Agenda

- Web Architecture
- Introduction to HTML
- Tags

Application

- It is a program that contains set of instructions for the CPU
- It can be developed in any type of langugage
- Their are generally 2 types of application

1. Native Application
    - Developed using C and CPP
    - It is bit faster
    - It is OS Dependent
2. Web Application
    - Developed using HTML,CSS and JavaScript
    - slower than the native application
    - It is OS Independent

Server

- It is a physical device with highest configuration that is used to process the data on large scale and give out the response
- It generally consists of a-

1. Web server
2. DB Server
3. Languages
4. Operating System (platform)

- Server can have a Software Stack which is a Collection of softwares
- examples

1. WISA
    - Windows
    - IIS
    - SQL Server
    - ASP.net
2. MEAN
    - MongoDB
    - Express
    - Angular
    - Node

Browser Architecture

1. User Interface

- This is the user interface for the browser.
- It includes the Address Bar, back button, Bookmarking options, Refresh button, etc.
- The browser's user interface is not specified in any formal specification, but comes from practices shaped over decades of experience (and browsers copying each other).
- As a result, all browsers have UIs that are extremely similar to each other.

2. The Browser Engine

- The browser engine marshals actions between the browser's user interface and the browser's rendering engine.
- When you type in a new website and press the enter key, the browser UI will tell the browser engine, which will then communicate with the rendering engine.

3. The Rendering Engine

- The rendering engine is responsible for displaying the requested content.
- The rendering engine will start by getting the contents of the requested document from the networking layer.
- It takes in the HTML code and parses it to create the DOM (Document Object Model) tree.
- Examples of rendering engine include
    1. Safari - WebKit Rendering Engine
    2. Chrome - Blink Rendering Engine (Blink is a fork of WebKit)
    3. FireFox - Gecko Rendering Engine

4. Networking Layer

- The Networking Layer is responsible for making network calls to fetch resources.
- It imposes the right connection limits, formats requests, deals with proxies, caching, and much more.

5. JavaScript Engine

- The JavaScript Engine is used to parse and execute JavaScript code on the DOM.
- The JavaScript code is provided by the web server, or it can be provided by the web browser
- Early browsers used JavaScript interpreters, but modern JavaScript engines use Just-In-Time compilation for improved performance.
- Examples of JavaScript Engine include
    1. Safari - JavaScriptCore
    2. Chrome - V8 JavaScript Engine
    3. FireFox - SpiderMonkey Engine

6. UI Backend

- This layer is responsible for drawing the basic widgets like select or input boxes and windows. Underneath it uses operating system UI methods.
- The rendering engine uses the UI backend layer during the layout and painting stages to display the web page on the browser.

7. Data Storage

- The browser needs to save data locally (cookies, cache, etc.) so the Data Storage component handles this part.

Web Architecture

1. Web Server

- A web server is a system (hardware and/or software) that delivers web content, such as websites, web pages, and other resources, to users over the internet or an intranet.
- It handles requests from client devices, typically browsers, and serves the requested resources using standard web protocols like HTTP (Hypertext Transfer Protocol) or HTTPS (HTTP Secure).
- There are many web server software options, each with unique features and use cases. Popular ones include:
    1. Apache HTTP Server: Open-source and widely used, known for flexibility and extensive module support.
    2. Nginx: High-performance and lightweight, often used for handling large amounts of traffic.
    3. Microsoft IIS (Internet Information Services): A web server for Windows environments, integrated with Microsoft technologies.

2. HTTP Request

- An HTTP request is sent by a client to a server to ask for a resource or perform an action.
- It consists mainly of two parts - Header and Body
- Headers Provide additional information about the request.
- Examples:
    - Host: Specifies the target host (e.g., Host: example.com).
    - User-Agent: Identifies the client making the request (e.g., browser or app).
    - Content-Type: Indicates the format of the request body (e.g., application/json).
- Body (Optional): Contains data sent to the server (used in methods like POST or PUT).

3. HTTP Response

- An HTTP response is sent by the server to the client in reply to a request, containing the requested resource, a status code, and other metadata.
- It consists mainly of two parts - Header and Body
- Headers Provide metadata about the response.
- Examples:
    - Content-Type: The format of the response data (e.g., text/html or application/json).
    - Content-Length: The size of the response body in bytes.
    - Set-Cookie: Used to send cookies to the client.
- Body (Optional): Contains the requested resource or additional information.
- Example:
    - HTML for a webpage.
    - JSON data for an API.

HTTP Request Methods

1. GET: Requests a resource without modifying it.
2. POST: Submits data to the server for processing.

3. PUT: Updates or replaces an existing resource.
4. DELETE: Deletes a resource.

HTTP Response Status Code

1. 1xx (Informational): Request received, continuing process.
    - 100 Continue: Initial part of a request received.
2. 2xx (Success): Request was successful.
    - 200 OK: Request succeeded.
    - 201 Created: Resource was successfully created.
3. 3xx (Redirection): Client needs to take further action.
    - 301 Moved Permanently: Resource has a new URL.
    - 302 Found: Resource temporarily moved.
4. 4xx (Client Errors): Issues with the client's request.
    - 400 Bad Request: The request is malformed.
    - 401 Unauthorized: Authentication required.
    - 404 Not Found: Resource not found.
5. 5xx (Server Errors): Issues on the server side.
    - 500 Internal Server Error: Generic server error.
    - 503 Service Unavailable: Server is overloaded or down.

HTML

- HTML (HyperText Markup Language) is the standard language used to create and structure content on the web.
- It provides the foundation for web pages by defining the structure and elements of the content, such as headings, paragraphs, links, images, tables, and more.
- The latest standard is HTML5
- To add html 5 code, start the document with

```
<!DOCTYPE html>
```

- DOCTYPE: document type (tag used to start the html document)
- It is Case in-sensitive
- To add comment

```
<!-- comment -->
```

Key Features of HTML

1. Markup Language: HTML uses "tags" to mark up different parts of content, indicating their roles (e.g., heading, paragraph, list).
2. HyperText: Enables linking to other web pages or resources using hyperlinks (`<a>` tag).
3. Platform-Independent: HTML can be rendered on any device with a web browser.

4. Extensible: Can be combined with other technologies like CSS (for styling) and JavaScript (for interactivity).

HTML History

1. HTML 1.0 (1993)

- Tim Berners-Lee, the inventor of the World Wide Web.
- Purpose was to share research documents and enable hyperlinking between them.
- Key Features:
    - Basic text formatting (headings, paragraphs, lists).
    - Hyperlinks (`<a>` tag).

2. HTML 2.0 (1995)

- Standardize HTML for broader use.
- Key Features:
    - Support for tables and forms.
    - Basic image embedding (tag).
    - Standardized syntax rules.

3. HTML 3.2 (1997)

- Managed by the World Wide Web Consortium (W3C).
- Introduced features for richer content.
- Key Features:
    - Support for scripting languages like JavaScript.
    - Improved table support for better layouts.
    - Introduction of new elements like for styling.

4. HTML 4.01 (1999)

- W3C continued to refine HTML.
- Enhanced web functionality and accessibility.
- Key Features:
    - Separation of content and style via CSS.
    - Introduction of attributes like id and class for better styling and scripting.
    - Improved support for forms and multimedia.

5. XHTML 1.0 (2000)

- W3C attempted to merge HTML with XML.
- Introduced stricter syntax and better compatibility.
- Key Features:
    - All tags must be properly closed (e.g.,
).
    - Case sensitivity for tags and attributes.
- Challenges:
    - Too strict for practical use; adoption was limited.
    - Developers found it difficult to migrate from HTML 4.

6. HTML5 (2014, Official Recommendation)

- Modernize HTML for web applications and multimedia.
- Key Features:
  - Semantic Elements: `<header>, <footer>, <article>, <nav>` for better content structure.
  - Multimedia Support: Native audio (`<audio>`) and video (`<video>`) elements.
  - Canvas and SVG: For drawing graphics and animations directly in the browser.
  - APIs: Built-in APIs for features like geolocation, drag-and-drop, and offline storage.
  - Backward Compatibility: Maintains support for older HTML versions.
- It provided Simplified coding practices.

Tags

- Word enclosed by < and > signs
- It is also called as an element
- All tags in HTML are pre-defined by W3C
- E.g. `<h1>, <p>, <table>`

Types of tags

1. Opening
   - Used to open a data/information
   - E.g. `<h1>, <p>, <html>`
2. Closing
   - Used to close the data/information
   - E.g. `</h1>, </p>, </html>`
3. Empty
   - Tag having no data/content
   - Two ways of representing it
     - `<tag></tag>`
     - `<tag />`
     - E.g. `<br/>, <hr/>`
4. Root
   - Tag which starts and ends the document
   - Is also called as Document Type or Document Tag or Document Element
   - E.g. `<html>` is root tag for html document

Agenda

- Textual tags
- Formatting tags
- List tag
- Table tag
- Resource tags
- Anchor tag
- Form tag
- ~~GET vs POST method~~
- ~~Header,Footer,article,aside,nav~~

Attribute

- Extra information about the tag
- Attribute always present in name=value format
- E.g. `<meta charset=”utf-8”>`
  - meta: tag
  - charset: attribute name
  - utf-8: attribute value
- A tag may have one or more attributes
- Every tag has following attributes
- name: used to create query string
- id: used to identify an element uniquely
- style: used to write inline css
- class: used for css

HTML Structure

- It consists of 2 parts

1. Head

- Contains extra information about the page
- tags that can be used inside the head are
  1. title: used to set the title for the tab
  2. script: used to add JS code in the page
  3. style: used to add CSS code
  4. meta: used to add more information about the page
  5. link: used to link external documents (files)
  6. base: used to set the base url used in the page

2. Body

- Contains actual design
- tags that can be used inside the body are
  1. Textual
  2. Resources

3. List
4. Table
5. Linking(Anchor)
6. Form

- The tags can be inline or block tags
- the inline tags keep the data on the same line however the block tags add the data on the next line

Textual tags

1. Header: used to add header in page
    - There 6 levels
    - Tags: h1 to h6
    - H1 is the biggest while h6 is the smallest
2. Paragraph (

   ): used to add a para

3. Division(
   ): used to create groups of Tags and Textual contents

- All the above are block tags

Formatting Tags

- These all are inline tags

1. span - used during CSS
2. bold - `<b></b> or <strong></strong>`
3. itallic - `<i></i>`
4. underline - `<u></u>`
5. strike - `<strike></strike>`
6. monospace - `<tt></tt>`
7. superscript - `<sup></sup>`
8. subscript - `<sub></sub>`
9. font - `<font></font>`
10. formatted - `<pre></pre>`

List tags

1. Unordered list

- Does not render the order

```
<ul>
    <li>list item1</li>
    <li>list item2</li>
</ul>
```

2. Ordered list

- Renders the list item order

```
<ol>
    <li>list item1</li>
    <li>list item2</li>
</ol>
```

3. Definition list

- Used to create list of definitions

```
<dl>
    <dt>term 1</dt>
    <dd>definition 1</dd>

    <dt>term 2</dt>
    <dd>definition 2</dd>
</dl>
```

Table tag

- Used to create tabular representation
- Divided into 3 sections

1. Header
    - Use `<thead>` (table header)
2. Body
    - Use `<tbody>` (table body)
3. Footer
    - Use `<tfoot>` (table footer)

- To create row Use `<tr>`
- To create column In header Use `<th>` (table header column)
- In Body and Footer Use `<td>` (table data)
- Example

```
<table>
    <thead>
        <tr>
            <th></th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td></td>
```

```
        </tr>
    </tbody>
    <tfoot>
        <tr>
            <td></td>
        </tr>
    </tfoot>
</table>
```

- Attributes:

1. border: used to create border
2. colspan: used to merge multiple columns horizontally
3. rowspan: used to merge multiple columns vertically

Resource tag

1. Images

- img: used to add image
    - src: specify the source
    - alt: used to alternative text (rendered only in case of missing source)

2. Audio

- audio: used to play audio
    - src: to set the file to play
    - controls: to render controls
    - autoplay: to play the file automatically

3. Video

- video: used to play video
    - src: to set the file to play
    - controls: to render controls
    - autoplay: to play the file automatically

Linking Tag (Anchor)

- Used to link multiple pages
    - `<a href=”page.html”>text</a>`
- multiple sections within same page
    - `<a href=”#top”>text</a>`
- multiple pages with multiple sections
    - `<a href=”page#section”>text</a>`

Form tag

- Used to get inputs from user
- Inside the form tag we can use below tags

1. input

- used to get input from the user
- type attribute is used to specify the input type.
- value to type attribute can be
    - text: textual value
    - number: number value
    - email: email value
    - tel: telephone
    - date: date input
    - time: time input
    - radio: create radio button
    - checkbox: multiple selection
    - password: masked characters
    - submit: to submit values
    - reset: to clear the form
    - file: used to upload file

2. textarea

- used to get multi-line input

3. select

- used to render drop-down

```
<select>
    <option>op1</option>
    <option>op2</option>
</select>
```

Agenda

- Form tag
- GET vs POST method
- Header,Footer,section,article,aside,nav Tags
- CSS
- Selector
- CSS Box Model
- CSS Display

GET Method

- Values will be sent using Request Head
- Values will be appended on url (visible)
- Restriction on maximum size of data to be passed
- Only ascii values can be sent using GET
- URL can be bookmarked with the values
- Files can NOT sent using GET

POST Method

- Values will be sent using Request Body
- Values will be invisible
- There is no restriction on size of data
- Any value of any type (binary) can be sent by using POST
- URL can be bookmarked without the values
- Files can be using POST

article

- The
  HTML element represents a self-contained composition in a document, page, application, or site, which is intended to be independently distributable or reusable.
- Examples include a forum post, a magazine or newspaper article, or a blog entry, a product card, a user-submitted comment, an interactive widget or gadget, or any other independent item of content.

aside

- The
  HTML element represents a portion of a document whose content is only indirectly related to the document's main content.
- Asides are frequently presented as sidebars or call-out boxes.

header

- The
  element can define a global site header, described as a banner in the accessibility tree.

- It usually includes a logo, company name, search feature, and possibly the global navigation or a slogan.
- It is generally located at the top of the page.

footer

- The
  HTML element represents a footer for its nearest ancestor sectioning content or sectioning root element.
- It typically contains information about the author of the section, copyright data or links to related documents.

nav

- The
  HTML element represents a section of a page whose purpose is to provide navigation links, either within the current document or to other documents.
- Common examples of navigation sections are menus, tables of contents, and indexes.
- It's not necessary for all links to be contained in a
  element.
- is intended only for a major block of navigation links

section

- The
  HTML element represents a generic standalone section of a document, which doesn't have a more specific semantic element to represent it.

