COSC

COSC

REACTJS & FASTAPI BOOTCAMP

13th – 25th
AUGUST

TPO LABS
CBIT

COSC ReactJS and FastAPI Bootcamp

BOOTCAMP OVERVIEW

- This boot camp will provide hands-on experience with ReactJS, focusing on component-based architecture, state management, and hooks.
- This boot camp will also cover FastAPI, emphasizing speed and simplicity in creating robust backend services.
- You will learn to integrate ReactJS with FastAPI, enabling them to build full-stack web applications.
- Also, a structured schedule includes practical assignments and a final assessment, with certificates awarded upon completion

HTML

What is HTML?
Hypertext Markup Language (HTML) is a tag-based language used to format static content, which is unchanging information.
Dynamic content, on the other hand, uses scripting languages, applets, or Flash files to interact with the user.
HTML files have file name extensions of either .htm (from the older three-character file name extension limit) or .html.
HTML files are ASCII files, meaning they are text-based files.

Creating HTML Pages
Develop the entire site or its sections locally on your computer
You can create the site on your PC using two approaches:

1. The easy way:
Design your page visually using an integrated development environment (IDE) that converts your design into HTML code.
Tools like Webflow, Wix, Squarespace, or Adobe XD offer intuitive drag-and-drop interfaces and automatically generate the necessary HTML, CSS, and JavaScript for you.

2. The hard way:
Hand-code the HTML using a plain text editor like Visual Studio Code.

1. Use Visual Studio Code to hand-code your HTML.
2. Save the file with a `.html` extension on your computer.
3. Open the file in your browser or refresh to view changes.
4. Edit, save, and test repeatedly.

Introduction to HTML:
We won't cover everything in HTML, we will cover the basics, and you can look up anything else when the need arises.
The homepage should be named *index.html*
This is the default file a web server delivers if no specific file is requested. Search engines also look for this file.
Make sure to use this naming convention for your homepage every time.

If there are errors in your HTML, the browser might still try to display your page. However, "try" is the key word,your page might not display at all or may not look as you intended. Unlike compilers, browsers don't provide error messages to help you diagnose issues.

Tag Overview:
HTML wraps content in tags enclosed by < >. Most tags require both a start tag and an end tag, with the content placed between them.

For Example:

```
<p> CBIT Open Source Community </p>
```

A start tag and its end tag are viewed as a container.
In some situations, in some browsers, the browser may forgive you if you forget an end tag, but other browsers in other situations don't.
Well-formed HTML following the newer standards requires the end tag even if the browser lets you get away with omitting it.

Some tags have only a start tag, with all the necessary information embedded within the tag.
For Example:

```
<img src="COSC-logo.jpg"/>
```

or

```
<img src="COSC-logo.jpg"/>
```

Tags that start with <! never display within the browser. There are two types:

1. Comments

```
<!-- This is a comment -->
```

2. DOCTYPE tag: it is used to identify that the file is a html code.

```
<!DOCTYPE html>
```

- An html code starts with the tag <html> and ends with the tag </html>.

[an example picture can be shown]

- The <html> tag contains two direct child elements:
    1. <head> </head>
    2. <body> </body>

<head> : head tag contains additional information such as

- Meta data such as:

```
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

The <meta charset="UTF-8"> tag specifies the character encoding for the HTML document. UTF-8 is a widely used encoding that supports virtually all characters and symbols from all languages. The following tag is used to control the layout on mobile browsers.

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

- <title> : used to give the web page a title
- <link>: It is used to link other files such as css files to the html file

```
<link rel="stylesheet" href="/Users/example/example.css">
```

- <script>: it is used to link Javascript file to the html file.

```
<script>src=loginjs.js</script>
```

<body> : The body tag contains all the elements which go into the webpage.

- <h1>: h1 tag is used to write headings in the webpage.
- <p>: p tag is used to write paragraphs in the webpage.
-
: This tag is used to break the line and go to the next line.

SOME OTHER IMPORTANT TAGS:

1) *<img>:* The img tag is used to place images in the webpage. Img tag is a self closing tag meaning you don't need to add the end tag (</img>). The img tag also consists elements like:

1. **Src:** 'src' is used to specify the path to the image file. This can be a relative path or a URL.
2. **Alt:** This element is used to display an alternate text if the image is not loading.

For Example:

```
<img src="https://cbitosc.github.io/logo.jpg" alt="Description of the image">
```

2) : div tag is used to group different elements. Div tag creates a section for the web page which can be easily styled and managed. The div tag closes with the end tag .

```
<div class="about-content">
```

```
    <h2>COSC </h2>
    <p> COSC is a technical club.</p>
 </div>
```

Styling a div tag:

The div tag specified in the html file can be styled in the css file with the attributes like height, weight, background color etc.
For Example:

```
.myDiv {
    background-color: lightblue;
    padding: 20px;
    margin: 10px 0;
    }
```

3) *<a> :* The <a> tag in HTML is used to create hyperlinks, allowing users to navigate from one web page to another or to specific sections within the same page. It is one of the fundamental elements of web design, enabling interactivity and connectivity between different web resources.

Characteristics of the <a> Tag:

Href : The href attribute specifies the URL of the page the link goes to.
In the following example we can see that the href part specifies google website

```
<a href="https://www.google.com">Visit Google</a>
```

When the user clicks on 'Visit Google' in the webpage the webpage takes it to www.google.com website.

Text : The content to be clicked on can be written in between the <a> and tags.
In the following example we can see that the text 'Visit Google' is written in between the <a> and tags.

```
<a href="https://www.google.com">Visit Google</a>
```

Target : The target attribute can be used to open the link in a new tab or window.

For Example:

```
<a href="https://www.example.com" target="_blank">Visit Example</a>
```

3) **lists** : In HTML, lists are used to group related items together, making content more organized and easier to read. There are three main types of lists: ordered lists, unordered lists, and description lists. The tag for lists can be simply represented as <li> and .

1. **Ordered list:** An ordered list is used when the order of the items is important. Items in an ordered list are typically displayed with numbers. The ordered list is represented with start tag <ol> and end tag .

For Example:

```
<ol>
    <li>First item</li>
    <li>Second item</li>
    <li>Third item</li>
</ol>
```

2. **Unordered list:** An unordered list is used when the order of the items does not matter. Items in an unordered list are typically displayed with bullet points. The unordered list is represented with start tag <ul> and end tag .

For Example:

```
<ul>
    <li>apple</li>
    <li>mango</li>
    <li>orange</li>
</ul>
```

3. **Description list:** Description list is written in between the tags <dl> and </dl>. A description list is used to group terms and their descriptions. It consists of pairs of <dt> (description term) and <dd> (description definition) elements.

For Example:

```
<dl>
    <dt>HTML</dt>
    <dd>HyperText Markup Language.</dd>
    <dt>CSS</dt>
    <dd>Cascading Style Sheets.</dd>
</dl>
```

4. **Table:** Tables in HTML are used to organize and present data in a structured format, consisting of rows and columns. Tables are represented with the start tag <table> and end tag </table>. The rows are represented with <tr> tags and columns are represented with <td> tags inside the <tr> </tr> tags.

For Example:

```
<table>
    <tr>
```

```
            <th>Name</th>
            <th>Age</th>
            <th>Email</th>
    </tr>
    <tr>
            <td>John Doe</td>
            <td>25</td>
            <td>john@example.com</td>
    </tr>
    <tr>
            <td>Jane Smith</td>
            <td>30</td>
            <td>jane@example.com</td>
    </tr>
</table>
```

4) Form : Forms in HTML are essential for collecting user input and facilitating interaction on web pages. They allow users to submit data to a server for processing, such as logging in, signing up, or providing feedback.

Basic Structure of a Form:

- <form>: All the information is written in between the <form> tags.
- <input>: The input given by the user is collected through <input> tag. The different input fields which can be specified are text, password, checkbox, radio, etc.

```
Ex:
<input type="text" id="username" name="username" required>
<input type="password" id="password" name="password" required>
<input type="submit" value="Login">
```

- <label>: The label tag provides a label for an input element, improving accessibility.

For Example:

```
<label for="name">Name:</label>
<input type="text" id="name" name="name" required>
```

- <select>:The select tag creates a dropdown list for selecting options.

For Example:

```
<select id="country" name="country" required>
       <option value="">--Please choose an option--</option>
```

```
        <option value="nepal">Nepal</option>
        <option value="usa">USA</option>
        <option value="canada">Canada</option>
        <option value="australia">Australia</option>
</select>
```

- <button>: The button tag is used as a clickable button to submit the form.

For Example:

```
<button type="submit">Submit</button>
```

Form Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Registration Form</title>

</head>
<body>

<div class="container">
    <h2>Registration Form</h2>

    <p>Please fill in this form to create an account.</p>

    <img src="https://via.placeholder.com/150" alt="Registration Image" style="display:
block; margin: 0 auto;">

    <form action="/submit_registration" method="post">
        <table>
            <tr>
                <th><label for="fname">First Name:</label></th>
                <th><input type="text" id="fname" name="firstname" required></th>
            </tr>
            <tr>
                <td><label for="lname">Last Name:</label></td>
                <td><input type="text" id="lname" name="lastname" required></td>
            </tr>
```

```
            <tr>
                <td><label for="email">Email:</label></td>
                <td><input type="email" id="email" name="email" required></td>
            </tr>
            <tr>
                <td><label for="password">Password:</label></td>
                <td><input type="password" id="password" name="password" required></td>
            </tr>
            <tr>
                <td><label for="country">Country:</label></td>
                <td>
                    <select id="country" name="country" required>
                        <option value="">Select your country</option>
                        <option value="usa">United States</option>
                        <option value="canada">Canada</option>
                        <option value="uk">United Kingdom</option>
                        <option value="australia">Australia</option>
                    </select>
                </td>
            </tr>
        </table>

        <div style="text-align: center;">
            <input type="submit" value="Register">
            <button type="reset">Reset</button>
        </div>
    </form>

    <p>Already have an account? <a href="/login">Login here</a>.</p>
</div>

</body>
</html>
```

CSS

What is CSS?
CSS, or Cascading Style Sheets, is a styling language used to describe the presentation of a document written in HTML or XML. It allows developers to control the layout, color, fonts, and overall visual aesthetics of web pages, making them more attractive and user-friendly. By separating content from design, CSS enhances maintainability and flexibility in web development.

