  <div id="root"></div>
</body>
</html>

```

## Step 6: Webpack Configuration

### 1. \*\*webpack.config.js\*\*

```

const path = require('path');
const HWP = require('html-webpack-plugin'); // importing the html plugin
module.exports = {
  entry: path.join(__dirname, '/src/index.js'), // indicates the starting point
  of the application.

```

```

output: { // in production what should be the name of the main file
  (build.js) and where will it be created (inside c:\app\dist) automatically.
  filename: 'build.js',
  path: path.join(__dirname, '/dist')},
  module:{ 
    rules:[{// the rules webpack should be bind to
      test: /\.js$/,
      exclude: /node_modules/,
      loader: 'babel-loader'
    }]
  },
  plugins:[// other plugins that webpack needs to know for the purpose of
  bundling or transpiling
    new HtmlWebpackPlugin({template: path.join(__dirname,'/public/index.html')})
  ]
}

```

## Step 7: Update package.json

Ensure your `package.json` has the scripts defined:

```

"scripts": {
  "start": "webpack-dev-server --mode development --open --hot",
  "build": "webpack --mode production"
},

```

## Step 8: create ./babelrc

```

{
  "presets": ["@babel/preset-env", "@babel/preset-react"]
}

```

## Step 9: Run Your Application

### 1. \*\*Start the Development Server\*\*

**npm start**

webpack-dev-server is going to fire up the application in **mode=development**, then display ( — **open**) it in your default browser automatically and keep watching for any changes made to the application ( — **hot**).

### Step 10: Build the Application for Production\*\*

**npm run build**

## Example 2 : webpack with babel and dynamic router

### Project structure

```
wpack/
└── src/
    ├── components/
    │   ├── Header.js
    │   ├── Home.js
    │   └── StudentDetail.js
    ├── Footer.js
    └── MainContent.js
    └── utils/
        └── greetings.js
    └── index.js
    └── App.js
    └── public/
        └── index.html
    └── webpack.config.js
    └── package.json
```

**Repeat the same steps , additionally import router dom**

```
npm install react-router-dom
```

```
// src/components/Home.js
import React from 'react';
import { Link } from 'react-router-dom';

const students = [
{ id: 1, name: 'Alice', age: 20 },
{ id: 2, name: 'Bob', age: 22 },
{ id: 3, name: 'Charlie', age: 21 },
];

const Home = () => {
return (
<div>
<h2>Student List</h2>
<ul>
{students.map((student) => (
<li key={student.id}>
<Link to={`/student/${student.id}`}>{student.name}</Link>
</li>
))}
</ul>
```

```

</div>
);
};

export default Home;

// src/components/StudentDetail.js
import React from 'react';
import { useParams } from 'react-router-dom';

const studentData = [
{ id: 1, name: 'Alice', age: 20, major: 'Computer Science' },
{ id: 2, name: 'Bob', age: 22, major: 'Mathematics' },
{ id: 3, name: 'Charlie', age: 21, major: 'Physics' },
];

const StudentDetail = () => {
const { id } = useParams();
const student = studentData.find((s) => s.id === parseInt(id));

if (!student) {
return <h2>404 - student not found!</h2>;
}

return (
<div>
<h2>Student Details</h2>
<p><strong>Name:</strong> {student.name}</p>
<p><strong>Age:</strong> {student.age}</p>
<p><strong>Major:</strong> {student.major}</p>
</div>
);
};

export default StudentDetail;

```

## src/App.js

```

import React from 'react';
import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';
import Header from './components/Header';
import Footer from './components/Footer';
import MainContent from './components/MainContent';
import Home from './components/Home';
import StudentDetail from './components/StudentDetail';

function App() {
return (

```

```
<div>
  <Header />
  <MainContent />
  <h1>Student Details App</h1>
  <Router>
    <Routes>
      <Route path="/" element={<Home/>} />
      <Route path="/student/:id" element={<StudentDetail/>} />
    </Routes>
  </Router>
  <Footer />
</div>
);
};