CSS

- CSS stands for Cascading Style Sheets.
- It makes an HTML website presentable.
- It adds style to various HTML elements.
- It helps you to define how the elements should look, where they should be placed and whether they should be displayed or not.
- Types of CSS

1. Inline
2. Internal
3. External

Inline CSS

- Use style attribute of an element to decorate it
- Simplest way to add decoration
- It is very difficult to manage because it can target only one element at a time
- It is discouraged to use inline CSS
- E.g. `<p style=”color:red;”>test</p>`

Internal CSS

- Use style tag in header section
- It can target multiple elements at a time in the given page
- It can target only one page at a time
- E.g.

```
p {
color: red;
}
```

External CSS

- Use external CSS file to hold all the rules
- Link the external CSS with link tag in header section
- E.g.

```
<link rel=”stylesheet” href=”mystyles.css”>
```

Terminology

- Rule or Ruleset:
  - Pair of CSS selector and declaration block
- Declaration block:
  - Collection of declarations
- Declaration:
  - Pair of CSS property and its value separated by colon(😃 and terminated by semi-colon(😉
- Selector:
  - Used to select one or more elements from the page

Units

- px: pixel (Picture Element)
- deg: degree
- s: Seconds
- %: with respect to its parent

CSS Selector

- used to select one or more elements from given page
- Types of selector

1. Element selector
   - Also called as type selector
   - Targets only similar type of element

- E.g. : only paragraph will have red color

```
p { color: red; }
```

2. Multiple element selector (,)
   - Multiple type selector
   - Select multiple type of elements
   - E.g. : paragraph, h2 and h3 will have color green

```
p, h2, h3 { color: green; }
```

3. ID selector (#)
   - Select only element matching the given Id
   - Id can appear in a page only once
   - E.g. :

```
/* select only paragraph having id para1  */
p#para1 { color:red; }

/* select any element having id para1 */
#para1 { color: green; }
```

4. Class selector (.)
   - Select only element matching the given class
   - E.g.

```
/* select only paragraph having class para1 */
p.para1 { color:red; }

/* select any element having class para1 */
.para1 { color: green; }
```

5. Descendant selector (white-space)
   - Used to select child elements at any level
   - E.g.

```
/* paragraph at any level inside div will have color red */
div p { color: red; }
```

6. Child selector (>)
   - Used to select child elements which are at first level

- Used to select only direct child elements
- E.g.

```
/* paragraph at first level (direct) inside div will have color red */
div > p { color: red; }
```

7. Attribute selector ([])
   - Used to select an element based on the given attribute value
   - E.g.

```
/* input of type submit will have color red */
input[type=”submit”] { color: red; }
```

8. Universal selector (*)
   - Used to select all elements
   - E.g.

```
/* all elements will have font family as arial */
* { font-family: arial; }
```

9. Pseudo selector (😃)
   - Used to apply CSS rules in specific condition
   - The conditions are also known as pseudo classes
   - E.g. hover, nth-child, active, focus, visited
   - E.g.

```
/* when mouse gets over on div, the color will change to red */
div:hover { color: red; }
```

CSS Box Model

- Every element in html is rendered as a box (rectangle)
- Their are 3 Properties
  1. Border
  2. Padding
  3. Margin

CSS Position

- Used to control the position
- Values are

1. static:
    - by default static is used
    - ignores top, bottom, left and right
2. relative:
    - element is aligned with respective with top,bottom,left and right
3. absolute:
    - It is absolute with the current displayed window and moves up as window scrolls
4. fixed:
    - It is fixed at the position. Even if window scrolls the element will not move from the position

CSS Display

- Used to control the display behavior of an element
- Values are

1. block:
    - considers width and height and displays elements on new line
2. inline:
    - ignores the width and height and displays in same line
3. none:
    - hides the element
4. inline-block:
    - considers width and height and displays elements on same line

CSS Float

- The float property in CSS is used to position elements to the left or right of a container, allowing text or other elements to wrap around them.
- values
    1. right:
    2. left:

CSS Flex

- Flexbox (Flexible Box Layout) is a powerful, one-dimensional layout system in CSS designed for organizing elements in rows or columns.
- It simplifies alignment, spacing, and distribution of elements, making it ideal for responsive design.
- To use Flexbox, apply display: flex; to a container.
- This makes all child elements (flex items) automatically adjust according to the rules of Flexbox.
- Flex Container and Flex Items

1. Flex Container: The parent element that holds flex items.
2. Flex Items: The child elements inside the container.

- Below properties apply to the container (display: flex;).

1. flex-direction

1. row: Default. Items are placed left to right.

2. row-reverse: Items are placed right to left.
3. column: Items are placed top to bottom.
4. column-reverse: Items are placed bottom to top.

2. justify-content

1. flex-start: Default. Items align to the start (left).
2. flex-end: Items align to the end (right).
3. center: Items are centered.
4. space-between: First item at start, last item at end, space between them.

CSS3 Advanced Properties

- border-radius
  - Used to add rounded corner
- Shadow
  - Text shadow
  - Box shadow
- Gradients
  - Linear
  - Radial
- column-counts
  - Used to divide an element in number of columns
- CSS Animations
  - Transition
  - Transform
    - scale
    - rotate
    - translate

Agenda

- Media-Queries
- BootStrap
- Breakpoints
- Layout
- Grid
- Components
- JavaScript

BootStrap

- It is the CSS Framework for developing responsive and mobile-first websites.
- Bootstrap 5 is the newest version of Bootstrap

BootStrap Breakpoints

- Breakpoints are customizable widths that determine how your responsive layout behaves across device.
- They are the building blocks of responsive design.
- Use media queries to architect your CSS by breakpoint.
- Media queries are a feature of CSS that allow you to conditionally apply styles based on a set of browser and operating system parameters.
- We most commonly use min-width in our media queries.

Available breakpoints

- Bootstrap includes six default breakpoints, sometimes referred to as grid tiers, for building responsively.

1. Extra Small (None) - <576px (For mobile no prfefix)
2. Small(sm) - ≥576px
3. Medium(md) - ≥768px
4. Large(lg) - ≥992px
5. Extra large(xl) - ≥1200px
6. Extra extra large(xxl) - ≥1400px

Container

- Containers are the most basic layout element in Bootstrap and are required when using our default grid system.
- Containers are used to contain, pad, and (sometimes) center the content within them.
- Bootstrap comes with three different containers:

1. .container, which sets a max-width at each responsive breakpoint
2. .container-{breakpoint}, which is width: 100% until the specified breakpoint
3. .container-fluid, which is width: 100% at all breakpoints
    - a full width container, spanning the entire width with some padding on right and left

Grid System

- Bootstrap’s grid system uses a series of containers, rows, and columns to layout and align content.

1. grid supports six responsive breakpoints.
    - Breakpoints are based on min-width media queries, meaning they affect that breakpoint and all those above it (e.g., .col-sm-4 applies to sm, md, lg, xl, and xxl).
    - This means you can control container and column sizing and behavior by each breakpoint.
2. Containers center and horizontally pad your content.
3. Rows are wrappers for columns.
    - Each column has horizontal padding (called a gutter) for controlling the space between them.
    - This padding is then counteracted on the rows with negative margins to ensure the content in your columns is visually aligned down the left side.
    - Rows also support modifier classes to uniformly apply column sizing and gutter classes to change the spacing of your content.
4. Columns are incredibly flexible.
    - There are 12 template columns available per row, allowing you to create different combinations of elements that span any number of columns.
    - Column classes indicate the number of template columns to span (e.g., col-4 spans four). widths are set in percentages so you always have the same relative sizing.
5. Row columns
    - Use the responsive .row-cols-* classes to quickly set the number of columns that best render your content and layout.
    - Whereas normal .col-* classes apply to the individual columns (e.g., .col-md-4), the row columns classes are set on the parent .row as a shortcut. With .row-cols-auto you can give the columns their natural width.

Gutters

- These are the padding between your columns, used to responsively space and align content in the Bootstrap grid system.
- Gutters are also responsive and customizable. Gutter classes are available across all breakpoints, with all the same sizes as our margin and padding spacing.
- Change horizontal gutters with .gx-* classes, vertical gutters with .gy-*, or all gutters with .g-* classes.*
- .g-0 is also available to remove gutters.
- gx-1(horizintal padding of 2px) and gy-1 (vertical margin of 4px)
- gx-2(horizintal padding of 4px) and gy-2 (vertical margin of 8px)
- gx-3(horizintal padding of 8px) and gy-3 (vertical margin of 16px)
- gx-4(horizintal padding of 12px) and gy-4 (vertical margin of 24px)
- gx-5(horizintal padding of 24px) and gy-5 (vertical margin of 48px)

Margin and Padding

- Assign responsive-friendly margin or padding values to an element or a subset of its sides with shorthand classes.
- Includes support for individual properties, all properties, and vertical and horizontal properties.
- Spacing utilities that apply to all breakpoints, from xs to xxl, have no breakpoint abbreviation in them.
- This is because those classes are applied from min-width: 0 and up, and thus are not bound by a media query.The remaining breakpoints, however, do include a breakpoint abbreviation.
- The classes are named using the format {property}{sides}-{size} for xs and {property}{sides}-{breakpoint}-{size} for sm, md, lg, xl, and xxl.
- Where property is one of:
    - m - for classes that set margin
    - p - for classes that set padding
- Where sides is one of:
    - t - for classes that set margin-top or padding-top
    - b - for classes that set margin-bottom or padding-bottom
    - s - (start) for classes that set margin-left or padding-left in LTR, margin-right or padding-right in RTL
    - e - (end) for classes that set margin-right or padding-right in LTR, margin-left or padding-left in RTL
    - x - for classes that set both *-left and *-right
    - y - for classes that set both *-top and *-bottom
    - blank - for classes that set a margin or padding on all 4 sides of the element
- Where size is one of:
    - 0 - for classes that eliminate the margin or padding by setting it to 0
    - 1 - (by default) for classes that set the margin or padding to $spacer * .25
    - 2 - (by default) for classes that set the margin or padding to $spacer * .5
    - 3 - (by default) for classes that set the margin or padding to $spacer
    - 4 - (by default) for classes that set the margin or padding to $spacer * 1.5
    - 5 - (by default) for classes that set the margin or padding to $spacer * 3
    - auto - for classes that set the margin to auto
- spacer is calculated in terms of rem(root em) an unit of measurement in css.
- 1 rem is 8px for padding and 16px for margin

Typography

- arranging text to make it readable, clear, and visually appealing
- we can change the text font, style, color of the text
- To use heading font in the paragraph or div use class h1..h6
- Traditional heading elements are designed to work best in the meat of your page content.
- When you need a heading to stand out, we can use a display heading display1..display6

- All the text related font, style and color information is available under utilities section in the documentation

Flex

- It manages the layout, alignment, and sizing of grid columns, navigation, components, and more with a full suite of responsive flexbox utilities.
- Apply display utilities to create a flexbox container and transform direct children elements into flex items.
- Flex containers and items are able to be modified further with additional flex properties.
- Responsive variations for flex are .d-flex and .d-inline-flex.
- Set the direction of flex items in a flex container with direction utilities.
- Use .flex-row to set a horizontal direction (the browser default), or .flex-row-reverse to start the horizontal direction from the opposite side.
- Use .flex-column to set a vertical direction, or .flex-column-reverse to start the vertical direction from the opposite side.
- Use justify-content utilities on flexbox containers to change the alignment of flex items on the main axis (the x-axis to start, y-axis if flex-direction: column). Choose from start (browser default), end, center, between, around, or evenly.
- Use align-items utilities on flexbox containers to change the alignment of flex items on the cross axis (the y-axis to start, x-axis if flex-direction: column). Choose from start, end, center, baseline, or stretch (browser default).

JavaScript

- Invented by Brendan Eich in 1995 at Netscape Corporation for Netscape2
- It is a Scripting language for web development
- It is an interpreted langugage
- Used to develop server side programs (Node) as well
- It is an Object Oriented Programming Language
- Use script tag to write JS code in html page
- Loosely typed language
- It is used to dynamically modify the html pages.It has full integration with HTML/CSS
- All major browsers support and by default enabled for javascript
- ECMAScript is the official name of the language.
- ECMAScript versions have been abbreviated to ES1, ES2, ES3, ES5, and ES6.
- Since 2016, versions are named by year (ECMAScript 2016, 2017, 2018, 2019, 2020).

Types of javascript

1. Internal
    - It is inserted into the documents by using the script tag
    - spript tag provides a block to write the java script programs

```
<script>
    JS code goes here
</script>
```

2. External
    - To use the predefined programs of any javascript library.

```
<script src = "myscript.js"></script>
```

Built-in Objects

1. console
    - represents the web console (terminal)
    - use log method to write output on console
2. window
    - used to dispay alert,prompt or confirmation
3. document:
    - represents DOM (Document Object Model)
    - Collection of objects of all the elements present inside the page

Variables

- It is a container to store the data
- To declare a variable data type MUST NOT be used in its declaration
- To create variables in JS we can use

1. var
    - It is a normal variable whose value can be changed multiple times
    - var is not used as it is depricated, it is recommend to use let instead
2. let
    - It is a normal variable whose value can be changed multiple times
3. const
    - It is a variable whose value cannot be changed once initialized.

- Syntax: let name = initial value; const name = initial value; E.g.

```
let num = 100; // number
const salary = 4.5; // number
let = “test”; // string
const firstName = ‘steve’; // string
let canVote = true; // Boolean
```

Data Types

- In JS, all Data Types are inferred (automatically decided by JS .....)
- Types

1. number:
    - It supports both whole and decimal numbers

    - E.g.
        - num = 100;
        - salary = 4.5;

2. string: collection of characters E.g. - firstName = “steve”; - lastName = ‘Jobs’;
3. boolean:
    - may have only true or false value
    - E.g.
        - canVote = true;
        - canVote = false;
4. undefined:
5. object:

Built-in Values

1. NaN
    - Not a Number
    - Is of type number
    - E.g. console.log(parseInt(“test”));
2. Infinity:
    - When a number is divided by 0
    - E.g. answer = 10 / 0; // Infinity
3. undefined:

Agenda

- Language Fundamentals
- Functions
- Object
- Class
- ~~Array~~

Variable Scope

1. global
    - a variable declared outside any function
    - can be declared with or without var keyword
    - can be accessed outside or inside any function
    - E.g.

```
num = 100;
var salary = 4.5;
```

    - can be declared inside a function without using a var keyword
    - E.g.

```
function function1() {
// global
firstName = “test”;
}
```

2. Local
    - Must be declared inside a function with keyword var
    - Can NOT be accessed outside the function in which it is declared
    - E.g.

```
function function1() {
// local
var firstName = “test”;
}
```

Function Alias

- Another way/name to call a function
- Syntax: var = ;
- E.g.

```
function function1() {
console.log(“inside function1”);
}

// function alias
var myFunction1 = function1;
myFunction1();
```

Anonymous function

- Function without a name is called as anonymous function
- Syntax: var = function() { // body }
- E.g.

```
var multiply = function(p1, p2) {
console.log(“p1 * p2 = “ + (p1 * p2));
}
```

Properties of function

- A function CAN NOT decide the data type of parameters
- Only caller decides the data type of parameters
- E.g.

```
function function1 (p1) {
console.log(“p1 = ”  + p1 + “ type = “ + typeof(p1));
}

function1(10); // number
function1(“test”); // string
function1(true); // boolean
```

- If a program contains multiple functions with same name then only bottom-most function’s definition will be used
- E.g.

```
function function1 () {
console.log(“function1 - 1”);
}

function function1 () {
console.log(“function1 - 2”);
}

function function1 () {
```

```
console.log("function1 - 3");
}

// output: function1 - 3
function1();
```

- A function can be called with excess number of parameters
- Excess parameters can be used by using a hidden parameter arguments
- E.g.

```
function function1 (p1, p2) {
// body
}
function1(10, 20, 30, 40, 50); // p1 = 10, p2 = 20
```

- A function can be called with less number of parameters
- E.g.

```
function function1 (p1, p2) {
// body
}
function1(); // p1 = undefined, p2 = undefined
function1(10); // p1 = 10, p2 = undefined
function1(10, 20); // p1 = 10, p2 = 20
```

- A function can be called before its declaration
- E.g.

```
function1();
function function1() {
// body
}
```


Object

- collection of properties (data members or fields) and methods
- Everything in JS is an Object, even functions are also objects
- To create an object (instance)
  - Use Object Literal ({})
  - Use new keyword
  - Use constructor function

```

```

```
// using Object
var c1 = new Object();
c1.model = "i10";
c1.company = "Hyundai";

// using construction function
function Car(model, company) {
    this.model = model;
    this.company = company;
}
var c2 = new Car("Fabia", "Skoda");

// using Object Literal
// also called as JSON
var c3 = {
    model: "X5",
    company: "BMW"
};

var cars = [c1, c2, c3];
for (var index = 0; index < cars.length; index++) {
    var car = cars[index];
    console.log("Model: " + car.model);
    console.log("Company: " + car.company);
}
```

Function alias

- We can create an alias for our function.
- declare a varaible and initialize it with the existing function which will create an alias for that function

```
function function1() {
    console.log("Inside function1")
}
function1()

// function alias
var myfunction1 = function1
myfunction1()
```

Agenda

- Array
- DOM

Array

- An array in JavaScript is a special variable that can hold multiple values.
- Arrays allow you to store, access, and manipulate lists of data efficiently.
- array declaration can be done in two ways:

1. Using Square Brackets (Recommended)
2. Using the Array Constructor

```
let fruits = ["Apple", "Banana", "Mango"];
let fruits = new Array("Apple", "Banana", "Mango");
```

- Square brackets [] are preferred because they are more concise.
- Array elements are accessed using index numbers, starting from 0.

Window Object

- It represents an open window in the browser. It is browser's object(not JS object) which is created automatically
- It is a global object with lot of properties and methods

DOM (Document Object Model)

- When a webpage is loaded the browser creates DOM of the page
- It is the data representation of the objects that comprise the structure and content of a document on the web.
- DOM represents an HTML document in memory
- The DOM represents the document as nodes and objects so that programming languages can interact with the page.
- In both cases, it is the same document but the Document Object Model (DOM) representation allows it to be manipulated.
- console.dir(document) will display all the properties and methods from the document
- It is a tree like structure (window-> document -> html -> and all its sub nodes)

DOM Manipulation

Selection

1. Selecting with id
    - document.getElementById("myId") (#)
2. Selecting with class
    - document.getElementsByClassName("myClass") (.)