Basic Syntax

```
<head>
    <style>
        h1 {
         color: red;
        }
    </style>
</head>
```

This sets the text color of all `<h1>` elements to red.

Including styles
Including styles in CSS can be done in three main ways: inline, internal, and external.

1.**Inline CSS**: This involves adding styles directly to the HTML element using the style attribute.

Syntax :

```
    <h1 style="color: red;">This is a heading</h1>
```

This sets the text color of this specific <h1> element to red.

2.**Internal CSS**:This involves adding a <style> tag within the <head> section of the HTML document.

Syntax :

```
<head>
    <style>
        h1 {
         color: red;
        }
    </style>
</head>
```

This sets the text color of all <h1> elements in the document to red.

3.**External CSS**: This involves linking to an external CSS file using the <link> tag within the <head> section.

Syntax :
HTML FILE :

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

CSS FILE :

```
  h1{
        color:red;
    }
```

This sets the text color of all <h1> elements in the document to red, using an external stylesheet.

Color Property : Used to set the color of foreground

Syntax :`

```
color: red;
color: pink;
color: blue;
color: green;
```

Background Color Property: Used to set the color of background

Syntax :

```
background-color: red;
background-color: pink;
background-color: blue;
background-color: green;
```

Selectors

Selectors in CSS are used to target HTML elements that you want to style. There are various types of selectors, each serving a different purpose.
1.Universal Selector (*):

- Selects all elements on the page.

Syntax:

```
* {
  margin: 0;
  padding: 0;
}
```

This removes the default margin and padding from all elements.

2.Element Selector:

- Targets all instances of a specific HTML element.

Syntax:

```
h1 {
  color: blue;
}
```

This sets the text color of all <h1> elements to blue.

3.Id Selector (#):

- Targets a single element with a specific id attribute.

Syntax:

```
        .myClass {
  background-color: yellow;
}
```

This sets the font size of the element with the id="myId" to 20 pixels.

4.Class Selector (.):

- Targets all elements with a specific class attribute.

Syntax :

```
        .myClass {
  background-color: yellow;
}
```

This sets the background color of all elements with the class="myClass" to yellow.

Selectors Example:

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>CSS Selectors Demo</title>
    <style>
        /* Universal Selector */
        * {
            margin: 10px;
            padding: 10px;
            box-sizing: border-box;
            font-family: Arial, sans-serif;
        }

        /* Element Selector */
        h1 {
            color: rgb(91, 91, 207); /* Applies to all <h1> elements */
            text-align: center;
        }



 /* Class Selector */
        .highlight {
            background-color: rgb(144, 102, 127); /* Applies to all elements with
class="highlight" */
            padding: 10px;
        }
        /* ID Selector */
        #unique {
            color: rgb(214, 42, 42); /* Applies to the element with id="unique" */
            font-size: 1.5em;
```

```
        }

    </style>

</head>

<body>

    <h1>CSS Selectors Demonstration</h1>

    <p class="highlight">This paragraph has a pink background and padding.</p>

    <p id="unique">This paragraph has a unique style with red text and larger font
size.</p>

    <p>This is a regular paragraph.</p>

    <div>

        <p class="highlight">Another highlighted paragraph inside a div.</p>

    </div>

</body>

</html>
```

Box Model in CSS

The Box Model in CSS describes the structure of an HTML element, consisting of four parts: content, padding, border, and margin. It defines how these components are layered and how they affect the element's layout and spacing on the page.

Here are examples for each component of the CSS Box Model:
Height: Sets the height of the content area.

```
div {
height: 50px;
}
```

Width: Sets the width of the content area.

```
div {
```

```
width: 50px;
}
```

Border: Adds a border around the padding and content areas. You can define its style, width, and color.
border-width : 2px;
border-style : solid / dotted / dashed
border-color : black;

Syntax :

```
.box {
  border: 2px solid black;
}
```

To round the corners of an element's outer border edge , we use :
border-radius : 10px;
border-radius : 50%;

Padding: Adds space between the content and the border.
Padding-bottom
Padding-top
Padding-left
Padding-right

Syntax :

```
.box {
  padding: 20px;
}
```

***Shorthand** : padding: 1px 2px 3px 4px;*

Margin: Creates space outside the border, separating the element from other elements.
Margin-right
Margin-left
Margin-top
Margin-bottom
Syntax :

```
.box {
  margin: 30px;
}
```

***Shorthand** : margin: 1px 2px 3px 4px;*

Box Model Example:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>CSS Box Model Demo</title>
    <style>
        /* Applying box model styles */
        .box {
            width: 200px;
            height: 150px;
            background-color: lightblue;
            padding: 20px;  /* Space between content and border */
            border: 5px solid navy; /* Border around the box */
            margin: 30px; /* Space outside the border */
            box-sizing: border-box; /* Includes padding and border in width/height */
        }
    </style>
</head>
<body>
    <div class="box">
        This is a box model demonstration.
    </div>
</body>
</html>
```

Position

The position CSS property sets how an element is positioned in a document.

Here are examples for each CSS position property:

Static: This is the default position. The element is positioned according to the normal flow of the document, and the top, right, bottom, left, and z-index properties have no effect.

```
.static-box {
  position: static;
  /* Default behavior, no effect from top, right, bottom, left, or z-index */
}
```

Relative: The element is positioned relative to its normal position. The top, right, bottom, and left properties offset the element from where it would normally be, and z-index can be used to stack the element.

```
.relative-box {
  position: relative;
  top: 10px;
  left: 20px;
  /* Moves the element 10px down and 20px to the right from its normal position */
}
```

Absolute:
The element is positioned relative to its closest positioned ancestor (an ancestor with a position other than static). It is removed from the normal document flow, so it does not affect the position of other elements.

```
.relative-box {
  position: relative;
  top: 10px;
  left: 20px;
  /* Moves the element 10px down and 20px to the right from its normal position */
}
```

Fixed:
The element is positioned relative to the browser window and remains in place even when the page is scrolled. It is also removed from the normal document flow.

```css
.fixed-box {
  position: fixed;
  bottom: 10px;
  right: 20px;
  /* Positioned 10px from the bottom and 20px from the right of the browser window */
}
```

Sticky:
The element toggles between relative and fixed positioning depending on the user's scroll position. It acts like a relative element until it reaches a specified scroll position, then it behaves like a fixed element.

```css
.sticky-box {

  position: sticky;

  top: 0;

  /* Sticks to the top of the viewport when scrolled to the top */

}
```

Link : https://developer.mozilla.org/en-US/docs/Web/CSS/position

CSS Positioning Example:

```html
<!DOCTYPE html>

<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>CSS Positioning Example</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            line-height: 1.6;
            margin: 0;
            padding: 0;
        }
        .container {
            height: 300px;
            position: relative;
            background-color: #f0f0f0;
            margin-bottom: 50px;
            padding: 10px;
        }
        .box {
            width: 100px;
            padding: 10px;
            color: white;
            margin: 10px;
        }
        .static { background-color: #4CAF50; position: static; }
        .relative { background-color: #2196F3; position: relative; margin-top: 40px; } /*
Adjusted to avoid overlap */
```

```
        .absolute { background-color: #FF5722; position: absolute; top: 100px; left: 50px;
}

        .fixed { background-color: #9C27B0; position: fixed; top: 10px; right: 10px; }

        .sticky { background-color: #FFC107; position: -webkit-sticky; position: sticky;
top: 0; }

        .content {

            padding: 20px;

        }

        .dummy-text {

            height: 1000px;

        }

    </style>

</head>

<body>



    <div class="container">

        <div class="box static">Static</div>

        <div class="box relative">Relative</div>

        <div class="box absolute">Absolute</div>

        <div class="box sticky">Sticky (Scroll down)</div>

    </div>



    <div class="content">

        <p>This is some dummy content to demonstrate how the different positioning works.
Keep scrolling to see the effect of the sticky and fixed elements.</p>

        <div class="dummy-text"></div>

    </div>



    <div class="box fixed">Fixed (Always here)</div>
```

```
    <div class="content">
        <p>More content after the fixed element.</p>
        <div class="dummy-text"></div>
    </div>
</body>
</html>
```

Text Properties

Text properties in CSS control various aspects of text appearance on a webpage, such as alignment, size, and style.

1. **text-align:**

Sets the horizontal alignment of text within its container.

- Syntax:

```
`text-align: left | right | center | justify;`
```

2. **font-family:**

Specifies the typeface to be used for the text.

- Syntax:

```
`font-family: font-name, generic-family;`
```

3. **font-size:**

Defines the size of the text.