export default App;
```

npm start

# Welcome to My React App

## Hello, User!

This is the main content of the app.

# Student Details App

## Student List

- [Alice](#)
- [Bob](#)
- [Charlie](#)

© 2024 My React App



# SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

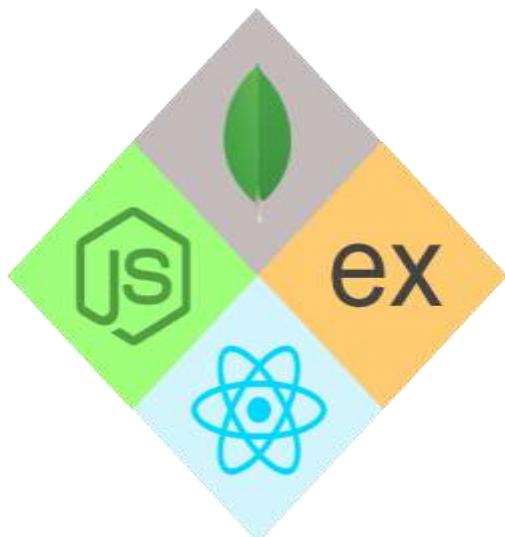
(U/S 3 of the UGC Act, 1956)

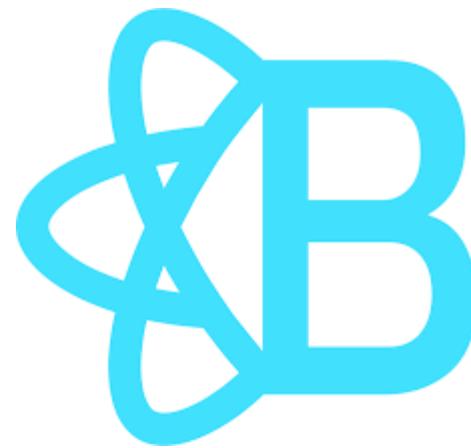
THINK MERIT | THINK TRANSPARENCY | THINK SASTRA



## INT436

# FULL STACK WEB APPLICATION DEVELOPMENT





# Introduction

- Bootstrap is a **free, open source front-end development framework** for the creation of websites and web apps.
- Bootstrap provides a **responsive grid system** that automatically **adjusts the layout** based on the screen size, ensuring your website looks great on various devices.
- Bootstrap offers a **wide range of pre-built components**, such as navigation bars, forms, buttons, and more, saving you time and effort in coding from scratch.

# Why React-Bootstrap?

- React-Bootstrap is a **complete re-implementation** of the Bootstrap components using React.
- It has no dependency on either **bootstrap.js** or **jQuery**.
- If you have **React** setup and **React-Bootstrap** installed, you have everything you need.
- Methods and events using **jQuery** is done **imperatively** by directly manipulating the **DOM**.
- In contrast, React uses updates to the state to update the **virtual DOM**.

# How to add Bootstrap to React?

- The **three most common ways** to add Bootstrap to your React app are:
  - Using the Bootstrap CDN
  - Importing Bootstrap in React as a dependency
  - Installing a React Bootstrap package such as React-Bootstrap or Reactstrap

# How to add Bootstrap to React?

## 1. Adding Bootstrap using the Bootstrap CDN

- The Bootstrap CDN is the **easiest way** to add Bootstrap to your React app.
- You just have to **include a link to the CDN** in the head section of your application entry file;
  - no extra installation or downloads are required.
- React application created with Create React App - **index.html file**.
- ```
<link rel="stylesheet"
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.1/dist/css/bootstrap.min.css" integrity="sha256-2TnSHycBDAm2wpZmgdi0z81kykGPJAkiUY+Wf97RbvY="
      crossorigin="anonymous">
```

# How to add Bootstrap to React?

- If your project also requires using the **JavaScript components** that ship with Bootstrap, such as toggling a modal, dropdown, or navbar, we'll need to link the **bootstrap.bundle.min.js file**, which comes precompiled with Popper.js.
- We can do this by placing the following `<script>` tag
  - before the closing `</body>` tag.
- ```
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.1/dist/js/bootstrap.mi
n.js" integrity="sha256-
gOQJla9+K/XdfAuBkg2ONAdw5EnQbokw/s2b8BqsRFg="
crossorigin="anonymous">
</script>
```

# How to add Bootstrap to React?

## 2. Importing Bootstrap as a dependency

- If you are using a **build tool** or a **module bundler** such as **webpack** or **Vite**, this will be the preferred option for adding Bootstrap to your React application.
- You can easily start the installation by running the following command.

**npm install bootstrap**

# OR **npm install react-bootstrap bootstrap**

**yarn add bootstrap**

- This command will install the most recent version of Bootstrap.
- Once the installation is complete, we can include it in our app's entry file:

```
// Bootstrap CSS
import "bootstrap/dist/css/bootstrap.min.css";
// Bootstrap Bundle JS
import "bootstrap/dist/js/bootstrap.bundle.min";
```

# How to add Bootstrap to React?

### 3. Installing a React Bootstrap package such as React-Bootstrap or Reactstrap

- The third way to add Bootstrap to a React app is to **use a package that has prebuilt Bootstrap components designed to work as React components.**
- The benefit of this method is that **practically all Bootstrap components are bundled as a React component in these libraries.**
- For example, a full Bootstrap modal component can now be easily imported as **<Modal />** in our React application.
- While there are various packages you can use to install Bootstrap in React, the two most popular are **React-Bootstrap** and **Reactstrap**.
- These packages share very similar characteristics.

# How to use React Bootstrap Buttons?

# Button

Username

Email

Password

Register

Primary Button

Secondary Button

Success Button

Warning Button

Danger Button

Info Button

Light Button

Dark Button

Link Button

# Buttons

- Use Bootstrap's custom **button styles** for actions in forms, dialogs, and more with support for **multiple sizes, states**, and more
- Use any of the **available button style types** to quickly create a styled button.
- Just modify the **variant prop**.

```
<Button variant="primary">Primary</Button>{' '}  
<Button variant="secondary">Secondary</Button>{' '}  
<Button variant="success">Success</Button>{' '}  
<Button variant="warning">Warning</Button>{' '}  
<Button variant="danger">Danger</Button>{' '}  
<Button variant="info">Info</Button>{' '}  
<Button variant="light">Light</Button>{' '}  
<Button variant="dark">Dark</Button>  
<Button variant="link">Link</Button>
```

# Buttons

Primary Button

Secondary Button

Success Button

Warning Button

Danger Button

Info Button

Light Button

Dark Button

Link Button

```
<Button variant="outline-primary">Primary</Button>{' '}  
<Button variant="outline-secondary">Secondary</Button>{' '}  
<Button variant="outline-success">Success</Button>{' '}  
<Button variant="outline-warning">Warning</Button>{' '}  
<Button variant="outline-danger">Danger</Button>{' '}  
<Button variant="outline-info">Info</Button>{' '}  
<Button variant="outline-light">Light</Button>{' '}  
<Button variant="outline-dark">Dark</Button>
```

Bootstrap also supports lighter stylish outlined buttons without having background color. The outlined button's variant name starts with outlined-\* ; for example, if we want to use the success button as outline, then its variant name will be outlined-success.

Link

Button

Input

- Generally the `<Button>` component renders the normal HTML `<button>` tag,
  - but, when it is used **with as or href prop** it renders the respective element.

```
<Button href="#">Link</Button> <Button type="submit">Button</Button>{}  
<Button as="input" type="button" value="Input" />{}  
<Button as="input" type="submit" value="Submit" />{}
```

it will render the `<a>` tag instead of the `<button>` tag. However, it will keep the style of the button.



- Bootstrap also supports large and small sizes buttons.
- The **size prop** is used to define the size of a button.
- The large buttons have a larger font size and padding value.