    - returns HTML collection an array of objects
3. Selecting with tag
    - document.getElementsByTagName("tagName")
    - returns HTML collection an array of objects
4. Query Selector
    - used to select the id,name and class automatically
    - document.querySelector("myId/myClass/tag")
        - returns first element
    - document.querySelectorAll("myId/myClass/tag")
        - returns a NodeList

Properties

1. tagName
    - returns tag for element nodes
2. innerText
    - returns text content of the element and all its children
    - It represents only the text part
3. innerHTML
    - returns the plain text or html contents in the elements
    - It represents text as well as any element/tag inside sit
4. textContent
    - returns textual content even for hidden elements

Attribute

1. getAttribite("attr")
    - to get the attribute value
2. setAttribute("attr",value)
    - to set the attribute value

Style

node.style - It helps to style the elements i.e apply css on it

Insert elements

1. node.append(e)
    - add at the end of the node (inside)
2. node.prepend(e)
    - add at the start of the node (inside)
3. node.before(e)

- add before the node (outside)

4. node.after(e)

- add after the node (outside)

Delete elements

- node.delete(e)
    - Used to delete the node

Agenda

- jQuery
- AJAX

jQuery

- jQuery is a fast, small, and feature-rich JavaScript library.
- It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers.
- It is a lightweight, "write less, do more", JavaScript library.
- With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript.
- The jQuery library contains the following features:

1. HTML/DOM manipulation
2. CSS manipulation
3. HTML event methods
4. Effects and animations
5. AJAX

Adding jQuery

- The jQuery library is a single JavaScript file, and you reference it with the HTML `<script>` tag
- make sure that the script tag should be inside the head section ``HTML

```

# Basic Syntax

- $(selector).action()
    - $ sign to define/access jQuery
    - (selector) to "query (or find)" HTML elements
    - action() to be performed on the element(s)
- Example
    - $(this).hide() - hides the current element.
    - $("p").hide() - hides all

      elements.
    - $(".testclass").hide() - hides all elements with class="testclass".
    - $("#testId").hide() - hides the element with id="testId".

# AJAX
```

- AJAX stands for Asynchronous JavaScript and XML. - It is a technique used in web development to update parts of a web page without reloading the entire page.
- By using AJAX, web applications can send and retrieve data asynchronously from a server in the background, improving the user experience by making the app feel faster and more interactive.
- The basic flow of an AJAX request:

1. User Interaction:
    - A user performs an action (e.g., clicking a button).
2. JavaScript Sends a Request:
    - JavaScript sends an HTTP request to the server using the XMLHttpRequest object or the newer fetch API.
3. Server Processes the Request:
    - The server processes the request (e.g., fetches data from a database) and sends back a response.
4. JavaScript Updates the Web Page:
    - JavaScript processes the server's response and updates the web page dynamically, without requiring a page reload.

JSON

- JSON (JavaScript Object Notation) is a lightweight data format used for storing and exchanging data between a server and a web application.
- It is easy to read and write for humans and machines.
- JSON data is written in key-value pairs and uses {} for objects and [] for arrays.
- the data in the form of JSON is converted entirely in the string (using JSON.stringfy()) and then sent through the request or response.
- once we receive the request on server or the response on client we convert the string data back into JSON format (using JSON.parse())

```
{
  "name": "John Doe",
  "age": 30,
  "email": "john@example.com",
  "isStudent": false
}

[
  {
    "id": 1,
    "name": "Alice"
  },
  {
    "id": 2,
    "name": "Bob"
```

```
  }
]
```

XHR (XMLHttpRequest)

- It is a JavaScript API that allows developers to make HTTP requests (to fetch data, send data, etc.) from a web server without reloading the page.
- It is commonly used for building AJAX functionality.

Steps to Use XHR

1. Create an XHR Object:
    - Use let helper = new XMLHttpRequest();.
2. Configure the Request:
    - Specify the request method (GET, POST, etc.) and the URL
    - helper.open(method, url, async).
    - async:Boolean, true (asynchronous, default) or false (synchronous).
3. Send the Request:
    - helper.send() to send the request.
4. Handle the Response:
    - helper.onreadystatechange or helper.onload to process the server's response.

using jQuery Ajax method

Promise

- A Promise in JavaScript is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value.
- Promises were introduced to solve the lot of callback problem, which occurs when multiple asynchronous operations are nested, leading to messy, hard-to-read, and error-prone code.
- A Promise has three possible states:

1. Pending: The initial state when the Promise is neither fulfilled nor rejected.
2. Fulfilled: The operation completed successfully, and the Promise has a resolved value.
3. Rejected: The operation failed, and the Promise has a reason (error).

- Once a Promise is either fulfilled or rejected, it becomes settled, and its state will not change again.

Fetch API

- It is a modern alternative to XHR.
- Fetch uses Promises, making the code cleaner and easier to manage.
- Easier to read and manage with .then() and .catch() or async/await.
- No need for xhr.open(), xhr.send(), or xhr.onload.

- All configurations are included in the fetch() function.
- Use .catch() to handle errors, such as network issues or HTTP errors.
- Built-in JSON Support: Use .json() to parse JSON responses easily.

```
//using then() and catch()
function btn1Clicked() {
fetch("http://127.0.0.1:5500/data.json", { "method": "GET" })
.then((response) => {
    if (response.ok)
        return response.json()
    else
        throw new Error("Something went wrong")
})
.then((data) => {
    console.log(data)
})
.catch((error) => {
    console.log("error = " + error)
})
}

//using async/await
async function btn2Clicked() {
try {
    let result = await fetch("http://127.0.0.1:5500/data.json", { "method": "GET"
})
    if (result.ok) {
        const data = await result.json()
        console.log(data)
    }
    else
        throw new Error("Something went wrong")
} catch (error) {
    console.log("error -" + error)
}
}
```


- Both async/await and .then() are ways to handle Promises in JavaScript, but they differ in syntax, readability, and how they handle asynchronous code.
- Which One to Use?

1. Use Async/Await:

- When you have multiple asynchronous tasks that need to be executed in sequence.
- When you want better readability and maintainability.
- When you want to handle errors more cleanly with try/catch.

2. Use .then():

- When you're working with simpler tasks (single or few Promises).
- When you need backward compatibility (older browsers without async/await).

Agenda

- NodeJs
- ~~Express~~

NodeJs

- A software platform that is used to build scalable network applications.
- It is developed by Rayn Dahl in 2009
- It is a javascript runtime built on Google's V8 JavaScript Engine
- Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.
- A Node.js app runs in a single process, without creating a new thread for every request.
- It provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.
- When an I/O operation like reading from the network, accessing a database or the filesystem, then instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.
- This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.
- It has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.
- In Node.js the new ECMAScript standards can be used without problems.

Components of NodeJs

1. JavaScript runtime: V8
2. EventLoop : Libuv (A multi-platform support library which focuses on asynchronous I/O, primarily developed for use by Node.js.)
3. Standard Components
    - Filesystem Access
    - Crypto
    - TCP/UDP
    - HTTP
    - Buffer
    - etc

Use Cases of NodeJs

- Where to use
    - I/O bound applications
    - Data Streaming applications
    - IoT applications
    - JSON API based applications
    - Single Page applications

- Where not to use
    - CPU intensive applications
    - Heavy server sided processing

Installation

1. click on the below link to download the node

```
https://nodejs.org/en/download
```

```
# to check for the installed node version
node --version

# to run the node application
node hello.js
```

JSON (JavaScript Object Notation)

- Created by Douglas Crockford is defined as a part of javascript langugage in the early 2000s.
- It wasn't untill 2013 that the format was offically specified
- Javascript Objects are simple associative containers where a string key is mapped with a value.
- JSON shows up in many different cases.

1. API
2. Configuration Files
3. Log Messages
4. Database storage

Module

- It is a collection of functions
- We can share a javascript library with multiple applications with the help of this module
- Their are two types of modules

1. Built-in Modules
    - OS
    - FileSystem
    - HTTP
2. User defined Modules

REST (Representational State Transfer) Services

- It is a design pattern
- A request is sent and response is received
- response is always in the form of JSON
- request consists of 2 parts atleast

1. HTTP method (get,post,put,delete) operation
    - get used to select
    - post used to submit
    - put used for update/edit
    - delete used to remove
2. path

Agenda

- Express
- Mysql
- ~~Crypto Js~~
- ~~JWT~~

nodemon

- It is a Node monitor which is used to monitor changes in server and refreshes the server

Express

- Developed by many developers around the world
- It is a fast,minimilistic web framework for nodejs
- It is a lightweight and flexible routing framework with minimal core features meant to be augmented through the use of Express middleware modules.

```
# to start using express application install express using npm or add using yarn
npm install express
# OR
yarn add express
```

Routing

- Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).
- Each route can have one or more handler functions, which are executed when the route is matched.
- Route definition takes the following structure:

```
app.METHOD(PATH, HANDLER)
```

- Where:

1. app is an instance of express.
2. METHOD is an HTTP request method, in lowercase.
3. PATH is a path on the server.
4. HANDLER is the function executed when the route is matched.

- we define routing using methods of the Express app object that correspond to HTTP methods
- for example, app.get() to handle GET requests and app.post to handle POST requests.
- These routing methods specify a callback function (sometimes called “handler functions”) called when the application receives a request to the specified route (endpoint) and HTTP method.
- In other words, the application “listens” for requests that match the specified route(s) and method(s), and when it detects a match, it calls the specified callback function.

express.Router

- Use the express.Router to create modular, mountable route handlers.
- A Router instance is a complete middleware and routing system.
- for this reason, it is often referred to as a “mini-app”.

Middleware

- Middleware is a function that runs between the request and the response in a web application.
- It is commonly used in backend frameworks like Express.js to modify, process, or control HTTP requests before sending a response.
- It Access Request (req) and Response (res) Objects
- It Modify Requests/Responses → Example: Adding headers, parsing JSON
- Run Code Before Sending a Response → Example: Authentication, Logging
- Call next() to Continue Execution → Passes control to the next middleware
- It is commonly used in express for

1. Handling CORS (cors package)
2. Parsing JSON data (express.json())
3. Authentication & authorization

Connecting with mysql database

1. add/install the mysql module

```
npm install mysql2
```

2. Create the pool object

```
const pool = mysql.createPool({
    host: 'localhost',
    user: 'root',
    password: 'root',
    database: 'kdac_db'
})
```

3. Call the query method

```
const sql = `SELECT * FROM product`
pool.query(sql, (error, data) => {
    if(data)
        response.send(data)
    else
        response.send(error)
})
```

Agenda

- Middleware
- Crypto-Js
- JWT

Middleware

- Middleware is a function that runs between the request and the response in a web application.
- It is commonly used in backend frameworks like Express.js to modify, process, or control HTTP requests before sending a response.
- It Access Request (req) and Response (res) Objects
- It Modify Requests/Responses → Example: Adding headers, parsing JSON
- Run Code Before Sending a Response → Example: Authentication, Logging
- Call next() to Continue Execution → Passes control to the next middleware
- It is commonly used in express for

1. Handling CORS (cors package)
2. Parsing JSON data (express.json())
3. Authentication & authorization

crypto-js

1. add/install the mysql module

```
npm install crypto-js
#OR
yarn add crypto-js
```

2. Usage

```
// import the module crypto-Js
const cryptoJs = require("crypto-js")

// use the encryption method to encrypt the password
const encrypted = cryptoJs.SHA256(password)
```

JWT

- JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.
- This information can be verified and trusted because it is digitally signed.
- JWTs can be signed using a secret
- These are useful in

1. Authorization
    - This is the most common scenario for using JWT.
    - Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token.
2. Information Exchange
    - JSON Web Tokens are a good way of securely transmitting information between parties.
    - Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are.
    - Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

JSON Web Token structure

In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:

1. Header
    - The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used.
2. Payload
    - The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data.
3. Signature
    - To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that. Therefore, a JWT typically looks like the following:

```
xxxxx.yyyyy.zzzzz
```

- Putting all together the output is three Base64-URL strings separated by dots that can be easily passed in HTML and HTTP environments.

Encoding and Decoding a JWT

- Encoding a JWT involves transforming the header and payload into a compact, URL-safe format.
- The header, which states the signing algorithm and token type, and the payload, which includes claims like subject, expiration, and issue time, are both converted to JSON then Base64URL encoded.
- These encoded parts are then concatenated with a dot, after which a signature is generated using the algorithm specified in the header with a secret or private key.
- This signature is also Base64URL encoded, resulting in the final JWT string that represents the token in a format suitable for transmission or storage.
- Decoding a JWT reverses this process by converting the Base64URL encoded header and payload back into JSON allowing anyone to read these parts without needing a key.
- However, "decoding" in this context often extends to include verification of the token's signature.

- This verification step involves re-signing the decoded header and payload with the same algorithm and key used initially, then comparing this new signature with the one included in the JWT.
- If they match, it confirms the token's integrity and authenticity, ensuring it hasn't been tampered with since issuance.

JWT installation and usage

1. add/install the jwt module

```
npm install jsonwebtoken
#OR
yarn add jsonwebtoken
```

2. Usage

```
// import the module jwt
const jwt = require("jsonwebtoken")

// create token
const token = jwt.sign({id:'234336653'},'secret')

//verify the token
const decoded = jwt.verify(token,'secret')
```

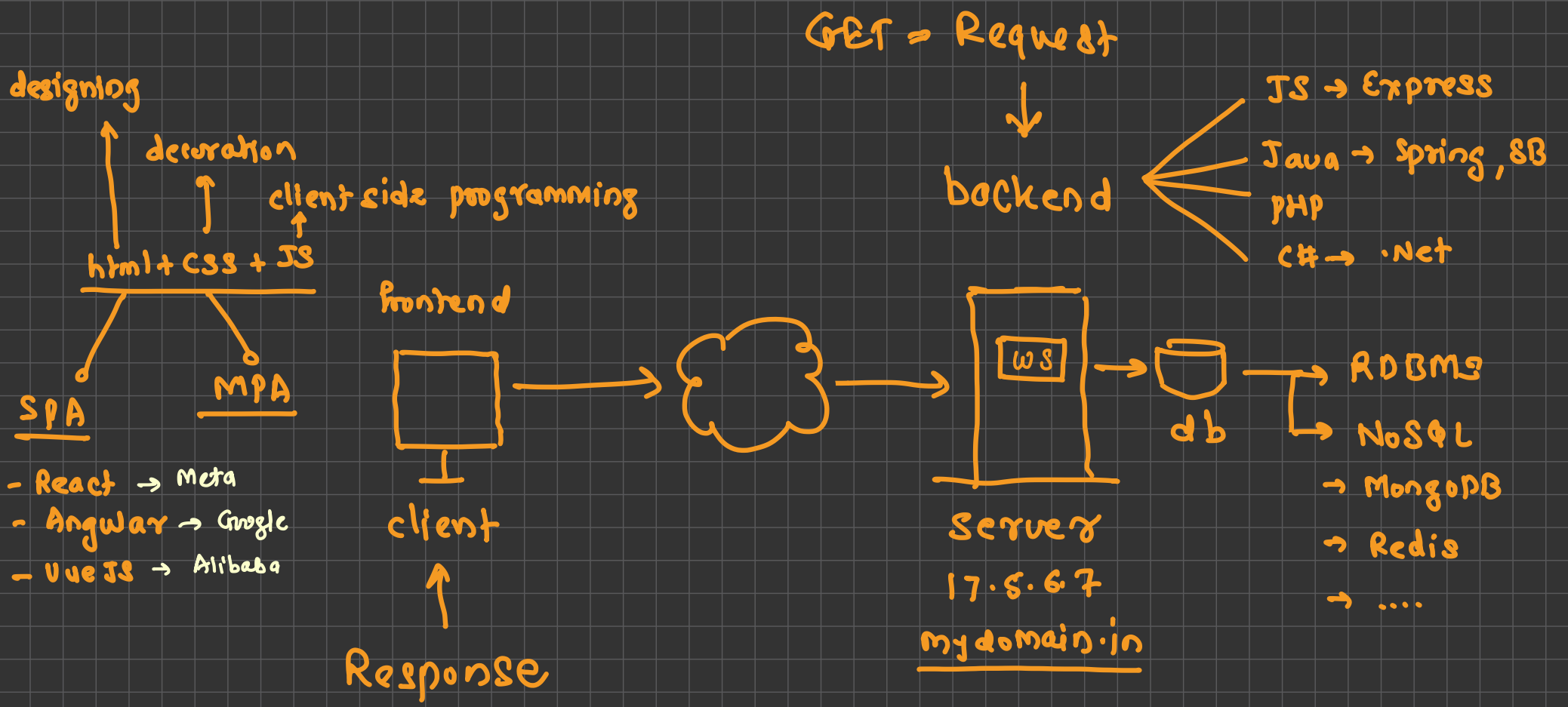
Server communication

- → REST → JSON | XML | YAML → design Pattern
- → SOAP → Simple object Access Protocol → Protocol
- → GraphQL → Graph Query Language ***
- → gRPC → google Remote Procedure Call → protobuff ***
- → WebSocket → Uses Socket (socket.io)