- Syntax:

```
`font-size: size;` (e.g., `font-size: 16px;`)
```

4. **font-weight:**

Controls the thickness of the text.

- Syntax:

```
`font-weight: normal | bold | 100 | 200 | ... | 900;`
```

5. **line-height:**

Sets the amount of space between lines of text.

- Syntax:

```
`line-height: normal | number | length;` (e.g., `line-height: 1.5;`)
```

6. **text-transform:**

Alters the capitalization of text.

- Syntax:

```
`text-transform: capitalize | uppercase | lowercase | none;`
```

7. text-decoration:

Adds decorations to text like underline or strikethrough.

- Syntax:

```
`text-decoration: none | underline | overline | line-through;`
```

8. letter-spacing:

Adjusts the space between characters.

- Syntax:

```
`letter-spacing: length;` (e.g., `letter-spacing: 2px;`)
```

Display Property :

The display property in CSS controls the layout and visibility of elements on a webpage. It determines how an element is rendered in the document flow.

1. Inline - Takes only the space required by the element. (no margin/ padding)

Example:

```
 span {
display: inline;
 }
```

2.block - Takes full space available in width.

Example:

```
div {
 display: block;
 }
```

3.inline-block - Similar to inline but we can set margin & padding.

Example:

```
div {
 display: none;
 }
```

4.none - To remove element from document flow.

Example:

```
div {
 display: none;
 }
```

Units in CSS

In CSS, units specify the size or spacing of elements. One common unit is pixels (px).

- Pixels (px): Pixels are a fixed unit of measurement used to define the size of elements on a screen. For example, 96px roughly equals 1 inch. Using font-size: 2px; sets the text size to 2 pixels, which is very small and typically not readable.

This unit is absolute and does not change with screen size or resolution, making it useful for precise control over element dimensions.

pixels (px) :

96px = 1 inch

font-size: 2px;

Background Image

Used to set an image as background

background-image : url("image.jpeg");

background-image : url("image src link");

Background Size

Used to set background size

background-size : cover / contain / auto

Flexbox

Flexible Box Layout

It is a one-dimensional layout method for arranging items in rows or columns.

Flexbox Direction

It sets how flex items are placed in the flex container, along which axis and direction.

flex-direction : row; (default)

flex-direction : row-reverse;

flex-direction : column;

flex-direction : column-reverse;

Flex Properties for Flex Container

1.justify-content : alignment along the main axis.

flex-start / flex-end / centre / space-evenly /

2.flex-wrap : nowrap / wrap / wrap-reverse

3.align-items : alignment along the cross axis.

4.align-content : alignment of space between & around the content along cross-axis

Flex Properties for Flex Item

1.align-self : alignment of individual along the cross axis.

2.flex-grow : how much a flex item will grow relative to the rest of the flex items if space is available

3.flex-shrink : how much a flex item will shrink relative to the rest of the flex items if space is available

FlexBox Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Flexbox Layout Example</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            margin: 0;
            padding: 0;
            background-color: #f8f9fa;
```

```
}

.flex-container {
    display: flex;          /* Enables Flexbox */
    flex-wrap: wrap;        /* Allows items to wrap onto multiple lines*/
    justify-content: space-around; /* Distributes items with equal space
                                                                around*/
    align-items: center;     /* Centers items vertically within the
                                                            container*/
    padding: 20px;
    background-color: #343a40;
}

.flex-item {
    background-color: #007bff;
    color: white;
    padding: 20px;
    margin: 10px;
    font-size: 18px;
    text-align: center;
    border-radius: 5px;
    flex: 1 1 150px;   /* Flexible basis with growth and shrink values */
    max-width: 300px;        /* Limits the width of each item */
    min-width: 150px;        /* Ensures a minimum width for each item */
}

.header, .footer {
```

```
            background-color: #6c757d;
            color: white;
            text-align: center;
            padding: 10px;
        }
    </style>
</head>
<body>

    <div class="header">
        <h1>Flexbox Layout Example</h1>
    </div>

    <div class="flex-container">
        <div class="flex-item">Item 1</div>
        <div class="flex-item">Item 2</div>
        <div class="flex-item">Item 3</div>
        <div class="flex-item">Item 4</div>
        <div class="flex-item">Item 5</div>
    </div>

    <div class="footer">
        <p>Footer Content</p>
    </div>

</body>
</html>
```

Transitions

Transitions in CSS allow smooth animation between two states of an element, enhancing user interactions.

Transition Basics: Transitions define how elements change from one state to another. They are used to add animation and improve user experience.

- **Transition Property**: Specifies which CSS properties should be animated
  Syntax :
  transition-property : property you want to transition (font-size, width etc.)
- **Transition Duration**: Defines how long the transition takes to complete (e.g., 2s for 2 seconds, 400ms for 400 milliseconds).
  Syntax :
  transition-duration : 2s / 4ms
- **Transition Timing Function**: Controls the pace of the transition, affecting acceleration and deceleration.
  - ease: Default, gradual acceleration and deceleration.
  - ease-in: Starts slow and accelerates.
  - ease-out: Slows down towards the end.
  - linear: Constant speed.

Syntax :

transition-timing-function : ease-in / ease-out / linear / steps ..

- **Transition Delay**: Optional, delays the start of the transition effect.

  Syntax :

transition-delay : 2s / 4ms ..

Transition Examples:

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>CSS Transitions Example</title>
```

```
    <style>
        body {
            font-family: Arial, sans-serif;
            background-color: #f8f9fa;
            display: flex;
            justify-content: center;
            align-items: center;
            height: 100vh;
            margin: 0;
        }

        .box {
            width: 150px;
            height: 150px;
            background-color: #007bff;
            border-radius: 5px;
            transition: background-color 0.5s ease, transform 0.5s ease, width 0.5s ease;
            /* This line specifies the properties to animate and the duration */
        }

        .box:hover {
            background-color: #28a745; /* Changes background color */
            transform: rotate(45deg) scale(1.2); /* Rotates and scales the box */
            width: 200px; /* Increases the width */
        }
    </style>
</head>
```

```
<body>


    <div class="box"></div>


</body>

</html>
```

Media Queries

Media queries in CSS allow you to apply styles based on various conditions, such as screen size, device orientation, and resolution. They are essential for creating responsive websites that adapt to different devices and screen sizes.

Syntax :

```
@media (width : 600px) {
div {
background-color : red;
}
}
```

Alternate means :

@media (min-width : 200px) and (min-width : 300px) {

div {

background-color : red;

}

CSS Transform

CSS transform allows you to apply 2D and 3D transformations to an element, changing its shape, position, and orientation. This can be used to add various animations and effects.

1.**Rotate**

The rotate function rotates an element around a fixed point.

Syntax :

```
transform: rotate(45deg);
```

Rotates the element 45 degrees.

Angles are measured in degrees, with positive values for clockwise rotation and negative for counterclockwise.

2.**Scale**

The scale function resizes an element along the x and y axes.

Syntax :

```
transform: scale(2);       #  Scales the element to twice its size
transform: scale(0.5);    # Scales the element to half its size
transform: scaleX(0.5);  #  Scales the element's width to half
transform: scaleY(0.5);  #  Scales the element's height to half
transform: scale(1, 2);    #  Scales the element's width by 1 and height by 2
```

Animation in CSS

CSS animations allow you to animate elements over time, enhancing the user experience with dynamic effects.

Keyframes

Keyframes define the stages and styles of an animation using the @keyframes rule.

Syntax :

```
@keyframes myName {
  from {
    font-size: 20px;
  }
  to {
    font-size: 40px;
  }
}
```

Animation Properties

CSS animations are controlled by a set of properties that define the name, duration, timing, delay, iteration, and direction of the animation.

1.**Animation-name**

Specifies the name of the @keyframes animation to apply to the element.

Syntax:

```
.element {
  animation-name: myName;
}
```

2.**Animation-duration**

Defines how long the animation takes to complete one cycle.

Syntax :

```
.element {
  animation-timing-function: ease-in-out;
}
```

3.**Animation-timing-function**
Specifies the speed curve of the animation. Common values include linear, ease, ease-in, ease-out, ease-in-out, and steps.

Syntax:

```
.element {
  animation-timing-function: ease-in-out;
}
```

4.**Animation-delay**
Sets a delay before the animation starts.
Syntax :

```
.element {
  animation-delay: 1s; /* 1 second delay */
}
```

5.Animation-iteration-count

Defines the number of times the animation should repeat. Common values include a specific number or infinite for continuous looping.

Syntax :

```
.element {
  animation-iteration-count: infinite; /* Loops forever */
}
```

6.Animation-direction
Determines whether the animation should alternate direction with each iteration. Possible values are normal, reverse, alternate, and alternate-reverse.