```
<Button variant="primary" size="lg">  
  Large button  
</Button>{ ' '}
```

```
<Button variant="primary" size="sm">  
  Small button  
</Button>{ ' '}
```

Disabled Button

Disabled Button

Disabled Link

- Make buttons look inactive by adding the **disabled prop.**

```
<Button variant="primary" size="lg" disabled>  
  Disabled Button  
</Button>{''}
```

```
<Button variant="secondary" size="lg" disabled>  
  Disabled Button  
</Button>{''}
```

# Modal Component

- Modal Component provides a way **to add dialogs to our website** for user notifications or to display information to the user.
- import the modal component and the styling into your module.
- **import Modal from "react-bootstrap/Modal";**
- **import "bootstrap/dist/css/bootstrap.min.css";**
- A modal has a few basic sections:
  - **The Header, the Title, the Body, and the Footer.**
  - These sections will hold the content that we need to display.
- Here's an example displaying a basic modal using these components.

# Modal Component

```
import Modal from "react-bootstrap/Modal";
import "bootstrap/dist/css/bootstrap.min.css";

const App = () => {
  return (
    <Modal show={true}>
      <Modal.Header>SASTRA</Modal.Header>
      <Modal.Body>Computing</Modal.Body>
      <Modal.Footer>SASTRA</Modal.Footer>
    </Modal>
  );
};

};
```

- The **size property on the <Modal> component** can be used to set the width of the modal to the defaults defined in the bootstrap CSS.
- There are three options: **sm, lg, xl**.
- If you want to define your own custom class to set the width, you can do that as well.

```
const App = () => {
  return (
    <Modal show={true} size="lg">
      <ModalHeader>
        <ModalTitle>SASTRA</ModalTitle>
      </ModalHeader>
      <ModalBody>Computing</ModalBody>
      <ModalFooter>Thanjavur </ModalFooter>
    </Modal>
  );
};
```

# Hiding/Showing a Modal

- Our previous example shows how to display a modal using **show property** on the modal component.
- Hard-coded the value to **true**, but it is not very flexible.
- To improve this,
  - let's set the value to a variable or a toggle-able expression.
- We will also create a **button to toggle the visibility**.

# Hiding/Showing a Modal

```
const App = () => {
  const [isOpen, setIsOpen] = React.useState(false);

  const showModal = () => {
    setIsOpen(true);
  };

  const hideModal = () => {
    setIsOpen(false);
  };
}
```

# Hiding/Showing a Modal

```
return (
  <>
  <button onClick={showModal}>Display Modal</button>
  <Modal show={isOpen} onHide={hideModal}>
    <Modal.Header>
      <Modal.Title>SASTRA</Modal.Title>
    </Modal.Header>
    <Modal.Body>COMPUTING</Modal.Body>
    <Modal.Footer>
      <button onClick={hideModal}>Cancel</button>
      <button>Save</button>
    </Modal.Footer>
  </Modal>
</>
);
};
```

# Hiding/Showing a Modal

- Added a `showModal` and `hideModal` method to update a state property called `isOpen`.
- Assigning the `isOpen` variable as the value to the `show` property means we now have control over whether the modal is showing or not.
- The new `onHide property` is necessary if we want to hide the modal when clicking on the non-static backdrop or hitting the esc key.
- Also added a couple of buttons to the footer to make this modal more realistic.
- In addition to hiding the Modal by hitting the "Esc" key or clicking the backdrop,
  - put the `hideModal` method on the Cancel button to show a different way of hiding the modal.

# References

<https://react-bootstrap.netlify.app/docs/getting-started/why-react-bootstrap>

<https://www.positronx.io/how-to-integrate-tabs-in-react-js-with-react-bootstrap/>

<https://blog.logrocket.com/using-bootstrap-react-tutorial-examples/>

<https://www.pluralsight.com/guides/working-with-bootstraps-modals-react>