React

GET http://mydomain.in/mypage.php

mypage.php → PHP interpreter → html/css + JS



Server

→ always a software or application

→ types

1] Web servers → used to serve http/https requests
   eg. apache, nginx, tomcat, express, MS IIS

2] DB server → used to persist the data
   eg. RDBMS (mysql, sql server, postgre, oracle)
   NoSQL (mongoDB, redis, cassandra, firebase)

3] DNS server → used to get IP address from a domain name
   eg. Bind

4] SMTP → used to send emails [Simple Mail Transfer Protocol]
   eg. postfix

5] POP → used to receive emails. [Post office Protocol]
   eg. Dovecot

6] File server → use to serve files
   eg. FTP → File Transfer Protocol → windows/linux/mac
   samba [smb] - Server Message Block] → windows
   NFS → Network File System → Linux

Contents

- Introduction
  - SPA, advantages, use cases
- Why React?
  - what ? why ? when ? How ?
- Using JSX ***
  - variables, objects, arrays →
  - JSX = JS + XML (HTML)
- Class based Components
  - component → reusable entity used to create UI
  - stateFull
- Functional Components ***
  - Stateless → hooks → state
- React Props
  - properties → metadata used in a component
  - parent → send data child
  - Read only
- React State
  - Used to store data inside a component
  - Read Writable
- React Hooks
  - useNavigation, useSelector(), useDispatch()
  - useState(), useEffect(), useLocation(),
  - special function starts with 'use'
- Intro to Redux
  - global store management
- Redux vs Context
  - Context API → useContext(), createContext()
- Handling User Inputs
  - input, select, textarea, form
- React Router
  - used for switching between components
- Advanced Router
  - Nested routing, parameterized routing
- Consuming REST APIs
  - Fetch, Axios → Representational State Transfer
- Uploading Files
  - form Data → sending a file
- Styled Components
  - component + style (CSS) - Reusability
- Authentication
  - JWT tokens →
- Authorization
  - session storage + local storage
- Session Handling
  - skip and limit on server + API
- Pagination
  - make app accessible to end user → deployment
- Cloud Deployment
  - AWS

About Instructor

- 16+ years of experience
- Associate Technical Director at Sunbeam
- Freelance Developer working in various domains using different technologies
- Developed 180+ mobile applications on iOS and Android platforms
- Developed various websites using PHP, MEAN and MERN stacks
- Languages I love and use in every programming: C, C++, Python, JavaScript, TypeScript, PHP

Golang, Rust,

Introduction

- React, also known as ReactJS, is a popular and powerful JavaScript library used for building dynamic and interactive user interfaces, primarily for single-page applications (SPAs)
- It was developed and maintained by Facebook and has gained significant popularity due to its efficient rendering techniques, reusable components, and active community support
- React is a declarative, component based library that allows developers to build reusable UI components and It follows the Virtual DOM (Document Object Model) approach, which optimizes rendering performance by minimizing DOM updates. React is fast and works well with other tools and libraries
- Prerequisite of React
  - For learning React first you have a clear understanding of HTML, CSS and JavaScript
  - As React is a JavaScript library and uses most of its concept so you really have to understands the major concepts of it
  - HTML and CSS
  - JavaScript and ES6
  - JSX (JavaScript XML)
  - Node + NPM
  - Git

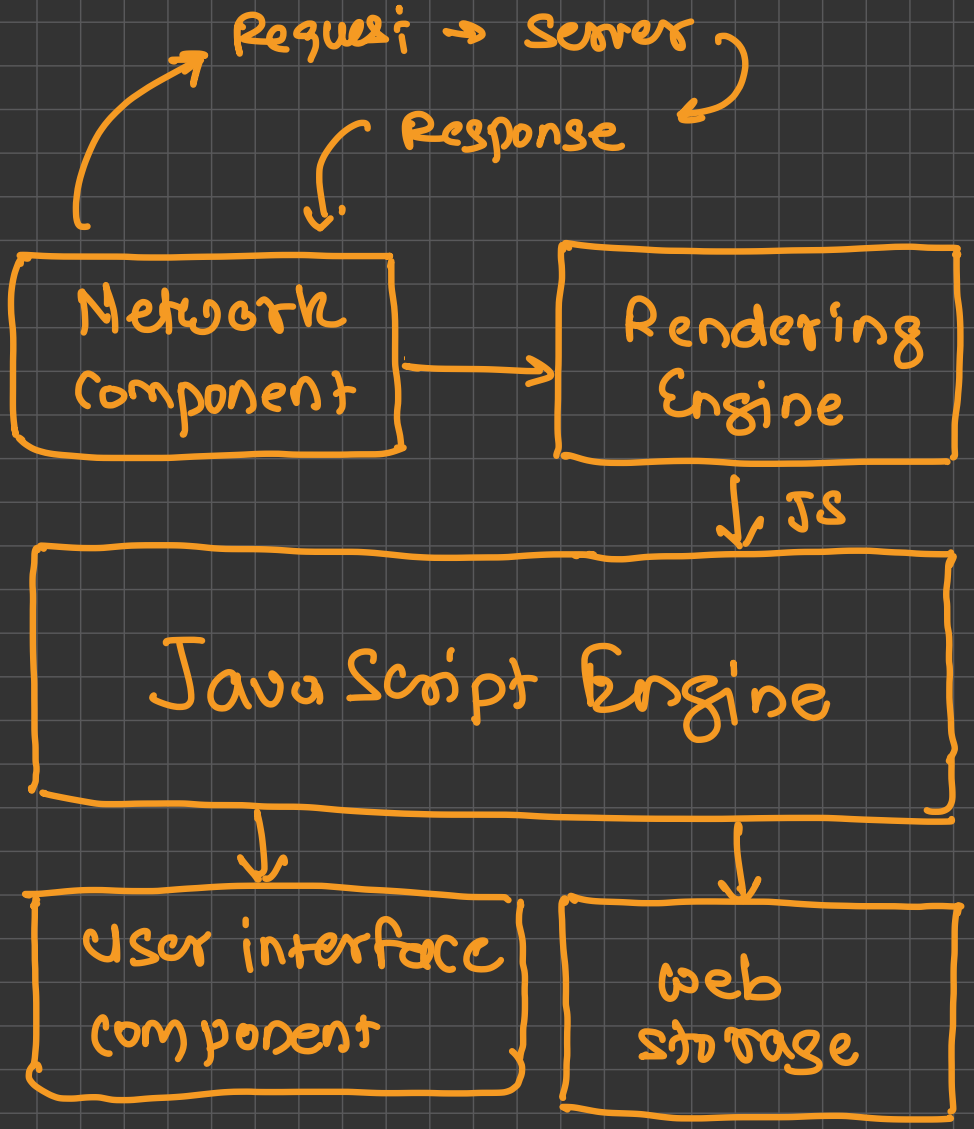
less memory footprint → requires less memory

faster

DOM → tree like structure of objects of each HTML element in a page
→ JS → document object
→ created by Rendering Engine

→ functions
→ promises
→ function References
→ functional programming
→ Rest operator
→ destructuring

Browser Architecture → based on JS Engine



* Network → used to communicate with server (sending Request, and receiving response)

* Rendering Engine → used to convert HTML/CSS to JS objects, it builds the DOM of the website. Also known as layout Engine.

* JS Engine → used to execute JS code and produce the O/P

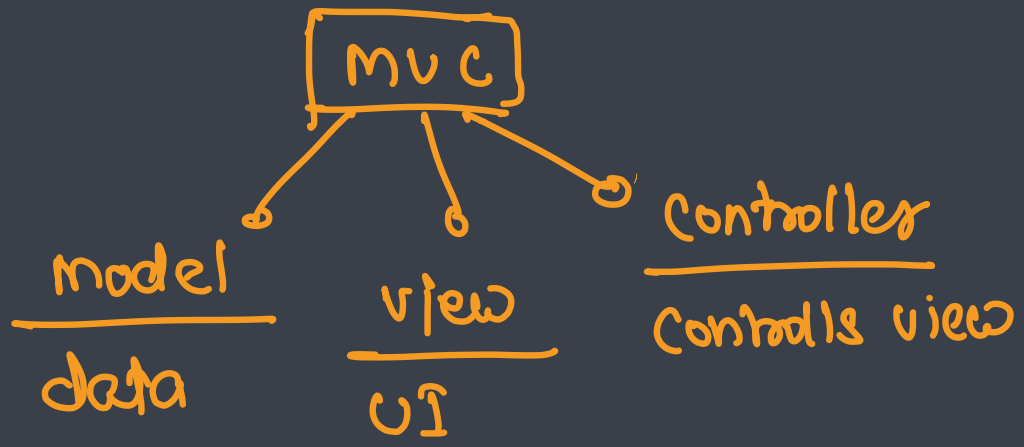
* UI component → used to load the user interface (output of WS)

* Web storage component → used to store the data on client.
eg. history, session storage, local storage

	Rendering Engine	Javascript Engine
Edge	EdgeHTML	chakra
Safari	Webkit	Nitro JS
Firefox	Gecko	SpiderMonkey
Chrome	Blink	V8 → C++

History

- It was created by **Jordan Walke,** who was a software engineer at **Facebook**
- It was initially developed and maintained by Facebook and was later used in its products like **WhatsApp** & **Instagram**
- Facebook developed ReactJS in **2011** in its newsfeed section, but it was released to the public in the month of **May 2013**
- Today, most of the websites are built using MVC (model view controller) architecture
- In MVC architecture, React is the 'V' which stands for view, whereas the architecture is provided by the Redux or Flux



Features

- **Declarative UI**
  - React allows developers to design user interfaces in a declarative way
  - Developers describe what the UI should look like for any given state, and React updates the DOM to match that state automatically
- **Component-Based Architecture**
  - Applications in React are built as a collection of small, reusable components
  - Encourages modularity
  - Makes code reusable and easier to test and maintain
- **JSX (JavaScript XML)**
  - React uses **JSX**, a syntax extension that lets you write HTML-like code inside JavaScript
  - Improves readability and allows developers to embed JavaScript expressions directly within the markup
- **Virtual DOM**
  - React uses a **virtual DOM**, a lightweight copy of the actual DOM
  - Efficiently updates the real DOM by minimizing changes
  - Enhances performance, especially in dynamic applications
- **Unidirectional Data Flow**
  - React enforces a **unidirectional data flow**, meaning data flows in a single direction (from parent to child)
  - Makes debugging easier and improves control over how data is passed and managed

Features

- React Hooks
  - Hooks are functions like useState and useEffect that allow function components to use React features such as state and lifecycle methods
  - Simplifies code by reducing the need for class components and makes managing state and side effects straightforward
- React Router
  - React Router is used for implementing dynamic routing in React applications
  - Allows the creation of single-page applications (SPAs) with seamless navigation
- Context API
  - The Context API enables global state management without needing third-party libraries like Redux
  - For managing themes, authentication, or other data shared across multiple components
- Code Splitting and Lazy Loading
  - React supports code splitting through React.lazy() and dynamic imports
  - Loads only the required code for a specific page or feature
  - Reduces initial load time and improves performance
- Server-Side Rendering (SSR)
  - With frameworks like Next.js, React supports server-side rendering
  - Improves SEO and reduces time-to-interactive for end users
- Performance Optimization
  - React offers built-in tools and techniques for optimization
  - Concurrent Rendering: React 18 introduced concurrent rendering to handle complex UIs efficiently
- Cross-Platform Development
  - With React Native, developers can build mobile applications using React
  - Share code between web and mobile platforms

Advantages

- Easy to learn and use
- Creating dynamic web application becomes simpler
- Reusable components
- Performance enhancements
- Support of handy tools and libraries
- Benefits of being a JS library
- Easy to unit test the application

Disadvantages

- The high pace of development
- Poor documentation
- Its only about the View
- Learning curve for JSX

React

Single Page Application (SPA)

- a application that loads a single HTML page and dynamically updates that page as the user interacts with the app
- to develop SPAs,
    - we need to use a JavaScript framework or library
    - like
        - React
        - Angular
        - VueJs
- advantages
    - fast: similar performance to native apps
    - responsive: the app responds to user interactions (browser size changes),
        - to make the app responsive
            - we need to use CSS media queries
            - frameworks: bootstrap, tailwind
    - user-friendly

functional programming language

- function is considered as first class citizen
    - function is created as a variable of type function
- function can be passed as an argument to another function
- function can be returned from another function as return value
- map()
    - used to iterate over a collection to transform the values to new ones
    - accepts a function as a parameter which gets called every time for every value
    - the parameter function must return a transformed value for original value
    - all the transformed values will be returned a collection as a return value of map function
    - the size of returned collection is always same as original collection

```
// array of numbers
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

// get squqre of each number
const squares = numbers.map((number) => number ** 2)
```

function reference

- a reference to a function
- a variable that holds a function body's address

```
// here the function1 is a function reference
// to the function body
function function1() {
  console.log('inside function1')
}
```

export and import

- export
    - used to export any entity from a file for others to import
    - a file can export multiple entities for others

    ```
    // App.jsx
    export function App() {
      ...
    }

    // main.jsx

    // importing with same name as that of the exported entity
    import {App} from './App.jsx'

    // importing with an alias
    import {App as MyApp} from './App.jsx'
    ```

- export default
    - by default only one entity (class, function, variable, constant) can be exported from a file with default keyword

    ```
    // App.jsx
    function App() {
      ...
    }
    export default App

    // main.jsx

    // importing with same name as that of exported entity
    import App from './App.jsx'

    // importing wth an alias
    import MyApp from './App.jsx'
    ```

React

- a JavaScript library for building user interfaces

React vs Angular

- React is a Library (developed in JS) and Angular is a Framework (developed in TypeScript)
- react development and performance is faster than angular
    - to make it faster, react uses a virtual DOM
    - it also has less memory consumption/footprint
- React has less learning curve than Angular
- React does not have any architecture whereas, Angular has a predefined architecture and tooling

important points

- do not use `class` as it is a reserved keyword in JavaScript, use `className` instead
- interpolation is done using `{}` in JSX
- interpolation always requires a scalar value and CAN NOT render an object

virtual DOM

- a lightweight copy of the real DOM (browser DOM) (document object)
- react uses virtual DOM to improve performance
- when we update the state of a component, react creates a new virtual DOM and compares it with the previous virtual DOM
- then it updates only the changed parts of the real DOM
- this process is called reconciliation

environment setup

- using CDN links
    - CDN: content delivery network
    - add react using CDN links

```
<html>
  <head>
    <!-- used for react development -->
    <script
      crossorigin
      src="https://unpkg.com/react@18/umd/react.development.js"
    ></script>

    <!-- used for react virtual dom development -->
    <script
      crossorigin
      src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"
```

```
    ></script>
  </head>

  <body>
    <div id="root"></div>
  </body>
</html>
```

- to create element use a function called `createElement`

```
// create element
// React is an object which will be used to create elements
// this object is provided by react library
(react.development.js)

// parameters
// 1st: name or type of the element (e.g. h1, h2 etc)
// 2nd: attributes or properties of the element (e.g. class, id,
style etc). this must be an object.
// 3rd: contents of the element (e.g. text, html etc)
const h2 = React.createElement('h2', {}, 'hello world')

// React 17 style of rendering an element
// get the root element
// this is the element where we will render our react elements
// const root = document.getElementById('root')

// render the element
// ReactDOM.render(h2, root)

// React 18 style of rendering an element

// create a root element
const root = ReactDOM.createRoot(document.getElementById('root'))

// render the element
root.render(h2)
```

- to create an element using JSX, use babel

```
<html>
  <head>
    <!-- used for react development -->
    <script
      crossorigin
      src="https://unpkg.com/react@18/umd/react.development.js"
    ></script>

    <!-- used for react virtual dom development -->
```

```
    <script
      crossorigin
      src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"
    ></script>

    <!-- babel compiler -->
    <script
src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>

  <body>
    <div id="root"></div>
    <script type="text/babel">
      const h2 = <h2>hello world</h2>
      const root =
ReactDOM.createRoot(document.getElementById('root'))
      root.render(h2)
    </script>
  </body>
</html>
```

- using package manager like vite

```
# install yarn on windows
> npm install -g yarn

# install yarn on linux or mac
> sudo npm install -g yarn

# create react application using vite
> npm create vite@latest <application name>
> yarn create vite <application name>

# go to the project directory
> cd <application name>

# install the dependencies
> npm install
> yarn

# start the application
> npm run dev
> yarn dev
```