Syntax

```
.element {
  animation-direction: alternate;
}
```

Animation Shorthand
animation : myName 2s linear 3s infinite normal

Percentage in Animation Keyframes
In CSS animations, percentages in @keyframes define specific points in time during the animation sequence.
% in Animation

```
@keyframe myName {
0% { font-size : 20px; }
50% { font-size : 30px; }
100% { font-size : 40px; }
}
```

Animation Example:

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>CSS Animation Example</title>

    <style>

        body {

            font-family: Arial, sans-serif;
```

```
        background-color: #f8f9fa;
        display: flex;
        justify-content: center;
        align-items: center;
        height: 100vh;
        margin: 0;
    }

    .box {
        width: 100px;
        height: 100px;
        background-color: #007bff;
        border-radius: 10px;
        animation-name: moveAndFade;          /* Specifies the
                            name of the keyframes animation */
        animation-duration: 4s;               /* Duration of the
                                                 animation */
        animation-timing-function: ease-in-out; /* Animation speed
                                                    curve */
        animation-delay: 1s;                  /* Delay before
                                       animation starts */
        animation-iteration-count: infinite;  /* Number of times
                                      the animation runs */
    }

    @keyframes moveAndFade {
        0% {
```

```
                transform: translateX(0);
                opacity: 1;
            }
            50% {
                transform: translateX(200px);
                opacity: 0.5;
            }
            100% {
                transform: translateX(0);
                opacity: 1;
            }
        }
    </style>
</head>
<body>
    <div class="box"></div>
</body>
</html>
```

JavaScript

What is JavaScript?

JavaScript is a powerful programming language used to create dynamic content on webpages. It can be included in HTML files either directly using <script> tags or by linking to external JavaScript files. It is a client-side scripting language, meaning these programming languages can be executed directly without any pre-compilation.

Creating and using JavaScript files:

Creating and using JavaScript files is essential for organizing and reusing code in web development. Here's a detailed look at how to create JavaScript files and include them in HTML documents with examples of three types of script inclusion:

1. **Internal JavaScript**

   Internal JavaScript is written directly within the HTML file using the <script> tag. This method is useful for small scripts or when you want to keep the HTML and JavaScript closely linked.

```
<head>
   <script>
       function showMessage() {
           alert("Hello from internal JS!");
       }
   </script>
</head>
```

2. **External JavaScript**

   External JavaScript is written in a separate file with a .js extension. This method is beneficial for maintaining a clean and modular code structure.

   - Create a file named script.js and add your JavaScript code.
   - Include this external file in your HTML document using the <script> tag with the src attribute.

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

3. **Deferred JavaScript**

   Deferred JavaScript ensures that the script is executed after the HTML document has been completely parsed. This can improve page load performance.

   - Create a file named deferred-script.js and add your JavaScript code.

- Include this external file in your HTML document using the <script> tag with the defer attribute.

By using these three types of script inclusion, you can manage your JavaScript code effectively, ensuring both readability and performance in your web projects.

Logging to Console

Use console.log(), a widely used tool for debugging and tracking the flow of a program by printing out values.

```
console.log("Hello, World!"); //Output: Hello, World!
let name = "React";
console.log("Name:", name); //Output: Name: React
```

The console object in JavaScript provides access to the browser's debugging console. It is essentially used for debugging purposes to display messages, variables, and other information during the execution of code.

Methods:

- **console.log()**: This method prints or logs any message
  Example:

```
console.log("Result:", add(5, 3));
```

- **console.error()**: This JavaScript method is used to output an error message to the console
  Example:

```
const user = null;
if (!user) {
 console.warn("Warning: User data is not available.");
}
```

- **console.clear()**: This method clears the entire console
  Example:

```
console.clear();
```

Understanding Scope

- **Global Scope:** Applies to the entire program
- **Local Scope:** On a functional level
- **Block Scope:** On a control structure level

Example:

```
let globalVar = 'Global scope';
function sampleFunction() {
   let localVar = 'Local scope';
   if (condition) {
       let blockVar = 'Block scope';
   }
}
```

Declarations:

var- Declares a variable, optionally initializing it to a value.This syntax can be used to declare both local and global variables.

- Example:

```
var color = "White";
```

let- Declares a block-scoped, local variable, optionally initializing it to a value.

const-Declares a block-scoped, read-only named constant.

- Example:

```
let num1 = 10;
```

This syntax of let and const can be used to declare a block-scope local variable.

const cannot be redeclared.

Example:

```
const Student= { name: "MAX" , rollno: 102 };
```

Operators in JavaScript

JavaScript supports various types of operators to perform operations on variables and values. This documentation covers the four main categories of operators in JavaScript.

Arithmetic Operators

- Used for performing mathematical calculations.
- Includes addition (+), subtraction (-), multiplication (*), division (/), and modulo (%).

Assignment Operators

- Used to assign values to variables.
- Includes simple assignment (=), addition assignment (+=), subtraction assignment (-=),
- multiplication assignment (*=), and division assignment (/=).

Comparison Operators

- Used to compare values and return a boolean result (true or false).
- Includes equal to (==), strictly equal to (===), not equal to (!=), strictly not equal to (!==),
- greater than (>), less than (<), greater than or equal to (>=), and less than or equal to
- (<=).

Logical Operators

- Used to combine or modify conditions.
- Includes logical AND (&&), logical OR (||), and logical NOT (!).

Example usage:

```
let a = 5;
let b = 10;
let sum = a + b;
let isEqual = (a === b);
```

Ternary Operators:

- A shorthand way to write an if-else statement.
- Checks if age is greater than or equal to 18.
- If true, returns Name1 .
- If false, returns Name2 .

```
let age = 20;
const Component() {
 return age >= 18 ? <div> Name1 </div> : <div> Name2 </div>;
}
```

Spread Operator:

- Used to expand an array into individual elements.
- Creates a new array moreNumbers by combining the elements of numbers with additional values.
- The ...numbers part spreads the elements of numbers into the new array.

```
let numbers = [1, 2, 3];
let moreNumbers = [...numbers, 4, 5, 6];
console.log(moreNumbers); // Output: [1, 2, 3, 4, 5, 6]
```

Data Types:

Primitive:

The primitive types like string, number, boolean, null, undefined, and symbol.

```
typeof("Hola") //string
typeof(12) // integer
typeof(true) // boolean
typeof(undefined) // undefined
typeof(null) // object
typeof({}) // object
typeof(Symbol()) // symbol
typeof(BigInt(12345678901234567890)) // bigint
typeof(function(){}) // function
```

Non Primitive:

Arrays:

- Arrays are a special type of object used to store ordered collections of values of the same datatype. They are generally declared using the 'const' keyword.

```
let emptyArray = []; // An empty array
let fruits = ["apple", "banana", "cherry"]; // An array with string elements
```

Common Methods:

- length():Returns the number of elements in the array
- push(): Appends elements to the end of the array
- pop(): Removes the last element from the array.
- map(): Creates a new array with the results of calling a provided function on every element.
- forEach(): Iterates through the array using .forEach(function(element,index,array){});
- shift(): Removes the first element from the array

Objects: Collections of key-value pairs of same or different data types including arrays and functions.

```
const person = {
   name: 'Alice',
   age: 25,
   greet() {
     return `Hi, I'm ${this.name}`;
   }
```

Object Dereferencing:

```
const person = { name: "Alice", age: 25 };
const { name, age } = person;
console.log(name); // Output: "Alice"
```

If-Else Statements:

Use the if statement to execute a statement if a logical condition is true. Use the optional else clause to execute a statement if the condition is false.

Syntax:

```
if (condition) {
  statement1;
} else {
  statement2;
}
```

You can also compound the statements using else if to have multiple conditions tested in sequence, as follows:

```
if (condition1) {
  statement1;
} else if (condition2) {
  statement2;
} else if (conditionN) {
  statementN;
} else {
  statementLast;
}
```

Looping Statements in JavaScript

Looping statements in JavaScript allow you to execute a block of code repeatedly. This section covers the different types of loops available and their use cases.

1.For Loop

Syntax:

```
for (initialization; condition; increment) {
  // Code to be executed
}
```

Example:

```
for (let i = 0; i < 5; i++) {
  console.log(i); // Output: 0 1 2 3 4
}
```

2. while Loop

Syntax:

```
while (condition) {
  // Code to be executed
}
```

Example:

```
let count = 0;
while (count < 5) {
  console.log(count); // Output: 0 1 2 3 4
  count++;
}
```

3. do...while Loop

Syntax:

```
do {
  // Code to be executed
```

```
} while (condition);
```

Example:

```
let count = 0;
do {
  console.log(count); // Output: 0
  count++;
} while (count < 5);
```

4. for...in Loop

Syntax:

```
for (key in object) {
  // Code to be executed
}
```

Example:

```
const person = { name: "Alice", age: 30 };
for (let key in person) {
  console.log(key, person[key]);} // Output: name Alice, age 30
```

5. for...of Loop

Syntax:

```
for (value of iterable) {
  // Code to be executed
}
```

Example:

```
const fruits = ["apple", "banana", "cherry"];
for (let fruit of fruits) {
  console.log(fruit); // Output: apple banana cherry
}
```

Note: The `for...of` loop is generally preferred over `for...in` for iterating over arrays.

Functions in JavaScript

- **Introduction to Functions:**
  - Functions are blocks of code designed to perform a specific task.
  - Functions promote code reusability and modularity in JavaScript.
- **Function Declaration:**
  - Functions are declared using the function keyword, followed by a name and parentheses for parameters.

Example:

```
function greet(name) {
    return 'Hello, ' + name + '!';
}
let message = greet('John');
console.log(message)
Output: Hello John!
```

Arrow function: An arrow function expression is a compact alternative to a traditional function expression.

Syntax: (parm1, parm2, ...) => expression

Example:

```
const countChar = (color) => {
    const count = color.length; // Use the length property to get the number of
characters
    return count;
}
```

Events:

Events are code structures that listen for activity in the browser, and a function is invoked to run code in response.for example, clicking a button, submitting a form, a loaded web page.

- **onClick**: Triggered when an element is clicked.

Example:

```
  <p>Welcome to COSC's BootCamp!</p>
 <p id="message">Hello</p>
    <button onclick="handleClick()">Click Me</button>
    <script>
        function handleClick() {
            document.getElementById('message').textContent = 'Button was clicked!';
        }
    </script>
```

or

```
document.getElementById('myButton').addEventListener('click', () => {
    console.log('Button clicked!');
  });
```

- **onChange**: Triggered when the value of an input, textarea, or select element change
- **onSubmit**: Triggered when a form is submitted

Async/Await

What are Async Functions? Async functions are declared with the async keyword. They always return a Promise and make asynchronous code easier to write and read.

What is the Await Expression? The await expression is used inside async functions. It pauses the execution of the async function until the Promise is resolved and returns its result, making asynchronous code appear synchronous.