    - project structure
        - node_modules

        - contains all the modules (dependencies) which are required to develop or run the application
        - will be downloaded every time when npm install or yarn install command is used
        - never commit this directory in your git repository
    - public
        - directory which contains public files
        - e.g. images, audio or video which are used in the application
    - src
        - directory which contains all the components of the application
        - contains
            - assets
                - directory which contains the assets (images, audio or video files)
            - main.jsx
                - contains the code to start react subsystem
                - contains the function createRoot(..).render()
            - index.css
                - contains global css rules which can be shared across the components in the application
            - App.jsx
                - contains the default component called as App
                - this is the startup component of every application
            - App.css
                - contains css rules need to applied on the App component
    - .gitignore
        - file which contains the rules of the files which needed to be committed in the git repository
    - eslint.config.js
        - contains configuration for eslint
        - lint is a program used to check the syntax of a selected language
    - index.html
        - only html file in the project which starts the application
    - package.json
        - contain the node configuration like name, dependencies or devDependencies etc.
    - vite.config.js
        - contains the vite configuration
    - yarn.lock
        - contains latest versions of the dependencies installed in the node_modules directory

react application startup

- vite will start a lite web server on port 5173
- the lite server starts loading index.html from the application
- index.html loads main.jsx file
- main.jsx calls createRoot() to create a root container to load react components
- and starts rendering first component named App
- App component start loading the user interface

component

- reusable entity which contains logic (in JS) and UI (in JSX)
- a component could as small as a part of an application
- or as big as an entire page
- types
    - class component
        - component created using a class
        - earlier (before react 16), class components were used for creating statefull component (component with state)
        - but after react 16 (in which react hooks were introduced), class components are not needed anymore
        - class components are having some overhead members compared to functional components
    - functional component
        - component created using a function
        - a javascript function which returns a JSX user interface
        - earlier (before react 16), functional components were used to create stateless components (components without state)
        - but with react 16 (react hooks), it is possible to store the state in a functional component
        - functiona components are preferred over class component
        - since the functional components do not have any overhead members, it is a way to create component compared to class component
- conventions
    - always start the component name with upper case
    - name of file should be same as the component name
    - if a component is reusable (like Person or Car), keep it in a directory named components
    - if a component is representing a page or screen, keep it in a directory named pages or screens
    - always use the props destructuring while defining the component
    - always keep one public component in a file

props

- is an object which is collection of all the properties passed to a component
- is the only way for a parent component to pass the data/properties to the child component
- props is a readonly object: the child component must not change the values sent by the parent component

```
function Person1(props) {
  return (
    <div>
      <div>name = {props['name']}</div>
      <div>address = {props['address']}</div>
    </div>
  )
}

function Person2(props) {
```

```
  const { name, address } = props
  return (
    <div>
      <div>name = {name}</div>
      <div>address = {address}</div>
    </div>
  )
}

function Person3({ name, address }) {
  return (
    <div>
      <div>name = {name}</div>
      <div>address = {address}</div>
    </div>
  )
}

function App() {
  return (
    <div>
      <Person1
        name='person1'
        address='pune'
      />
      <Person2
        name='person2'
        address='karad'
      />
      <Person3
        name='person3'
        address='satara'
      />
    </div>
  )
}
```

state

- object (collection of property-value pairs) maintained by compopent to trigger component re-render action
- if there is a change in the component state, the component will re-render itself
- unlike props, state object is readwritable
- every component will maintain its own state
- to add a state member inside a functional component use a react hook name useState()

state vs props

- state is read writable while props is read only
- when state changes, the component re-renders itself while, when props changes, component does not render itself

- state is maintained by individual component while, props will be sent by parent component to child component
- useState() hook is required to create state inside functional component while, no hook is required to send props to child component

react hook

- special function whihch starts with `use`
- types
    - built-in hooks
        - useState()
        - useEffect()
        - useReducer()
        - useMemo()
        - useId()
        - useRef()
        - useCallback()
        - useNavigate()
        - useLocation()
        - useParams()
        - useSelector()
        - useDispatch()
    - custom hooks
        - user defined hooks

useState()

- hook used to add a member inside a component's state
- accepts a parameter which is the initial value of the member
- returns an array with 2 values
    - 0th position: reference to the member (used to read the value from state)
    - 1st position: reference to the function to update the value in the state object

```
function Counter() {
  // create a state to store counter value
  const [couter, setCounter] = useState(0)
  return <div>counter: {counter}</div>
}
```

JSX

- a syntax extension for JavaScript
- allows you to write HTML code inside JavaScript
- how does it work?
    - babel is used to convert JSX code into JavaScript code

- babel is a JavaScript compiler

```
<script type="text/babel">
  // JSX code
  const h2 = <h2>hello world</h2>

  // babel converts the above code into the following code
  // const h2 = React.createElement('h2', {}, 'hello world')
</script>
```

VS extensions

- auto import:
  - https://marketplace.visualstudio.com/items/?itemName=NuclleaR.vscode-extension-auto-import
- auto tag renamer:
  - https://marketplace.visualstudio.com/items/?itemName=formulahendry.auto-rename-tag
- code snippets for react:
  - https://marketplace.visualstudio.com/items/?itemName=rodrigovallades.es7-react-js-snippets

React

Single Page Application (SPA)

- a application that loads a single HTML page and dynamically updates that page as the user interacts with the app
- to develop SPAs,
    - we need to use a JavaScript framework or library
    - like
        - React
        - Angular
        - VueJs
- advantages
    - fast: similar performance to native apps
    - responsive: the app responds to user interactions (browser size changes),
        - to make the app responsive
            - we need to use CSS media queries
            - frameworks: bootstrap, tailwind
    - user-friendly

functional programming language

- function is considered as first class citizen
    - function is created as a variable of type function
- function can be passed as an argument to another function
- function can be returned from another function as return value
- map()
    - used to iterate over a collection to transform the values to new ones
    - accepts a function as a parameter which gets called every time for every value
    - the parameter function must return a transformed value for original value
    - all the transformed values will be returned a collection as a return value of map function
    - the size of returned collection is always same as original collection

```
// array of numbers
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

// get squqre of each number
const squares = numbers.map((number) => number ** 2)
```

function reference

- a reference to a function
- a variable that holds a function body's address

```
// here the function1 is a function reference
// to the function body
function function1() {
  console.log('inside function1')
}
```

export and import

- export
    - used to export any entity from a file for others to import
    - a file can export multiple entities for others

    ```
    // App.jsx
    export function App() {
      ...
    }

    // main.jsx

    // importing with same name as that of the exported entity
    import {App} from './App.jsx'

    // importing with an alias
    import {App as MyApp} from './App.jsx'
    ```

- export default
    - by default only one entity (class, function, variable, constant) can be exported from a file with default keyword

    ```
    // App.jsx
    function App() {
      ...
    }
    export default App

    // main.jsx

    // importing with same name as that of exported entity
    import App from './App.jsx'

    // importing wth an alias
    import MyApp from './App.jsx'
    ```

web storage

- web storage object is used to store collection key-value pairs
    - where key and value must be scalar value (string)
- the data will be persisted on client side (inside the browser)
- web storage is a browser specific object
- types
    - sessionStorage
        - used to store the data till the browser tab is open
        - when the tab is closed, the sessionStorage losses all the data
        - acts a temporory storage

```
// add or set a key-value in session storage
sessionStorage['username'] = 'user1'
sessionStorage.setItem('username', 'user1')

// get the value from session storage using its key
console.log(`username: ${sessionStorage['username']}`)
console.log(`username: ${sessionStorage.getItem('username')}`)

// remove the key from session storage
sessionStorage.removeItem('username')
```

    - localStorage
        - used to store the data in the form of key-value pairs
        - both the key and value must be scalar values (string)
        - the local storage will persist the data permananently (till user explicitly removes it)

```
// add or set a key-value in local storage
localStorage['username'] = 'user1'
localStorage.setItem('username', 'user1')

// get the value from local storage using its key
console.log(`username: ${localStorage['username']}`)
console.log(`username: ${localStorage.getItem('username')}`)

// remove the key from local storage
localStorage.removeItem('username')
```

React

- a JavaScript library for building user interfaces

React vs Angular

- React is a Library (developed in JS) and Angular is a Framework (developed in TypeScript)
- react development and performance is faster than angular
    - to make it faster, react uses a virtual DOM
    - it also has less memory consumption/footprint
- React has less learning curve than Angular
- React does not have any architecture whereas, Angular has a predefined architecture and tooling

important points

- do not use `class` as it is a reserved keyword in JavaScript, use `className` instead
- interpolation is done using `{}` in JSX
- interpolation always requires a scalar value and CAN NOT render an object

virtual DOM

- a lightweight copy of the real DOM (browser DOM) (document object)
- react uses virtual DOM to improve performance
- when we update the state of a component, react creates a new virtual DOM and compares it with the previous virtual DOM
- then it updates only the changed parts of the real DOM
- this process is called reconciliation

environment setup

- using CDN links
    - CDN: content delivery network
    - add react using CDN links

```
<html>
  <head>
    <!-- used for react development -->
    <script
      crossorigin
      src="https://unpkg.com/react@18/umd/react.development.js"
    ></script>

    <!-- used for react virtual dom development -->
    <script
      crossorigin
      src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"
    ></script>
  </head>

  <body>
```

```
    <div id="root"></div>
  </body>
</html>
```

- to create element use a function called `createElement`

```
// create element
// React is an object which will be used to create elements
// this object is provided by react library
(react.development.js)

// parameters
// 1st: name or type of the element (e.g. h1, h2 etc)
// 2nd: attributes or properties of the element (e.g. class, id,
style etc). this must be an object.
// 3rd: contents of the element (e.g. text, html etc)
const h2 = React.createElement('h2', {}, 'hello world')

// React 17 style of rendering an element
// get the root element
// this is the element where we will render our react elements
// const root = document.getElementById('root')

// render the element
// ReactDOM.render(h2, root)

// React 18 style of rendering an element

// create a root element
const root = ReactDOM.createRoot(document.getElementById('root'))

// render the element
root.render(h2)
```

- to create an element using JSX, use babel

```
<html>
  <head>
    <!-- used for react development -->
    <script
      crossorigin
      src="https://unpkg.com/react@18/umd/react.development.js"
    ></script>

    <!-- used for react virtual dom development -->
    <script
      crossorigin
      src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"
```

```
    ></script>

    <!-- babel compiler -->
    <script
src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>

  <body>
    <div id="root"></div>
    <script type="text/babel">
      const h2 = <h2>hello world</h2>
      const root =
ReactDOM.createRoot(document.getElementById('root'))
      root.render(h2)
    </script>
  </body>
</html>
```

- using package manager like vite

```
# install yarn on windows
> npm install -g yarn

# install yarn on linux or mac
> sudo npm install -g yarn

# create react application using vite
> npm create vite@latest <application name>
> yarn create vite <application name>

# go to the project directory
> cd <application name>

# install the dependencies
> npm install
> yarn

# start the application
> npm run dev
> yarn dev
```

  - project structure
    - node_modules
      - contains all the modules (dependencies) which are required to develop or run the application
      - will be downloaded every time when npm install or yarn install command is used
      - never commit this directory in your git repository

- public
  - directory which contains public files
  - e.g. images, audio or video which are used in the application
- src
  - directory which contains all the components of the application
  - contains
    - assets
      - directory which contains the assets (images, audio or video files)
    - pages
      - directory which contains the application pages (screens)
    - components
      - directory which contains reusable components
    - services
      - directory which contains the services
      - a service file would contain the code to call the REST APIs
    - main.jsx
      - contains the code to start react subsystem
      - contains the function createRoot(..).render()
    - index.css
      - contains global css rules which can be shared across the components in the application
    - App.jsx
      - contains the default component called as App
      - this is the startup component of every application
    - App.css
      - contains css rules need to applied on the App component
- .gitignore
  - file which contains the rules of the files which needed to be committed in the git repository
- eslint.config.js
  - contains configuration for eslint
  - lint is a program used to check the syntax of a selected language
- index.html
  - only html file in the project which starts the application
- package.json
  - contain the node configuration like name, dependencies or devDependencies etc.
- vite.config.js
  - contains the vite configuration
- yarn.lock
  - contains latest versions of the dependencies installed in the node_modules directory

react application startup

- vite will start a lite web server on port 5173
- the lite server starts loading index.html from the application
- index.html loads main.jsx file

- main.jsx calls createRoot() to create a root container to load react components
- and starts rendering first component named App
- App component start loading the user interface

component

- reusable entity which contains logic (in JS) and UI (in JSX)
- a component could as small as a part of an application
- or as big as an entire page
- types
    - class component
        - component created using a class
        - earlier (before react 16), class components were used for creating statefull component (component with state)
        - but after react 16 (in which react hooks were introduced), class components are not needed anymore
        - class components are having some overhead members compared to functional components
    - functional component
        - component created using a function
        - a javascript function which returns a JSX user interface
        - earlier (before react 16), functional components were used to create stateless components (components without state)
        - but with react 16 (react hooks), it is possible to store the state in a functional component
        - functiona components are preferred over class component
        - since the functional components do not have any overhead members, it is a way to create component compared to class component
- conventions
    - always start the component name with upper case
    - name of file should be same as the component name
    - if a component is reusable (like Person or Car), keep it in a directory named components
    - if a component is representing a page or screen, keep it in a directory named pages or screens
    - always use the props destructuring while defining the component
    - always keep one public component in a file

component life cycle

- life cycle of component
- stages the component will pass from its creation to its death
- it provides the functionality for handling the life cycle event

props

- is an object which is collection of all the properties passed to a component
- is the only way for a parent component to pass the data/properties to the child component
- props is a readonly object: the child component must not change the values sent by the parent component

```
function Person1(props) {
  return (
    <div>
      <div>name = {props['name']}</div>
      <div>address = {props['address']}</div>
    </div>
  )
}

function Person2(props) {
  const { name, address } = props
  return (
    <div>
      <div>name = {name}</div>
      <div>address = {address}</div>
    </div>
  )
}

function Person3({ name, address }) {
  return (
    <div>
      <div>name = {name}</div>
      <div>address = {address}</div>
    </div>
  )
}

function App() {
  return (
    <div>
      <Person1
        name='person1'
        address='pune'
      />
      <Person2
        name='person2'
        address='karad'
      />
      <Person3
        name='person3'
        address='satara'
      />
    </div>
  )
}
```

state

- object (collection of property-value pairs) maintained by compopent to trigger component re-render action

- if there is a change in the component state, the component will re-render itself
- unlike props, state object is readwritable
- every component will maintain its own state
- to add a state member inside a functional component use a react hook name useState()

state vs props

- state is read writable while props is read only
- when state changes, the component re-renders itself while, when props changes, component does not render itself
- state is maintained by individual component while, props will be sent by parent component to child component
- useState() hook is required to create state inside functional component while, no hook is required to send props to child component

react hook

- special function whihch starts with `use`
- types
    - built-in hooks
        - useState()
        - useEffect()
        - useReducer()
        - useMemo()
        - useId()
        - useRef()
        - useCallback()
        - useNavigate()
        - useLocation()
        - useParams()
        - useSelector()
        - useDispatch()
    - custom hooks
        - user defined hooks

useState()

- hook used to add a member inside a component's state
- accepts a parameter which is the initial value of the member
- returns an array with 2 values
    - 0th position: reference to the member (used to read the value from state)
    - 1st position: reference to the function to update the value in the state object

```
function Counter() {
  // create a state to store counter value
  const [couter, setCounter] = useState(0)
  return <div>counter: {counter}</div>
}
```

useNavigate()

- react hook added by react-router-dom
- used to get the naigate() function reference
- navigate() function is used to perform dynamic navigation
- navigating from one to another component

```
import { useNavigate } from 'react-router-dom'

function Login() {
  // get navigate() function reference
  const navigate = useNavigate()

  const onLogin = () => {
    // check if user is successfully logged in
    navigate('/home')
  }

  return (
    <>
      <h1>Login</h1>
      <button onClick={onLogin}>login</button>
    </>
  )
}
```

useEffect()

- used to handle life cycle events of a component
- accepts 2 parameters
    - 1st parameter: callback function
    - 2ns parameter: dependency array
- event1: component is loaded (componentDidMount)
    - dependency array must be empty
    - the callback function gets called when the component gets mounted
    - it is similar to the constructor method in any class
    - it gets called only function

```
function Home() {
  // this code here will handle the component did mount event
  useEffect(() => {
    // this function gets called immediately after component is
mounted
```

```
    console.log('Home component is mounted')
  }, [])

  return (
    <div>
      <h1>Home</h1>
    </div>
  )
}
```