```
async function fetchData() {
  try {
    const response = await fetch('https://randomuser.me');
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    const data = await response.json();
    return data;
  } catch (error) {
    console.error('Error fetching data:', error);
    throw error; // Re-throw the error for handling in the calling code
  }
}
```

What is a Promise?

A Promise is an object representing the eventual completion (or failure) of an asynchronous operation and its resulting value.

States of a Promise:

1. Pending: Initial state, neither fulfilled nor rejected.
2. Fulfilled: Operation completed successfully.
3. Rejected: Operation failed.

```
const promise = new Promise((resolve, reject) => {
  // Simulate an asynchronous operation
  setTimeout(() => {
    const data = { message: 'Success!' };
    resolve(data); // Resolve the promise with data
  }, 1000);
});
```

Node JS

Ever wanted to run JavaScript outside of your browser, perhaps in your shell, your own environment, or even on your own server? That's where Node.js comes in. It's a JavaScript runtime environment that allows you to execute JavaScript code independently. To get started, install Node.js on your machine:

MacOS/Linux:

1. **Package**:
    a. Visit NodeJS and download the LTS(preferred) version package.
    b. Extract the package
    c. Move the extracted directory to /usr/local/
    d. sudo mv <extracted-package-name> /usr/local/node

2. **Terminal:**
    a. Install homebrew

        /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

    b. brew install node

Windows:

Installer: Visit NodeJS and download the installer and make sure to check add node to your path option.

Check Installation:

Run node -v to check the Node.js version.

Run npm -v to check the npm (Node Package Manager) version.

ReactJS

What is React?

React is a JavaScript library used to build user interfaces. When a web page is created using react, we call it a "React App". It allows developers to create large web applications that can update and render efficiently in response to data changes.

Why's React so popular?

React makes it painless to create interactive UIs. Design simple views for each state in your application and React will efficiently update and render just the right components when your data changes.

Example:

For rendering something on a webpage, by using vanilla JavaScript by creating and appending an h1 to our the code would look like

```
const h1 = document.createElement("h1")
h1.textContent = "This is an imperative way to program"
h1.className = "header"
document.getElementById("root").append(h1)
```

With React the entire code is built in the JavaScript file and for an interactive page with many more elements the code would get a lot lengthier. Instead, it can be relayed on react to figure out how to convert declaratively written html code into JavaScript code and append it to the Dom by writing the exact same code as below

```
ReactDOM.render(<h1>Hello, React!</h1>,
document.getElementById("root"))
```

React is component-based: A component in React is a reusable piece of code that represents a part of your user interface. They act as building blocks that you can use for building complex UIs. With React components the code becomes much more flexible and structured.

For example, in the below code, each component (MyAwesomeNavbar, MainContent, MyAwesomeFooter) encapsulates its own structure and behavior where the entire webpage is composed of three custom components.

```
<body>
    <MyAwesomeNavbar />
    <MainContent />
    <MyAwesomeFooter />
</body>
```

Virtual DOM

React uses a Virtual DOM to optimize rendering

But what is DOM?

The DOM is a tree-like structure representing the HTML elements of a webpage. Whenever a change occurs, the browser must update this structure, leading to performance issues.

This process involves recalculating styles, rearranging elements, and repainting the screen, which can be slow, especially for complex UIs.

To address this, React introduced the Virtual DOM.

React employs a Virtual DOM to optimize performance. Instead of directly manipulating the actual DOM, React creates a lightweight, inmemory copy. When changes occur, React updates this virtual

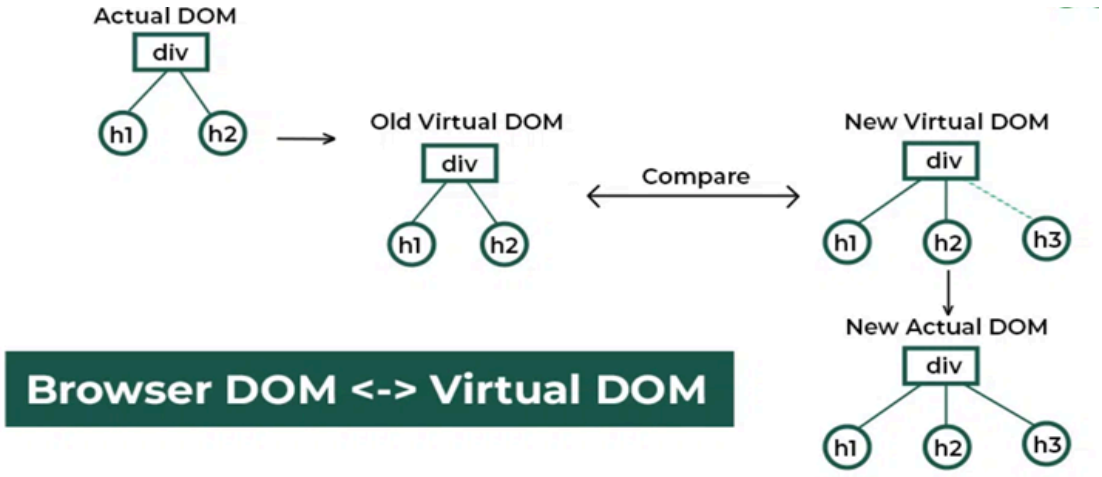
representation and then efficiently determines the minimal changes needed to synchronize with the real DOM.

This approach avoids unnecessary DOM manipulations, resulting in faster and smoother updates.

Changes to the Virtual DOM are fast because they are performed in memory and do not involve any direct browser operations

How the Virtual DOM Works

1. **Initial Render:** When a React component is first rendered, a Virtual DOM tree is created, representing the structure of the UI.
2. **State/Props Update:** When the state or props of a component change, a new Virtual DOM tree is created, representing the updated UI.
3. **Diffing:** React compares the new Virtual DOM tree with the previous one to identify what has changed. This process is called "diffing."
4. **Reconciliation:** Based on the differences found, React updates the actual DOM with the minimal set of changes required, ensuring efficient updates and rendering.



-> React has a vast ecosystem of libraries, tools, and extensions and has Developer Tools, available as browser extensions that can be used to enhance and extend its capabilities.

Installation

To start React installation, it is required to install Node first because React.js is a JavaScript library, and Node.js is a JavaScript runtime environment that allows you to run JavaScript on the server side.

To install Node, navigate to the Node.js website , download the current version and install it on the device. Now, type in the below code to check the version and confirm the installation

node -v

```sh
Microsoft Windows [Version 10.0.22621.3007]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Alex>node -v
v14.17.0