- event2: component is unloaded (componentDidUnmount)
  - it gets called when the component gets removed from the screen
  - it gets called only once in its life cycle
  - dependency array must be empty
  - similar to destructor of any class

```
function Home() {
  // this code here will handle the component did mount event
  useEffect(() => {
    return () => {
      // this function gets called just before the component is
unloading from screen
      console.log('component is getting unloaded')
    }
  }, [])

  return (
    <div>
      <h1>Home</h1>
    </div>
  )
}
```

- event3: component state is changed
  - called as soon as the state of a component changes
  - this event gets fired irrespective of any state member
  - must NOT pass the dependency array

```
function Home() {
  const [n1, setN1] = useState(10)
  const [n2, setN2] = useState(20)

  const onUpdateN1 = () => {
    setN1(n1 + 1)
  }

  const onUpdateN2 = () => {
```

```
    setN2(n2 + 1)
  }

  useEffect(() => {
    // this function here will get called everytime when
    // the state changes
    console.log(`state changed..`)
  })

  return (
    <div>
      <Navbar />
      <div className='container'>
        <h1 className='page-header'>Dummy</h1>

        <div>n1: {n1}</div>
        <div>
          <button
            onClick={onUpdateN1}
            className='btn btn-success'
          >
            Update N1
          </button>
        </div>
        <div>n2: {n2}</div>
        <div>
          <button
            onClick={onUpdateN2}
            className='btn btn-success'
          >
            Update N2
          </button>
        </div>
      </div>
    </div>
  )
}
```

- event4: component state is changed
    - called as soon as the state of a component changes because of a required dependency

```
function Home() {
  const [n1, setN1] = useState(10)
  const [n2, setN2] = useState(20)

  const onUpdateN1 = () => {
    setN1(n1 + 1)
  }

  const onUpdateN2 = () => {
    setN2(n2 + 1)
```

```
  }

  useEffect(() => {
    // this function gets called when n1 changes
    console.log(`n1 changed..: ${n1}`)
  }, [n1])

  useEffect(() => {
    // this function gets called when n1 changes
    console.log(`n2 changed..: ${n2}`)
  }, [n2])

  return (
    <div>
      <Navbar />
      <div className='container'>
        <h1 className='page-header'>Dummy</h1>

        <div>n1: {n1}</div>
        <div>
          <button
            onClick={onUpdateN1}
            className='btn btn-success'
          >
            Update N1
          </button>
        </div>
        <div>n2: {n2}</div>
        <div>
          <button
            onClick={onUpdateN2}
            className='btn btn-success'
          >
            Update N2
          </button>
        </div>
      </div>
    </div>
  )
}
```

JSX

- a syntax extension for JavaScript
- allows you to write HTML code inside JavaScript
- how does it work?
    - babel is used to convert JSX code into JavaScript code
    - babel is a JavaScript compiler

```
```

```
<script type="text/babel">
  // JSX code
  const h2 = <h2>hello world</h2>

  // babel converts the above code into the following code
  // const h2 = React.createElement('h2', {}, 'hello world')
</script>
```

VS extensions

- auto import:
  - https://marketplace.visualstudio.com/items/?itemName=NucllеaR.vscode-extension-auto-import
- auto tag renamer:
  - https://marketplace.visualstudio.com/items/?itemName=formulahendry.auto-rename-tag
- code snippets for react:
  - https://marketplace.visualstudio.com/items/?itemName=rodrigovallades.es7-react-js-snippets

external libraries

```
# install any package
> npm install <package name>
> yarn add <package name>
```

- react toastify
  - used to show the toast message
  - https://www.npmjs.com/package/react-toastify
  - yarn add react-toastify

```
import { ToastContainer } from 'react-toastify'

function App() {
  return (
    <div>
      <ToastContainer />
    </div>
  )
}

// to show the toast messages
// import {toast} from 'react-toastify'

// toast.warn('this is warning message')
// toast.error('this is error message')
```

```
// toast.info('this is info message')
// toast.success('this is success message')
```

- axios
  - used to make API calls
  - https://www.npmjs.com/package/axios
  - yarn add axios

```
import axios from 'axios'

async function makePostCall(email, password) {
  try {
    const url = 'http://localhost:4000/user/login'
    const body = { email, password }
    const response = await axios.post(url, body)
    console.log(response.data)
  } catch (ex) {
    console.log('exception: ', ex)
  }
}

async function makePostCallWithToken(title, description, price) {
  try {
    const url = 'http://localhost:4000/property/'
    const body = { title, description, price }
    const token = sessionStorage.getItem('token')
    const response = await axios.post(url, body, {
      headers: { token },
    })
    console.log(response.data)
  } catch (ex) {
    console.log('exception: ', ex)
  }
}

async function makeGetCall() {
  try {
    const url = 'http://localhost:4000/property'
    const response = await axios.get(url)
    console.log(response.data)
  } catch (ex) {
    console.log('exception: ', ex)
  }
}

async function makeGetCallWithToken() {
  try {
    const url = 'http://localhost:4000/my'
    const token = sessionStorage.getItem('token')
    const response = await axios.get(url, {
```

```
      headers: { token },
    })
    console.log(response.data)
  } catch (ex) {
    console.log('exception: ', ex)
  }
}
```

- react-router
  - used to add routing feature in react application
  - routing is used to provide navigation from one to another component
  - https://reactrouter.com/
  - yarn add react-router-dom
  - route in express is mapping of
    - http method (get, post, put, delete, patch)
    - url path
    - callback function (handler)
  - route in react is mapping of
    - url path
    - component
  - BrowserRouter
    - router provided by library to implement the routing
    - to load the required comopnent by inspecting the url path

```
// step1: wrap <App /> inside BrowserRouter
// main.jsx
import { BrowserRouter } from 'react-router-dom'

// Wrap the App component inside the BrowserRouter
createRoot(document.getElementById('root')).render(
  <BrowserRouter>
    <App />
  </BrowserRouter>
)
```

```
// step2: define all the routes
// App.jsx
import { Routes, Route } from 'react-router-dom'
import Login from './pages/Login'

function App() {

  return <>
    <Routes>
      <Route path="login" element={<Login />}>
    </Routes>
```

```
    </>
  }
```

- static navigation
  - navigation from one component to another without using JS code (logic)
  - navigation will be implemented by using JSX
  - use static navigation when the destination (component) needs to open without having any condition (criteria)

```
import { Link } from 'react-router-dom'

function Login() {
  return (
    <>
      <h2>Login</h2>
      <Link to='/register'>Register here</Link>
    </>
  )
}
```

- dynamic navigation
  - navigation performed using JS code
  - used dynamic navigation when the destination (componen) needs to open by validation some condition (criteria)

```
import { useNavigate } from 'react-router-dom'

function Login() {
  // get navigate() function reference
  const navigate = useNavigate()

  const onLogin = () => {
    // check if user is successfully logged in
    navigate('/home')
  }

  return (
    <>
      <h1>Login</h1>
      <button onClick={onLogin}>login</button>
    </>
  )
}
```

- tanstack router

- use to add routing feature in react application
- https://tanstack.com/router/latest

React

Single Page Application (SPA)

- a application that loads a single HTML page and dynamically updates that page as the user interacts with the app
- to develop SPAs,
    - we need to use a JavaScript framework or library
    - like
        - React
        - Angular
        - VueJs
- advantages
    - fast: similar performance to native apps
    - responsive: the app responds to user interactions (browser size changes),
        - to make the app responsive
            - we need to use CSS media queries
            - frameworks: bootstrap, tailwind
    - user-friendly

functional programming language

- function is considered as first class citizen
    - function is created as a variable of type function
- function can be passed as an argument to another function
- function can be returned from another function as return value
- map()
    - used to iterate over a collection to transform the values to new ones
    - accepts a function as a parameter which gets called every time for every value
    - the parameter function must return a transformed value for original value
    - all the transformed values will be returned a collection as a return value of map function
    - the size of returned collection is always same as original collection

```
// array of numbers
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

// get squqre of each number
const squares = numbers.map((number) => number ** 2)
```

function reference

- a reference to a function
- a variable that holds a function body's address

```
// here the function1 is a function reference
// to the function body
function function1() {
  console.log('inside function1')
}
```

export and import

- export
    - used to export any entity from a file for others to import
    - a file can export multiple entities for others

    ```
    // App.jsx
    export function App() {
      ...
    }

    // main.jsx

    // importing with same name as that of the exported entity
    import {App} from './App.jsx'

    // importing with an alias
    import {App as MyApp} from './App.jsx'
    ```

- export default
    - by default only one entity (class, function, variable, constant) can be exported from a file with default keyword

    ```
    // App.jsx
    function App() {
      ...
    }
    export default App

    // main.jsx

    // importing with same name as that of exported entity
    import App from './App.jsx'

    // importing wth an alias
    import MyApp from './App.jsx'
    ```

web storage

- web storage object is used to store collection key-value pairs
    - where key and value must be scalar value (string)
- the data will be persisted on client side (inside the browser)
- web storage is a browser specific object
- types
    - sessionStorage
        - used to store the data till the browser tab is open
        - when the tab is closed, the sessionStorage losses all the data
        - acts a temporory storage

```
// add or set a key-value in session storage
sessionStorage['username'] = 'user1'
sessionStorage.setItem('username', 'user1')

// get the value from session storage using its key
console.log(`username: ${sessionStorage['username']}`)
console.log(`username: ${sessionStorage.getItem('username')}`)

// remove the key from session storage
sessionStorage.removeItem('username')
```

    - localStorage
        - used to store the data in the form of key-value pairs
        - both the key and value must be scalar values (string)
        - the local storage will persist the data permananently (till user explicitly removes it)

```
// add or set a key-value in local storage
localStorage['username'] = 'user1'
localStorage.setItem('username', 'user1')

// get the value from local storage using its key
console.log(`username: ${localStorage['username']}`)
console.log(`username: ${localStorage.getItem('username')}`)

// remove the key from local storage
localStorage.removeItem('username')
```

React

- a JavaScript library for building user interfaces

React vs Angular

- React is a Library (developed in JS) and Angular is a Framework (developed in TypeScript)
- react development and performance is faster than angular
    - to make it faster, react uses a virtual DOM
    - it also has less memory consumption/footprint
- React has less learning curve than Angular
- React does not have any architecture whereas, Angular has a predefined architecture and tooling

important points

- do not use `class` as it is a reserved keyword in JavaScript, use `className` instead
- interpolation is done using `{}` in JSX
- interpolation always requires a scalar value and CAN NOT render an object

virtual DOM

- a lightweight copy of the real DOM (browser DOM) (document object)
- react uses virtual DOM to improve performance
- when we update the state of a component, react creates a new virtual DOM and compares it with the previous virtual DOM
- then it updates only the changed parts of the real DOM
- this process is called reconciliation

environment setup

- using CDN links
    - CDN: content delivery network
    - add react using CDN links

```
<html>
  <head>
    <!-- used for react development -->
    <script
      crossorigin
      src="https://unpkg.com/react@18/umd/react.development.js"
    ></script>

    <!-- used for react virtual dom development -->
    <script
      crossorigin
      src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"
    ></script>
  </head>

  <body>
```

```
    <div id="root"></div>
  </body>
</html>
```

- to create element use a function called `createElement`

```
// create element
// React is an object which will be used to create elements
// this object is provided by react library
(react.development.js)

// parameters
// 1st: name or type of the element (e.g. h1, h2 etc)
// 2nd: attributes or properties of the element (e.g. class, id,
style etc). this must be an object.
// 3rd: contents of the element (e.g. text, html etc)
const h2 = React.createElement('h2', {}, 'hello world')

// React 17 style of rendering an element
// get the root element
// this is the element where we will render our react elements
// const root = document.getElementById('root')

// render the element
// ReactDOM.render(h2, root)

// React 18 style of rendering an element

// create a root element
const root = ReactDOM.createRoot(document.getElementById('root'))

// render the element
root.render(h2)
```

- to create an element using JSX, use babel

```
<html>
  <head>
    <!-- used for react development -->
    <script
      crossorigin
      src="https://unpkg.com/react@18/umd/react.development.js"
    ></script>

    <!-- used for react virtual dom development -->
    <script
      crossorigin
      src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"
```

```
    ></script>

    <!-- babel compiler -->
    <script
src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>

  <body>
    <div id="root"></div>
    <script type="text/babel">
      const h2 = <h2>hello world</h2>
      const root =
ReactDOM.createRoot(document.getElementById('root'))
      root.render(h2)
    </script>
  </body>
</html>
```

- using package manager like vite

```
# install yarn on windows
> npm install -g yarn

# install yarn on linux or mac
> sudo npm install -g yarn

# create react application using vite
> npm create vite@latest <application name>
> yarn create vite <application name>

# go to the project directory
> cd <application name>

# install the dependencies
> npm install
> yarn

# start the application
> npm run dev
> yarn dev
```

  - project structure
    - node_modules
      - contains all the modules (dependencies) which are required to develop or run the application
      - will be downloaded every time when npm install or yarn install command is used
      - never commit this directory in your git repository

- public
  - directory which contains public files
  - e.g. images, audio or video which are used in the application
- src
  - directory which contains all the components of the application
  - contains
    - assets
      - directory which contains the assets (images, audio or video files)
    - pages
      - directory which contains the application pages (screens)
    - components
      - directory which contains reusable components
    - services
      - directory which contains the services
      - a service file would contain the code to call the REST APIs
    - main.jsx
      - contains the code to start react subsystem
      - contains the function createRoot(..).render()
    - index.css
      - contains global css rules which can be shared across the components in the application
    - App.jsx
      - contains the default component called as App
      - this is the startup component of every application
    - App.css
      - contains css rules need to applied on the App component
- .gitignore
  - file which contains the rules of the files which needed to be committed in the git repository
- eslint.config.js
  - contains configuration for eslint
  - lint is a program used to check the syntax of a selected language
- index.html
  - only html file in the project which starts the application
- package.json
  - contain the node configuration like name, dependencies or devDependencies etc.
- vite.config.js
  - contains the vite configuration
- yarn.lock
  - contains latest versions of the dependencies installed in the node_modules directory

react application startup

- vite will start a lite web server on port 5173
- the lite server starts loading index.html from the application
- index.html loads main.jsx file

- main.jsx calls createRoot() to create a root container to load react components
- and starts rendering first component named App
- App component start loading the user interface

component

- reusable entity which contains logic (in JS) and UI (in JSX)
- a component could as small as a part of an application
- or as big as an entire page
- types
    - class component
        - component created using a class
        - earlier (before react 16), class components were used for creating statefull component (component with state)
        - but after react 16 (in which react hooks were introduced), class components are not needed anymore
        - class components are having some overhead members compared to functional components
    - functional component
        - component created using a function
        - a javascript function which returns a JSX user interface
        - earlier (before react 16), functional components were used to create stateless components (components without state)
        - but with react 16 (react hooks), it is possible to store the state in a functional component
        - functiona components are preferred over class component
        - since the functional components do not have any overhead members, it is a way to create component compared to class component
- conventions
    - always start the component name with upper case
    - name of file should be same as the component name
    - if a component is reusable (like Person or Car), keep it in a directory named components
    - if a component is representing a page or screen, keep it in a directory named pages or screens
    - always use the props destructuring while defining the component
    - always keep one public component in a file

component life cycle

- life cycle of component
- stages the component will pass from its creation to its death
- it provides the functionality for handling the life cycle event

props

- is an object which is collection of all the properties passed to a component
- is the only way for a parent component to pass the data/properties to the child component
- props is a readonly object: the child component must not change the values sent by the parent component

```
function Person1(props) {
  return (
    <div>
      <div>name = {props['name']}</div>
      <div>address = {props['address']}</div>
    </div>
  )
}

function Person2(props) {
  const { name, address } = props
  return (
    <div>
      <div>name = {name}</div>
      <div>address = {address}</div>
    </div>
  )
}

function Person3({ name, address }) {
  return (
    <div>
      <div>name = {name}</div>
      <div>address = {address}</div>
    </div>
  )
}

function App() {
  return (
    <div>
      <Person1
        name='person1'
        address='pune'
      />
      <Person2
        name='person2'
        address='karad'
      />
      <Person3
        name='person3'
        address='satara'
      />
    </div>
  )
}
```

state

- object (collection of property-value pairs) maintained by compopent to trigger component re-render action

- if there is a change in the component state, the component will re-render itself
- unlike props, state object is readwritable
- every component will maintain its own state
- to add a state member inside a functional component use a react hook name useState()

state vs props

- state is read writable while props is read only
- when state changes, the component re-renders itself while, when props changes, component does not render itself
- state is maintained by individual component while, props will be sent by parent component to child component
- useState() hook is required to create state inside functional component while, no hook is required to send props to child component

react hook

- special function whihch starts with `use`
- types
    - built-in hooks
        - useState()
        - useEffect()
        - useReducer()
        - useMemo()
        - useContext()
        - useId()
        - useRef()
        - useCallback()
        - useNavigate()
        - useLocation()
        - useParams()
        - useSelector()
        - useDispatch()
    - custom hooks
        - user defined hooks

useState()

- hook used to add a member inside a component's state
- accepts a parameter which is the initial value of the member
- returns an array with 2 values
    - 0th position: reference to the member (used to read the value from state)
    - 1st position: reference to the function to update the value in the state object

```
function Counter() {
  // create a state to store counter value
  const [couter, setCounter] = useState(0)
```

```
  return <div>counter: {counter}</div>
}
```

useNavigate()

- react hook added by react-router-dom
- used to get the naigate() function reference
- navigate() function is used to perform dynamic navigation
- navigating from one to another component

```
import { useNavigate } from 'react-router-dom'

function Login() {
  // get navigate() function reference
  const navigate = useNavigate()

  const onLogin = () => {
    // check if user is successfully logged in
    navigate('/home')
  }

  return (
    <>
      <h1>Login</h1>
      <button onClick={onLogin}>login</button>
    </>
  )
}
```

useEffect()

- used to handle life cycle events of a component
- accepts 2 parameters
    - 1st parameter: callback function
    - 2ns parameter: dependency array
- event1: component is loaded (componentDidMount)
    - dependency array must be empty
    - the callback function gets called when the component gets mounted
    - it is similar to the constructor method in any class
    - it gets called only function

```
function Home() {
  // this code here will handle the component did mount event
  useEffect(() => {
```

```
    // this function gets called immediately after component is
mounted
    console.log('Home component is mounted')
  }, [])

  return (
    <div>
      <h1>Home</h1>
    </div>
  )
}
```

- event2: component is unloaded (componentDidUnmount)
  - it gets called when the component gets removed from the screen
  - it gets called only once in its life cycle
  - dependency array must be empty
  - similar to destructor of any class

```
function Home() {
  // this code here will handle the component did mount event
  useEffect(() => {
    return () => {
      // this function gets called just before the component is
unloading from screen
      console.log('component is getting unloaded')
    }
  }, [])

  return (
    <div>
      <h1>Home</h1>
    </div>
  )
}
```

- event3: component state is changed
  - called as soon as the state of a component changes
  - this event gets fired irrespective of any state member
  - must NOT pass the dependency array

```
function Home() {
  const [n1, setN1] = useState(10)
  const [n2, setN2] = useState(20)

  const onUpdateN1 = () => {
    setN1(n1 + 1)
  }
```

```
  const onUpdateN2 = () => {
    setN2(n2 + 1)
  }

  useEffect(() => {
    // this function here will get called everytime when
    // the state changes
    console.log(`state changed..`)
  })

  return (
    <div>
      <Navbar />
      <div className='container'>
        <h1 className='page-header'>Dummy</h1>

        <div>n1: {n1}</div>
        <div>
          <button
            onClick={onUpdateN1}
            className='btn btn-success'
          >
            Update N1
          </button>
        </div>
        <div>n2: {n2}</div>
        <div>
          <button
            onClick={onUpdateN2}
            className='btn btn-success'
          >
            Update N2
          </button>
        </div>
      </div>
    </div>
  )
}
```

- event4: component state is changed
    - called as soon as the state of a component changes because of a required dependency

```
function Home() {
  const [n1, setN1] = useState(10)
  const [n2, setN2] = useState(20)

  const onUpdateN1 = () => {
    setN1(n1 + 1)
  }
```

```
  const onUpdateN2 = () => {
    setN2(n2 + 1)
  }

  useEffect(() => {
    // this function gets called when n1 changes
    console.log(`n1 changed..: ${n1}`)
  }, [n1])

  useEffect(() => {
    // this function gets called when n1 changes
    console.log(`n2 changed..: ${n2}`)
  }, [n2])

  return (
    <div>
      <Navbar />
      <div className='container'>
        <h1 className='page-header'>Dummy</h1>

        <div>n1: {n1}</div>
        <div>
          <button
            onClick={onUpdateN1}
            className='btn btn-success'
          >
            Update N1
          </button>
        </div>
        <div>n2: {n2}</div>
        <div>
          <button
            onClick={onUpdateN2}
            className='btn btn-success'
          >
            Update N2
          </button>
        </div>
      </div>
    </div>
  )
}
```

JSX

- a syntax extension for JavaScript
- allows you to write HTML code inside JavaScript
- how does it work?
    - babel is used to convert JSX code into JavaScript code

- babel is a JavaScript compiler

```
<script type="text/babel">
  // JSX code
  const h2 = <h2>hello world</h2>

  // babel converts the above code into the following code
  // const h2 = React.createElement('h2', {}, 'hello world')
</script>
```

context api

- it is a built-in feature of react used to share data among multple components
- context
  - instance of Context component used to provide (share) data
  - to create a context use createContent() function
- to share the context with components use following syntax
  - ...

```
import { createContext, useState } from 'react'

// create a context to share the data
export const CounterContext = createContext()

function App() {
  const [counter, setCounter] = useState(0)
  return (
    <div>
      <h1>App Component</h1>
      <CounterContext value={{ couter, setCounter }}>
        <Counter1 />
        <Counter2 />
      </CounterContext>
    </div>
  )
}
```

- to use the contxt in the child components, use useContext() react hook

```
import { CounterContext } from '../App'

function Counter1() {
  // get the counter and setCounter from counter context created in
App.jsx
  const { counter, setCounter } = useContext(CounterContext)

  const onIncrement = () => setCounter(counter + 1)
```

```
  return (
    <div>
      <div>counter: {counter}</div>
      <div>
        <button onClick={onIncrement}>increment</button>{' '}
      </div>
    </div>
  )
}
```

- use case: to share simple values like
    - login status
    - themse used in the application

VS extensions

- auto import:
    - https://marketplace.visualstudio.com/items/?itemName=NuclleaR.vscode-extension-auto-import
- auto tag renamer:
    - https://marketplace.visualstudio.com/items/?itemName=formulahendry.auto-rename-tag
- code snippets for react:
    - https://marketplace.visualstudio.com/items/?itemName=rodrigovallades.es7-react-js-snippets

external libraries

```
# install any package
> npm install <package name>
> yarn add <package name>
```

- react toastify
    - used to show the toast message
    - https://www.npmjs.com/package/react-toastify
    - yarn add react-toastify

```
import { ToastContainer } from 'react-toastify'

function App() {
  return (
    <div>
      <ToastContainer />
    </div>
  )
```

```
}

// to show the toast messages
// import {toast} from 'react-toastify'

// toast.warn('this is warning message')
// toast.error('this is error message')
// toast.info('this is info message')
// toast.success('this is success message')
```

- axios
  - used to make API calls
  - https://www.npmjs.com/package/axios
  - yarn add axios

```
import axios from 'axios'

async function makePostCall(email, password) {
  try {
    const url = 'http://localhost:4000/user/login'
    const body = { email, password }
    const response = await axios.post(url, body)
    console.log(response.data)
  } catch (ex) {
    console.log('exception: ', ex)
  }
}

async function makePostCallWithToken(title, description, price) {
  try {
    const url = 'http://localhost:4000/property/'
    const body = { title, description, price }
    const token = sessionStorage.getItem('token')
    const response = await axios.post(url, body, {
      headers: { token },
    })
    console.log(response.data)
  } catch (ex) {
    console.log('exception: ', ex)
  }
}

async function makeGetCall() {
  try {
    const url = 'http://localhost:4000/property'
    const response = await axios.get(url)
    console.log(response.data)
  } catch (ex) {
    console.log('exception: ', ex)
  }
```

```
}

async function makeGetCallWithToken() {
  try {
    const url = 'http://localhost:4000/my'
    const token = sessionStorage.getItem('token')
    const response = await axios.get(url, {
      headers: { token },
    })
    console.log(response.data)
  } catch (ex) {
    console.log('exception: ', ex)
  }
}
```

- react-router
  - used to add routing feature in react application
  - routing is used to provide navigation from one to another component
  - https://reactrouter.com/
  - yarn add react-router-dom
  - route in express is mapping of
    - http method (get, post, put, delete, patch)
    - url path
    - callback function (handler)
  - route in react is mapping of
    - url path
    - component
  - BrowserRouter
    - router provided by library to implement the routing
    - to load the required comopnent by inspecting the url path

```
// step1: wrap <App /> inside BrowserRouter
// main.jsx
import { BrowserRouter } from 'react-router-dom'

// Wrap the App component inside the BrowserRouter
createRoot(document.getElementById('root')).render(
  <BrowserRouter>
    <App />
  </BrowserRouter>
)
```

```
// step2: define all the routes
// App.jsx
import { Routes, Route } from 'react-router-dom'
import Login from './pages/Login'
```

```
function App() {

  return <>
    <Routes>
      <Route path="login" element={<Login />}>
    </Routes>
  </>
}
```

- static navigation
  - navigation from one component to another without using JS code (logic)
  - navigation will be implemented by using JSX
  - use static navigation when the destination (component) needs to open without having any condition (criteria)

```
import { Link } from 'react-router-dom'

function Login() {
  return (
    <>
      <h2>Login</h2>
      <Link to='/register'>Register here</Link>
    </>
  )
}
```

- dynamic navigation
  - navigation performed using JS code
  - used dynamic navigation when the destination (componen) needs to open by validation some condition (criteria)

```
import { useNavigate } from 'react-router-dom'

function Login() {
  // get navigate() function reference
  const navigate = useNavigate()

  const onLogin = () => {
    // check if user is successfully logged in
    navigate('/home')
  }

  return (
    <>
      <h1>Login</h1>
```

```
        <button onClick={onLogin}>login</button>
      </>
    )
  }
```

- dynamic navigation with data

```
import { useNavigate } from 'react-router-dom'

function Properties() {
  const [properties, setProperties] = useState([])

  // get navigate() function reference
  const navigate = useNavigate()

  const onDetails = (property) => {
    // check if user is successfully logged in
    navigate('/details', { state: property })
  }

  return (
    <>
      <h1>Properties</h1>
      {properties.map((property) => {
        return (
          <div>
            <div>{property['title']}</div>
            <button onClick={() =>
onDetails(property)}>details</button>
          </div>
        )
      })}
    </>
  )
}
```

- tanstack router
  - use to add routing feature in react application
  - https://tanstack.com/router/latest
- bootstrap-icons
  - used to add the icons in react application
  - yarn add bootstrap-icons
- react-bootstrap-icons
  - used to add the icons in react application
  - yarn add react-bootstrap-icons

React

Single Page Application (SPA)

- a application that loads a single HTML page and dynamically updates that page as the user interacts with the app
- to develop SPAs,
    - we need to use a JavaScript framework or library
    - like
        - React
        - Angular
        - VueJs
- advantages
    - fast: similar performance to native apps
    - responsive: the app responds to user interactions (browser size changes),
        - to make the app responsive
            - we need to use CSS media queries
            - frameworks: bootstrap, tailwind
    - user-friendly

functional programming language

- function is considered as first class citizen
    - function is created as a variable of type function
- function can be passed as an argument to another function
- function can be returned from another function as return value
- map()
    - used to iterate over a collection to transform the values to new ones
    - accepts a function as a parameter which gets called every time for every value
    - the parameter function must return a transformed value for original value
    - all the transformed values will be returned a collection as a return value of map function
    - the size of returned collection is always same as original collection

```
// array of numbers
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

// get squqre of each number
const squares = numbers.map((number) => number ** 2)
```

function reference

- a reference to a function
- a variable that holds a function body's address

```
// here the function1 is a function reference
// to the function body
function function1() {
  console.log('inside function1')
}
```

export and import

- export
    - used to export any entity from a file for others to import
    - a file can export multiple entities for others

    ```
    // App.jsx
    export function App() {
      ...
    }

    // main.jsx

    // importing with same name as that of the exported entity
    import {App} from './App.jsx'

    // importing with an alias
    import {App as MyApp} from './App.jsx'
    ```

- export default
    - by default only one entity (class, function, variable, constant) can be exported from a file with default keyword

    ```
    // App.jsx
    function App() {
      ...
    }
    export default App

    // main.jsx

    // importing with same name as that of exported entity
    import App from './App.jsx'

    // importing wth an alias
    import MyApp from './App.jsx'
    ```

web storage

- web storage object is used to store collection key-value pairs
    - where key and value must be scalar value (string)
- the data will be persisted on client side (inside the browser)
- web storage is a browser specific object
- types
    - sessionStorage
        - used to store the data till the browser tab is open
        - when the tab is closed, the sessionStorage losses all the data
        - acts a temporory storage

```
// add or set a key-value in session storage
sessionStorage['username'] = 'user1'
sessionStorage.setItem('username', 'user1')

// get the value from session storage using its key
console.log(`username: ${sessionStorage['username']}`)
console.log(`username: ${sessionStorage.getItem('username')}`)

// remove the key from session storage
sessionStorage.removeItem('username')
```

    - localStorage
        - used to store the data in the form of key-value pairs
        - both the key and value must be scalar values (string)
        - the local storage will persist the data permananently (till user explicitly removes it)

```
// add or set a key-value in local storage
localStorage['username'] = 'user1'
localStorage.setItem('username', 'user1')

// get the value from local storage using its key
console.log(`username: ${localStorage['username']}`)
console.log(`username: ${localStorage.getItem('username')}`)

// remove the key from local storage
localStorage.removeItem('username')
```

React

- a JavaScript library for building user interfaces

React vs Angular

- React is a Library (developed in JS) and Angular is a Framework (developed in TypeScript)
- react development and performance is faster than angular
    - to make it faster, react uses a virtual DOM
    - it also has less memory consumption/footprint
- React has less learning curve than Angular
- React does not have any architecture whereas, Angular has a predefined architecture and tooling

important points

- do not use `class` as it is a reserved keyword in JavaScript, use `className` instead
- interpolation is done using `{}` in JSX
- interpolation always requires a scalar value and CAN NOT render an object

virtual DOM

- a lightweight copy of the real DOM (browser DOM) (document object)
- react uses virtual DOM to improve performance
- when we update the state of a component, react creates a new virtual DOM and compares it with the previous virtual DOM
- then it updates only the changed parts of the real DOM
- this process is called reconciliation

environment setup

- using CDN links
    - CDN: content delivery network
    - add react using CDN links

```
<html>
  <head>
    <!-- used for react development -->
    <script
      crossorigin
      src="https://unpkg.com/react@18/umd/react.development.js"
    ></script>

    <!-- used for react virtual dom development -->
    <script
      crossorigin
      src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"
    ></script>
  </head>

  <body>
```

```
    <div id="root"></div>
  </body>
</html>
```

- to create element use a function called `createElement`

```
// create element
// React is an object which will be used to create elements
// this object is provided by react library
(react.development.js)

// parameters
// 1st: name or type of the element (e.g. h1, h2 etc)
// 2nd: attributes or properties of the element (e.g. class, id,
style etc). this must be an object.
// 3rd: contents of the element (e.g. text, html etc)
const h2 = React.createElement('h2', {}, 'hello world')

// React 17 style of rendering an element
// get the root element
// this is the element where we will render our react elements
// const root = document.getElementById('root')

// render the element
// ReactDOM.render(h2, root)

// React 18 style of rendering an element

// create a root element
const root = ReactDOM.createRoot(document.getElementById('root'))

// render the element
root.render(h2)
```

- to create an element using JSX, use babel

```
<html>
  <head>
    <!-- used for react development -->
    <script
      crossorigin
      src="https://unpkg.com/react@18/umd/react.development.js"
    ></script>

    <!-- used for react virtual dom development -->
    <script
      crossorigin
      src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"
```

```
    ></script>

    <!-- babel compiler -->
    <script
src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>

  <body>
    <div id="root"></div>
    <script type="text/babel">
      const h2 = <h2>hello world</h2>
      const root =
ReactDOM.createRoot(document.getElementById('root'))
      root.render(h2)
    </script>
  </body>
</html>
```

- using package manager like vite

```
# install yarn on windows
> npm install -g yarn

# install yarn on linux or mac
> sudo npm install -g yarn

# create react application using vite
> npm create vite@latest <application name>
> yarn create vite <application name>

# go to the project directory
> cd <application name>

# install the dependencies
> npm install
> yarn

# start the application
> npm run dev
> yarn dev
```

  - project structure
    - node_modules
      - contains all the modules (dependencies) which are required to develop or run the application
      - will be downloaded every time when npm install or yarn install command is used
      - never commit this directory in your git repository