C:\Users\Alex>
```

Installing React using Vite:

In the command prompt window, navigate to the directory that you want to use in creating your React project. To do this, type the below then click enter.

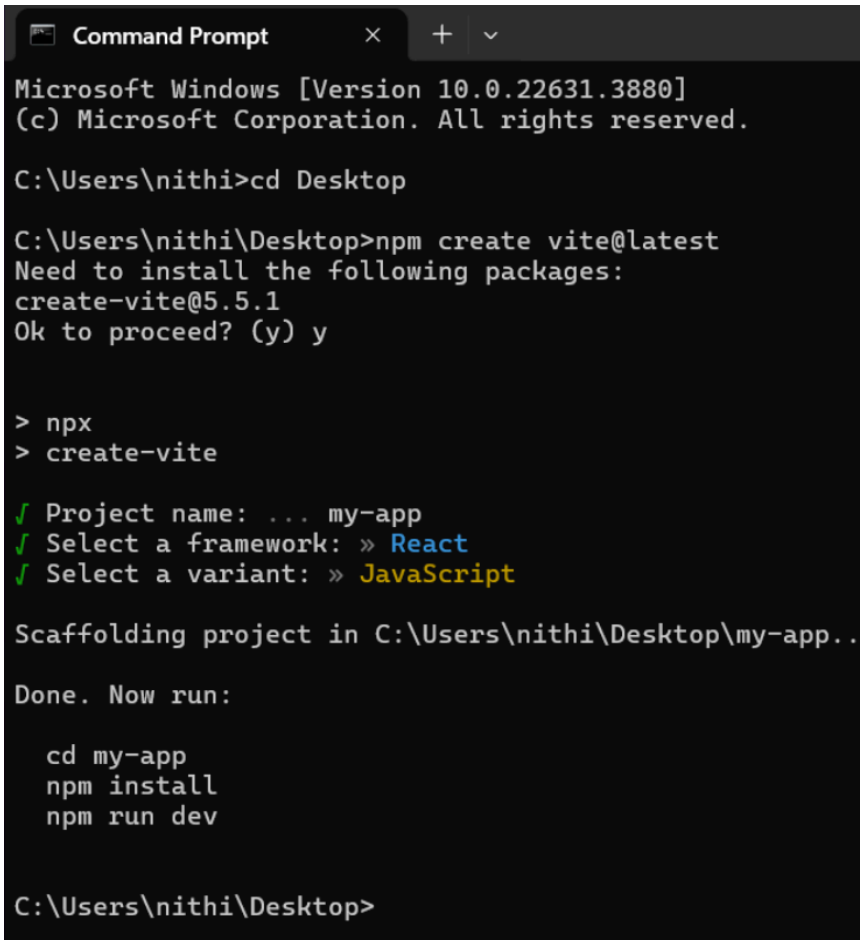
cd Documents

Type *mkdir [folder name]* then navigate to the newly created directory using

cd [folder name].

Now type in the below code to create the react app followed by the desired App Name. (Here my-app)

npm create vite@latest

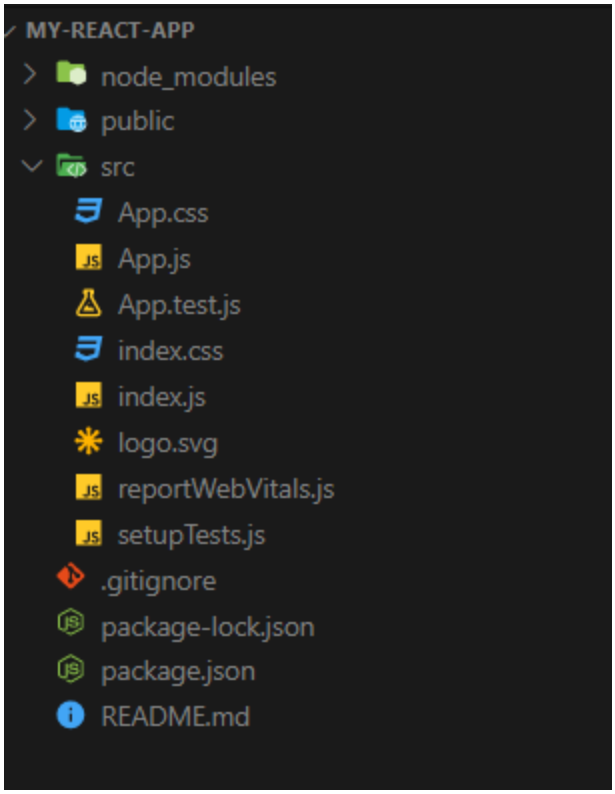


The installation should now have been successfully completed. In order to navigate to the source code editor and take a look at the required folders and files created, type in the below command on the same cmd

CODE:

```
C:\Users\Alex\Documents\my-folder\my-react-app>code .
```

The folder structure of React App should look as follows:

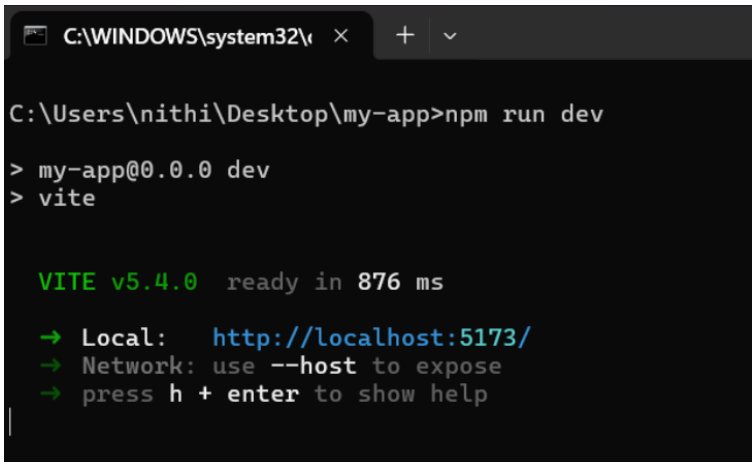


Running a React App

Once the installation is completed, type in the below commands to navigate to and start the react app:

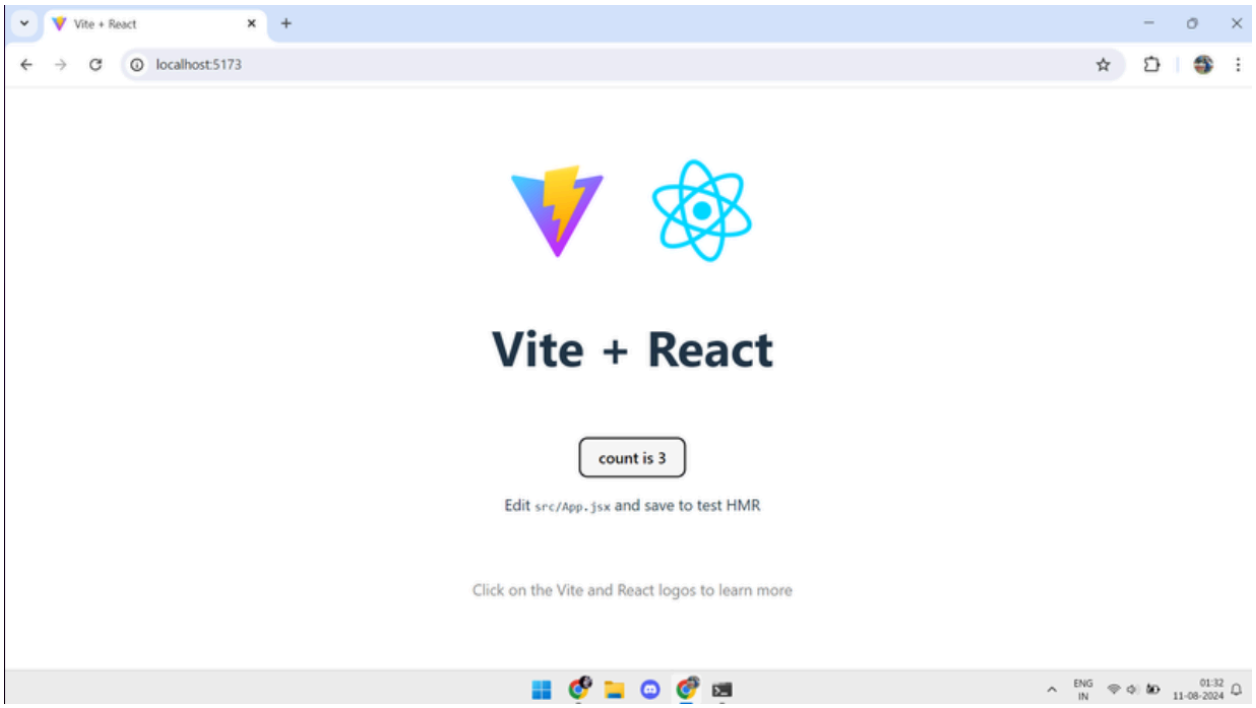
cd my-app

npm run dev



Click on the localhost link to open the react app.

OUTPUT:



JSX:

JSX (JavaScript XML) is a syntax extension for JavaScript that looks similar to HTML, which makes writing React components more intuitive and easier. JSX enables developers to embed HTML-like code directly within JavaScript, allowing for a more declarative way to create UI elements. JSX gets transpiled into React.createElement calls, which create the elements that React uses to build the DOM

Syntax:

```
const element = <h1>Hello, world!</h1>;
```

Implementation:

```
import React from 'react';
const App = () => {
 return <h1>Hello, world!</h1>;
};
export default App;
```

Explanation:

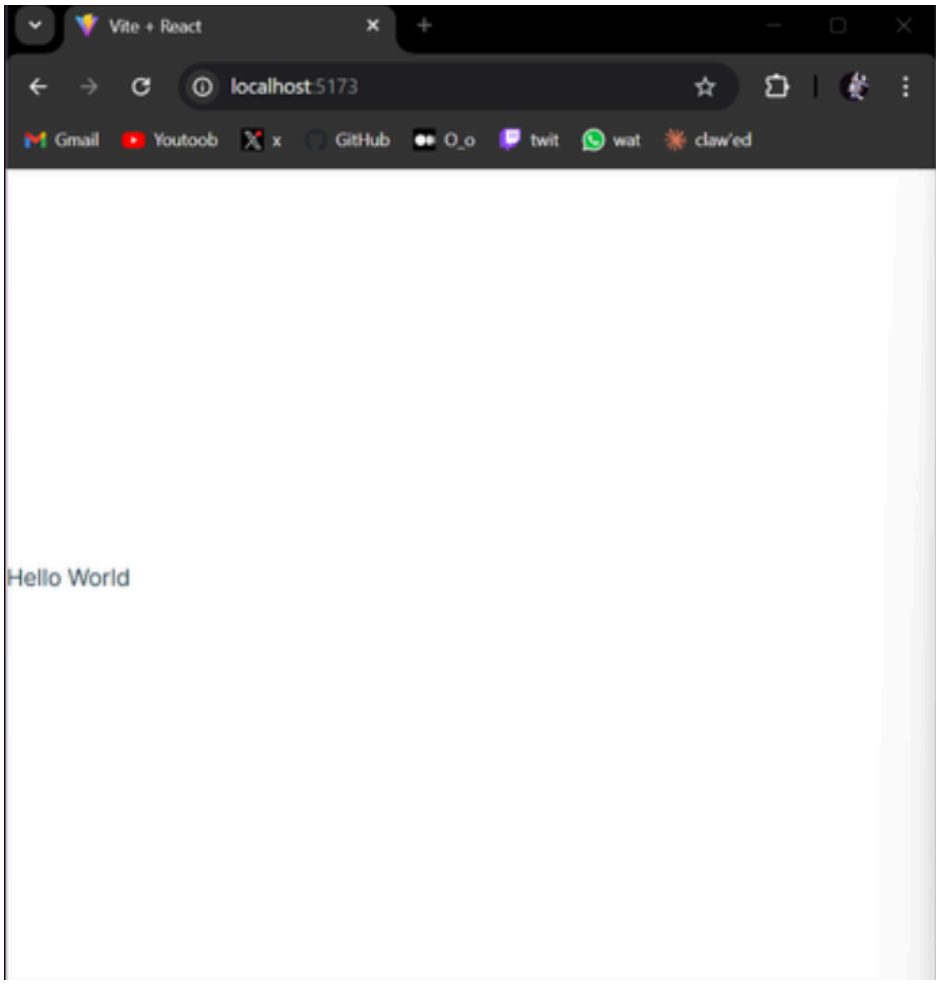
This code defines a functional component App that renders an <h1> heading element with the text "Hello, world!" When this component is rendered, it will display a large, bold "Hello, world!" title on the webpage.

React Arrow Function

```
import React from 'react';

const ComponentName = () => {
  return (
       <div>
       {/* JSX goes here */}
       </div>
  );
};

export default ComponentName;
```

Code - App.jsx

```
function App() {
  return (
       <>
       <p>Hello World</p>
       </>
  );
}
export default App;
```


Vite + React
localhost:5173
Gmail
Youtoob
x
GitHub
O_o
twit
wat
claw'ed
Hello World

Components and Props

Create a components folder in the app directory.

- Now create the files which contains the components.
- Insert the components wherever you want to use them in your project.
- Props can be used to send data and event handlers from child component to the parent components.
- Components are called with self closing tags with component name

Header.jsx

```
import React from 'react';

const Header = () => {
  return (
       <>
       <p>Header</p>
       </>
  );
};

export default Header;
```

ComponentOne.jsx

```
import React from 'react';

const ComponentOne= () => {
  return (
       <>
       <p>ComponentOne</p>
       </>
  );
};

export default ComponentOne;
```

Footer.jsx

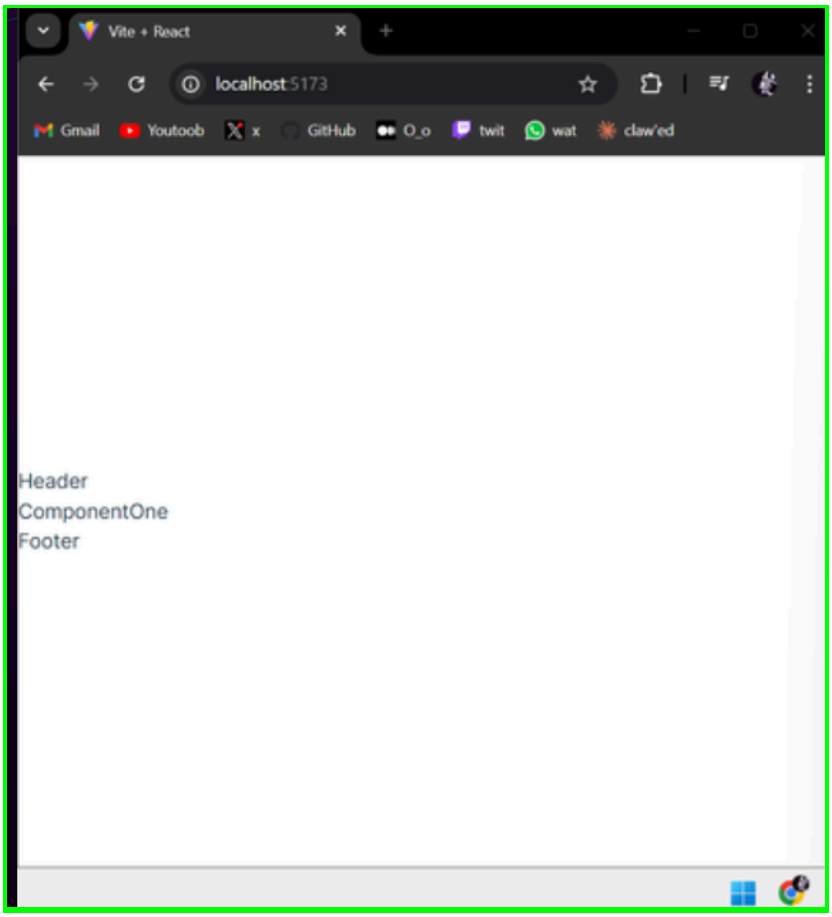
```
import React from 'react';

const Footer= () => {
  return (
       <>
       <p>Footer</p>
       </>
```

```
  );
};

export default Footer;
```

App.jsx

```
import Header from "./components/Header";
import ComponentOne from "./components/ComponentOne";
import Footer from "./components/Footer";
function App() {
    return (
        <>
            <Header />
            <ComponentOne />
            <Footer />
        </>
    );
}
export default App;
```

OUTPUT:



SENDING PROPS TO THE COMPONENT

SendingProps.jsx

```
const SendingProps = ({ name }) => {
    return (
        <div>Color is {name}</div>
    );
}

export default SendingProps;
```

App.jsx

```
 const SendingProps = ({ name }) => {
    return (
        <div>Color is {name}</div>
    );
}

export default SendingProps;
```

HOOKS (useState and useEffect)

1. Hooks, introduced in React 16.8, allow functional components to manage state and side effects.

2. The useState hook lets you add and manage state in functional components without class components.

3. State can be initialized with a default value, and you get the current state along with a function to update it.

4. The useEffect hook manages side effects such as data fetching, event subscriptions, and DOM manipulations in functional components.

5. You can specify dependencies in the useEffect hook to control when the effect should re-run

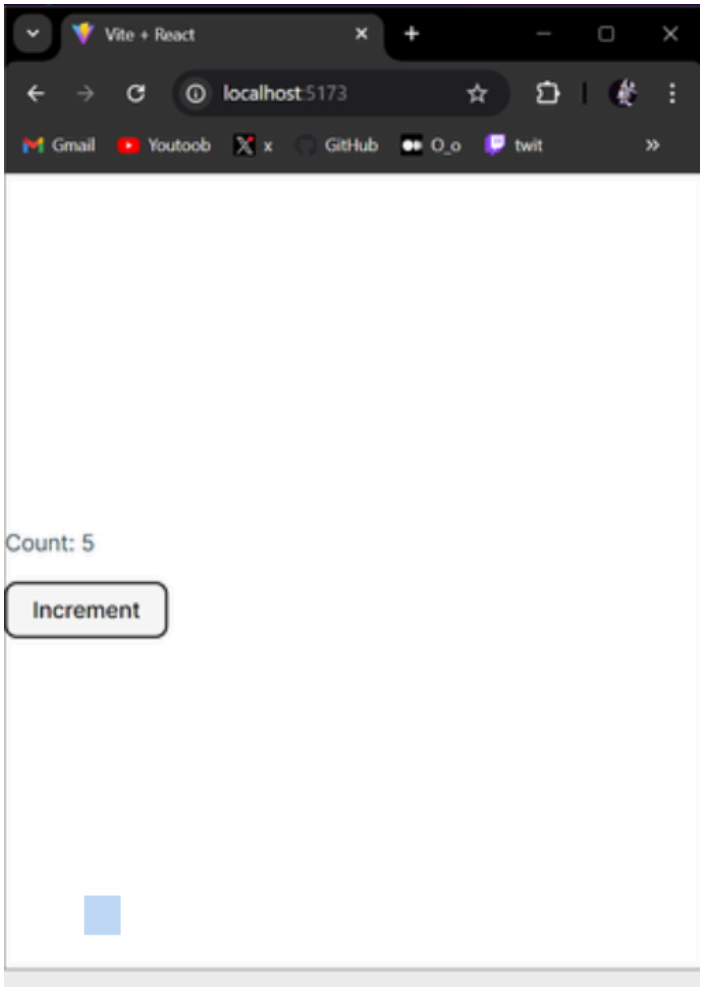
```
#Syntax for useState:
const [state, setState] = useState(initialState);
#Syntax for useEffect:
useEffect(() => { // Side effect code here }, [dependencies]);
```

App.jsx (useState)