- public
  - directory which contains public files
  - e.g. images, audio or video which are used in the application
- src
  - directory which contains all the components of the application
  - contains
    - assets
      - directory which contains the assets (images, audio or video files)
    - pages
      - directory which contains the application pages (screens)
    - components
      - directory which contains reusable components
    - services
      - directory which contains the services
      - a service file would contain the code to call the REST APIs
    - main.jsx
      - contains the code to start react subsystem
      - contains the function createRoot(..).render()
    - index.css
      - contains global css rules which can be shared across the components in the application
    - App.jsx
      - contains the default component called as App
      - this is the startup component of every application
    - App.css
      - contains css rules need to applied on the App component
- .gitignore
  - file which contains the rules of the files which needed to be committed in the git repository
- eslint.config.js
  - contains configuration for eslint
  - lint is a program used to check the syntax of a selected language
- index.html
  - only html file in the project which starts the application
- package.json
  - contain the node configuration like name, dependencies or devDependencies etc.
- vite.config.js
  - contains the vite configuration
- yarn.lock
  - contains latest versions of the dependencies installed in the node_modules directory

react application startup

- vite will start a lite web server on port 5173
- the lite server starts loading index.html from the application
- index.html loads main.jsx file

- main.jsx calls createRoot() to create a root container to load react components
- and starts rendering first component named App
- App component start loading the user interface

component

- reusable entity which contains logic (in JS) and UI (in JSX)
- a component could as small as a part of an application
- or as big as an entire page
- types
    - class component
        - component created using a class
        - earlier (before react 16), class components were used for creating statefull component (component with state)
        - but after react 16 (in which react hooks were introduced), class components are not needed anymore
        - class components are having some overhead members compared to functional components
    - functional component
        - component created using a function
        - a javascript function which returns a JSX user interface
        - earlier (before react 16), functional components were used to create stateless components (components without state)
        - but with react 16 (react hooks), it is possible to store the state in a functional component
        - functiona components are preferred over class component
        - since the functional components do not have any overhead members, it is a way to create component compared to class component
- conventions
    - always start the component name with upper case
    - name of file should be same as the component name
    - if a component is reusable (like Person or Car), keep it in a directory named components
    - if a component is representing a page or screen, keep it in a directory named pages or screens
    - always use the props destructuring while defining the component
    - always keep one public component in a file

component life cycle

- life cycle of component
- stages the component will pass from its creation to its death
- it provides the functionality for handling the life cycle event

props

- is an object which is collection of all the properties passed to a component
- is the only way for a parent component to pass the data/properties to the child component
- props is a readonly object: the child component must not change the values sent by the parent component

```
function Person1(props) {
  return (
    <div>
      <div>name = {props['name']}</div>
      <div>address = {props['address']}</div>
    </div>
  )
}

function Person2(props) {
  const { name, address } = props
  return (
    <div>
      <div>name = {name}</div>
      <div>address = {address}</div>
    </div>
  )
}

function Person3({ name, address }) {
  return (
    <div>
      <div>name = {name}</div>
      <div>address = {address}</div>
    </div>
  )
}

function App() {
  return (
    <div>
      <Person1
        name='person1'
        address='pune'
      />
      <Person2
        name='person2'
        address='karad'
      />
      <Person3
        name='person3'
        address='satara'
      />
    </div>
  )
}
```

state

- object (collection of property-value pairs) maintained by compopent to trigger component re-render action

- if there is a change in the component state, the component will re-render itself
- unlike props, state object is readwritable
- every component will maintain its own state
- to add a state member inside a functional component use a react hook name useState()

state vs props

- state is read writable while props is read only
- when state changes, the component re-renders itself while, when props changes, component does not render itself
- state is maintained by individual component while, props will be sent by parent component to child component
- useState() hook is required to create state inside functional component while, no hook is required to send props to child component

react hook

- special function whihch starts with `use`
- types
    - built-in hooks
        - useState()
        - useEffect()
        - useReducer()
        - useMemo()
        - useContext()
        - useId()
        - useRef()
        - useCallback()
        - useNavigate()
        - useLocation()
        - useParams()
        - useSelector()
        - useDispatch()
    - custom hooks
        - user defined hooks

useState()

- hook used to add a member inside a component's state
- accepts a parameter which is the initial value of the member
- returns an array with 2 values
    - 0th position: reference to the member (used to read the value from state)
    - 1st position: reference to the function to update the value in the state object

```
function Counter() {
  // create a state to store counter value
  const [couter, setCounter] = useState(0)
```

```
  return <div>counter: {counter}</div>
}
```

useNavigate()

- react hook added by react-router-dom
- used to get the naigate() function reference
- navigate() function is used to perform dynamic navigation
- navigating from one to another component

```
import { useNavigate } from 'react-router-dom'

function Login() {
  // get navigate() function reference
  const navigate = useNavigate()

  const onLogin = () => {
    // check if user is successfully logged in
    navigate('/home')
  }

  return (
    <>
      <h1>Login</h1>
      <button onClick={onLogin}>login</button>
    </>
  )
}
```

useEffect()

- used to handle life cycle events of a component
- accepts 2 parameters
    - 1st parameter: callback function
    - 2ns parameter: dependency array
- event1: component is loaded (componentDidMount)
    - dependency array must be empty
    - the callback function gets called when the component gets mounted
    - it is similar to the constructor method in any class
    - it gets called only function

```
function Home() {
  // this code here will handle the component did mount event
  useEffect(() => {
```

```
    // this function gets called immediately after component is
mounted
    console.log('Home component is mounted')
  }, [])

  return (
    <div>
      <h1>Home</h1>
    </div>
  )
}
```

- event2: component is unloaded (componentDidUnmount)
  - it gets called when the component gets removed from the screen
  - it gets called only once in its life cycle
  - dependency array must be empty
  - similar to destructor of any class

```
function Home() {
  // this code here will handle the component did mount event
  useEffect(() => {
    return () => {
      // this function gets called just before the component is
unloading from screen
      console.log('component is getting unloaded')
    }
  }, [])

  return (
    <div>
      <h1>Home</h1>
    </div>
  )
}
```

- event3: component state is changed
  - called as soon as the state of a component changes
  - this event gets fired irrespective of any state member
  - must NOT pass the dependency array

```
function Home() {
  const [n1, setN1] = useState(10)
  const [n2, setN2] = useState(20)

  const onUpdateN1 = () => {
    setN1(n1 + 1)
  }
```

```
  const onUpdateN2 = () => {
    setN2(n2 + 1)
  }

  useEffect(() => {
    // this function here will get called everytime when
    // the state changes
    console.log(`state changed..`)
  })

  return (
    <div>
      <Navbar />
      <div className='container'>
        <h1 className='page-header'>Dummy</h1>

        <div>n1: {n1}</div>
        <div>
          <button
            onClick={onUpdateN1}
            className='btn btn-success'
          >
            Update N1
          </button>
        </div>
        <div>n2: {n2}</div>
        <div>
          <button
            onClick={onUpdateN2}
            className='btn btn-success'
          >
            Update N2
          </button>
        </div>
      </div>
    </div>
  )
}
```

- event4: component state is changed
    - called as soon as the state of a component changes because of a required dependency

```
function Home() {
  const [n1, setN1] = useState(10)
  const [n2, setN2] = useState(20)

  const onUpdateN1 = () => {
    setN1(n1 + 1)
  }
```

```
  const onUpdateN2 = () => {
    setN2(n2 + 1)
  }

  useEffect(() => {
    // this function gets called when n1 changes
    console.log(`n1 changed..: ${n1}`)
  }, [n1])

  useEffect(() => {
    // this function gets called when n1 changes
    console.log(`n2 changed..: ${n2}`)
  }, [n2])

  return (
    <div>
      <Navbar />
      <div className='container'>
        <h1 className='page-header'>Dummy</h1>

        <div>n1: {n1}</div>
        <div>
          <button
            onClick={onUpdateN1}
            className='btn btn-success'
          >
            Update N1
          </button>
        </div>
        <div>n2: {n2}</div>
        <div>
          <button
            onClick={onUpdateN2}
            className='btn btn-success'
          >
            Update N2
          </button>
        </div>
      </div>
    </div>
  )
}
```

JSX

- a syntax extension for JavaScript
- allows you to write HTML code inside JavaScript
- how does it work?
    - babel is used to convert JSX code into JavaScript code

- babel is a JavaScript compiler

```
<script type="text/babel">
  // JSX code
  const h2 = <h2>hello world</h2>

  // babel converts the above code into the following code
  // const h2 = React.createElement('h2', {}, 'hello world')
</script>
```

context api

- it is a built-in feature of react used to share data among multple components
- context
  - instance of Context component used to provide (share) data
  - to create a context use createContent() function
- to share the context with components use following syntax
  - ...

```
import { createContext, useState } from 'react'

// create a context to share the data
export const CounterContext = createContext()

function App() {
  const [counter, setCounter] = useState(0)
  return (
    <div>
      <h1>App Component</h1>
      <CounterContext value={{ couter, setCounter }}>
        <Counter1 />
        <Counter2 />
      </CounterContext>
    </div>
  )
}
```

- to use the contxt in the child components, use useContext() react hook

```
import { CounterContext } from '../App'

function Counter1() {
  // get the counter and setCounter from counter context created in
App.jsx
  const { counter, setCounter } = useContext(CounterContext)

  const onIncrement = () => setCounter(counter + 1)
```

```
  return (
    <div>
      <div>counter: {counter}</div>
      <div>
        <button onClick={onIncrement}>increment</button>{' '}
      </div>
    </div>
  )
}
```

- use case: to share simple values like
    - login status
    - themse used in the application

VS extensions

- auto import:
    - https://marketplace.visualstudio.com/items/?itemName=NuclleaR.vscode-extension-auto-import
- auto tag renamer:
    - https://marketplace.visualstudio.com/items/?itemName=formulahendry.auto-rename-tag
- code snippets for react:
    - https://marketplace.visualstudio.com/items/?itemName=rodrigovallades.es7-react-js-snippets

external libraries

```
# install any package
> npm install <package name>
> yarn add <package name>
```

- react toastify
    - used to show the toast message
    - https://www.npmjs.com/package/react-toastify
    - yarn add react-toastify

```
import { ToastContainer } from 'react-toastify'

function App() {
  return (
    <div>
      <ToastContainer />
    </div>
  )
```

```
}

// to show the toast messages
// import {toast} from 'react-toastify'

// toast.warn('this is warning message')
// toast.error('this is error message')
// toast.info('this is info message')
// toast.success('this is success message')
```

- axios
  - used to make API calls
  - https://www.npmjs.com/package/axios
  - yarn add axios

```
import axios from 'axios'

async function makePostCall(email, password) {
  try {
    const url = 'http://localhost:4000/user/login'
    const body = { email, password }
    const response = await axios.post(url, body)
    console.log(response.data)
  } catch (ex) {
    console.log('exception: ', ex)
  }
}

async function makePostCallWithToken(title, description, price) {
  try {
    const url = 'http://localhost:4000/property/'
    const body = { title, description, price }
    const token = sessionStorage.getItem('token')
    const response = await axios.post(url, body, {
      headers: { token },
    })
    console.log(response.data)
  } catch (ex) {
    console.log('exception: ', ex)
  }
}

async function makeGetCall() {
  try {
    const url = 'http://localhost:4000/property'
    const response = await axios.get(url)
    console.log(response.data)
  } catch (ex) {
    console.log('exception: ', ex)
  }
```

```
}

async function makeGetCallWithToken() {
  try {
    const url = 'http://localhost:4000/my'
    const token = sessionStorage.getItem('token')
    const response = await axios.get(url, {
      headers: { token },
    })
    console.log(response.data)
  } catch (ex) {
    console.log('exception: ', ex)
  }
}
```

- react-router
  - used to add routing feature in react application
  - routing is used to provide navigation from one to another component
  - https://reactrouter.com/
  - yarn add react-router-dom
  - route in express is mapping of
    - http method (get, post, put, delete, patch)
    - url path
    - callback function (handler)
  - route in react is mapping of
    - url path
    - component
  - BrowserRouter
    - router provided by library to implement the routing
    - to load the required comopnent by inspecting the url path

```
// step1: wrap <App /> inside BrowserRouter
// main.jsx
import { BrowserRouter } from 'react-router-dom'

// Wrap the App component inside the BrowserRouter
createRoot(document.getElementById('root')).render(
  <BrowserRouter>
    <App />
  </BrowserRouter>
)
```

```
// step2: define all the routes
// App.jsx
import { Routes, Route } from 'react-router-dom'
import Login from './pages/Login'
```

```
function App() {

  return <>
    <Routes>
      <Route path="login" element={<Login />}>
    </Routes>
  </>
}
```

- static navigation
  - navigation from one component to another without using JS code (logic)
  - navigation will be implemented by using JSX
  - use static navigation when the destination (component) needs to open without having any condition (criteria)

```
import { Link } from 'react-router-dom'

function Login() {
  return (
    <>
      <h2>Login</h2>
      <Link to='/register'>Register here</Link>
    </>
  )
}
```

- dynamic navigation
  - navigation performed using JS code
  - used dynamic navigation when the destination (componen) needs to open by validation some condition (criteria)

```
import { useNavigate } from 'react-router-dom'

function Login() {
  // get navigate() function reference
  const navigate = useNavigate()

  const onLogin = () => {
    // check if user is successfully logged in
    navigate('/home')
  }

  return (
    <>
      <h1>Login</h1>
```

```
        <button onClick={onLogin}>login</button>
      </>
    )
  }
```

- dynamic navigation with data

```
import { useNavigate } from 'react-router-dom'

function Properties() {
  const [properties, setProperties] = useState([])

  // get navigate() function reference
  const navigate = useNavigate()

  const onDetails = (property) => {
    // check if user is successfully logged in
    navigate('/details', { state: property })
  }

  return (
    <>
      <h1>Properties</h1>
      {properties.map((property) => {
        return (
          <div>
            <div>{property['title']}</div>
            <button onClick={() =>
onDetails(property)}>details</button>
          </div>
        )
      })}
    </>
  )
}
```

- tanstack router
  - use to add routing feature in react application
  - https://tanstack.com/router/latest
- bootstrap-icons
  - used to add the icons in react application
  - yarn add bootstrap-icons
- react-bootstrap-icons
  - used to add the icons in react application
  - yarn add react-bootstrap-icons

- redux-toolkit
    - used to implement redux architecture in react application
    - redux provides global state management
    - global state management is implmented using a (global) store which will be accessible to all the components in the react application
    - components
        - store
            - a collection of slices
            - slice is a feature which is meant to store some data in the form of key-value pairs
        - action
            - is an event which can be fired to either read the contents of a slice or to update/modify the cotents of a slice
            - an action is responsible for changing the state of store
        - reducer
            - a function or event handler which gets invoked when an action is fired
            - an action uses reducer to update the store state
            - reducer contains the logic to update the store state
    - installation
        - yarn add @reduxjs/toolkit react-redux
    - implementation of redux in react application
        - step1: create an empty store in the application

```
// src/store.js

import { configureStore } from '@reduxjs/toolkit'

// create a store
export const store = configureStore({
  reducer: {},
})
```

        - step2: add the store in the react application

```
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'
import { Provider } from 'react-redux'
import { store } from './store.js'

createRoot(document.getElementById('root')).render(
  <Provider store={store}>
```

```
    <App />
  </Provider>
)
```

- step3: create a slice

```
// features/counter.slice.js
import { createSlice } from '@reduxjs/toolkit'

// create a slice
const counterSlice = createSlice({
  // unique name to identify the slice inside the store
  name: 'counter',

  // state to be maintained by the slice
  initialState: {
    count: 0,
  },

  // collection of actions and their respective reducers
  reducers: {
    // incrementAction is the action
    // the function is the reducer
    incrementAction: (state) => {
      // update the state
      state.count += 1
    },
    decrementAction: (state) => {
      state.count -= 1
    },
  },
})

// export actions
export const { incrementAction, decrementAction } =
counterSlice.actions

// export the reducers
export default counterSlice.reducer
```

- step4: add the slice into the store

```
// src/store.js

import { configureStore } from '@reduxjs/toolkit'

// create a store
export const store = configureStore({
```

```
  reducer: {},
})
```

- useSelector()
  - used to read the contents from store

```
import { useSelector } from 'react-redux'

// read the count value from store
const { count } = useSelector((store) => store.counter)
```

- useDispatch()
  - used to update the store's state
  - returns a dispatch function reference which is used to update the store state

```
import { useDispatch } from 'react-redux'
import { incrementAction } from '../features/counter.slice'

function Counter() {
  // get the dispatch function reference
  const dispatch = useDispatch()

  // increment the count
  const onIncrement = () => {
    // send the increment action
    dispatch(incrementAction())
  }

  return (
    <div>
      <button onClick={onIncrement}>increment</button>
    </div>
  )
}
```

- moment
  - used to manipulate the date or time
  - yarn add moment