```
import { useState } from "react";

const App = () => {
  const [count, setCount] = useState(0);
```

```
  return (
       <div>
       <p>Count: {count}</p>
       <button onClick={() => setCount(count + 1)}>
       Increment
       </button>
       </div>
  );
};

export default App;
```

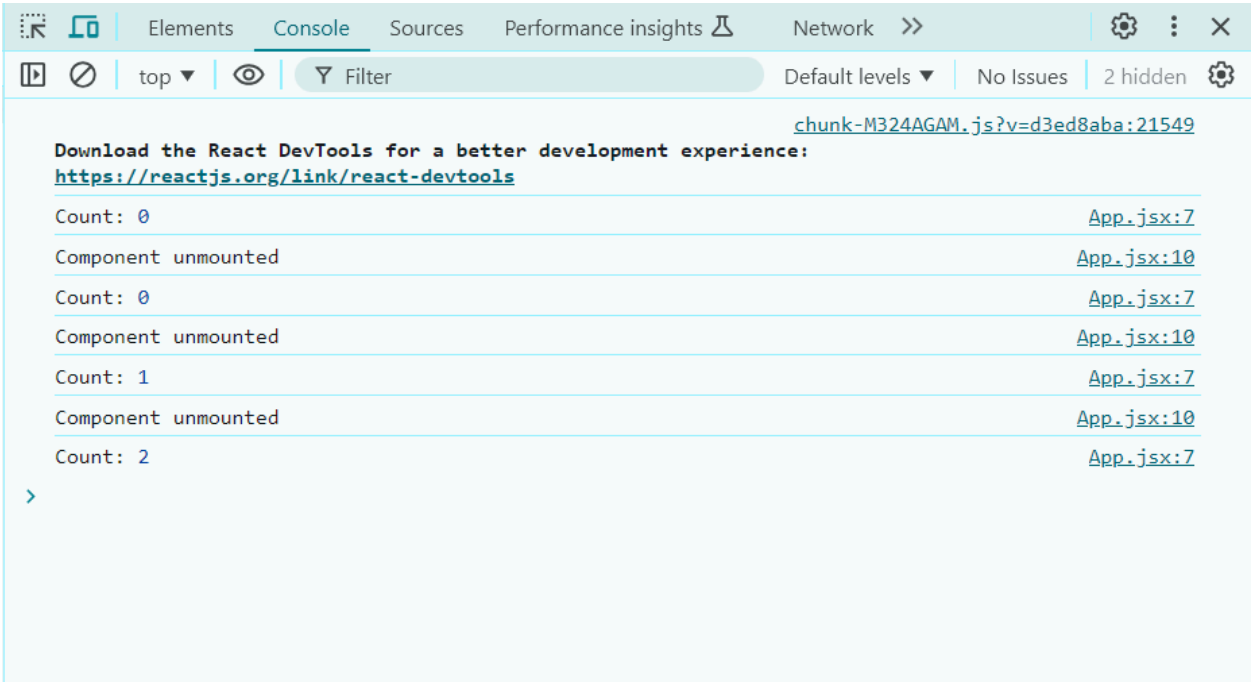
OUTPUT:



App.jsx(useEffect)

```
import { useState, useEffect } from "react";

const App = () => {
  const [count, setCount] = useState(0);

  useEffect(() => {
       console.log('Count:', count);

       return () => {
       console.log('Component unmounted');
       };
  }, [count]);

  return (
       <div>
       <p>Count: {count}</p>
       <button onClick={() => setCount(count + 1)}>Increment</button>
       </div>
  );
};

export default App;
```

OUTPUT:



CONDITIONAL RENDERING

1. Conditional rendering in React lets you display different UI elements based on specific conditions.

2. It is useful for dynamically displaying elements depending on the application's state or user interactions.

3. JavaScript expressions, such as the ternary operator, logical AND (&&), or helper functions, can be used to determine what to render.

4. This approach improves user experience by showing relevant content based on the current state.

5. Examples include displaying a login button when the user is logged out or a welcome message when logged in.

Syntax:

```
{condition ? <ComponentA /> : <ComponentB />}
```

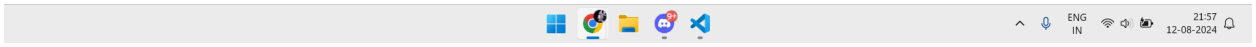
Implementation:

```
import React, { useState } from 'react';

import { useState } from "react";

 const App = () =>
 {
  const [isLoggedIn,setIsLoggedIn] = useState(false)
  return(
    <div>
      {isLoggedIn ? <h1>Welcome!</h1> : <h1>Please Log In</h1>}
      <button onClick={()=> setIsLoggedIn(!isLoggedIn)}>
        Toggle Login
      </button>
    </div>
  )
 }


 export default App
```

OUTPUT:

Welcome!

Toggle Login



USER INPUT FORMS

1. React forms allow you to collect and manage user input efficiently by controlling the form elements through state.

2. Controlled components in React use state to manage the value of form elements like inputs, text areas, and selects.

3. Handling form submission in React involves capturing the form data, often through an event handler like `onSubmit`.

4. Validation logic can be implemented to check the user input before form submission, ensuring data integrity.

5. Forms in React can be styled and structured dynamically, allowing for a customizable user input experience.

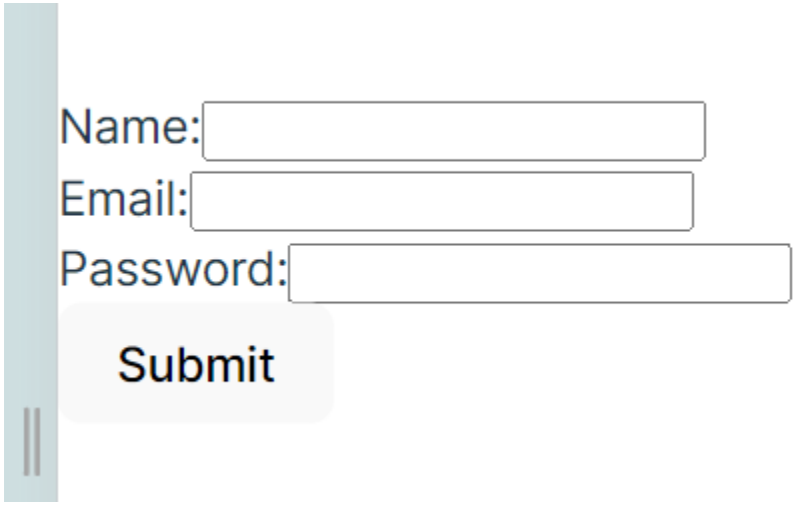
UserForm.jsx

```
import { useState } from 'react';

function UserForm() {
	const [name, setName] = useState('');
	const [email, setEmail] = useState('');
	const [password, setPassword] = useState('');

	const handleSubmit = (event) => {
	event.preventDefault();
	console.log('Form submitted:', { name, email, password });
	};

	return (
	<form onSubmit={handleSubmit}>
		<label>
		Name:
		<input
			type="text"
			value={name}
			onChange={(event) => setName(event.target.value)}
		/>
		</label>
		<br />
		<label>
		Email:
		<input
			type="email"
			value={email}
			onChange={(event) => setEmail(event.target.value)}
		/>
		</label>
		<br />
		<label>
		Password:
		<input
			type="password"
			value={password}
			onChange={(event) => setPassword(event.target.value)}
		/>
		</label>
		<br />
		<button type="submit">Submit</button>
	</form>
	);
}
```

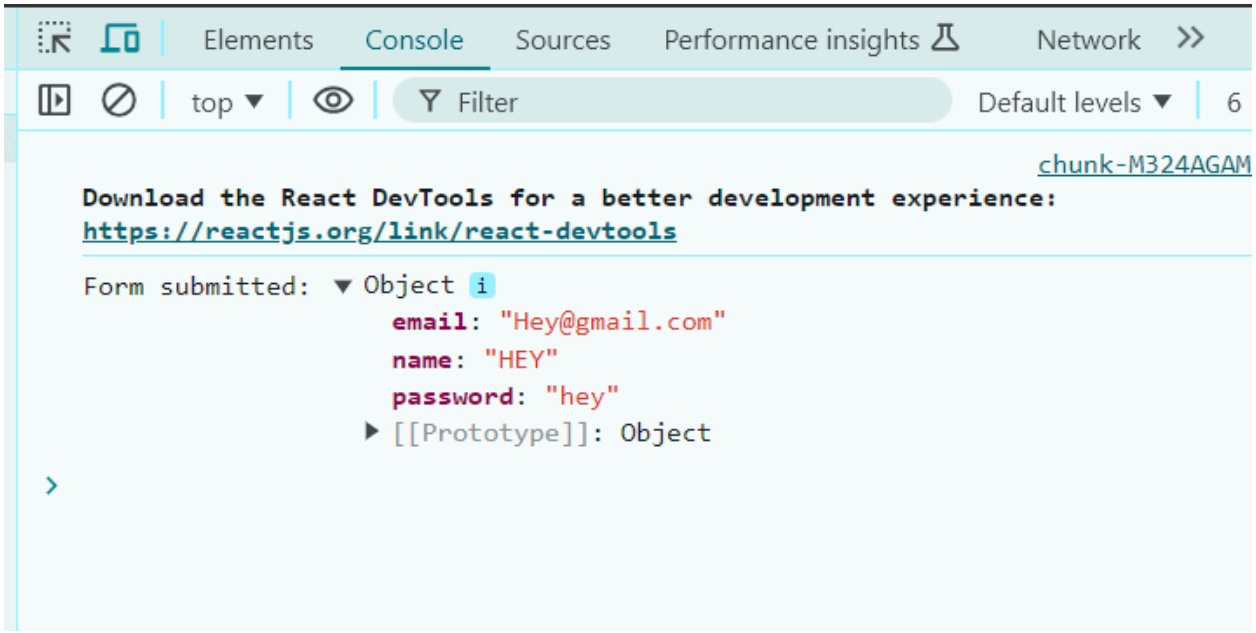
```
export default UserForm;
```

OUTPUT :

Form Data:



Console:



React Router

1. React Router DOM is a library for handling routing in React applications,enabling navigation between components or pages
2. It allows developers to define routes within their application and link to these routes from components.
3. The library uses a declarative approach, making route management straightforward.
4. Routes are defined using specific components provided by React Router DOM.
5. React Router DOM automatically renders the appropriate component when the URL matches a defined route's path.

Home.jsx

```
function Home() {
	return (
	<div>
		<h1>Home Page</h1>
		<p>Welcome to the Home page!</p>
	</div>
	);
}

export default Home;
```

About.jsx

```
function About() {
	return (
	<div>
		<h1>About Page</h1>
		<p>This is the About page.</p>
	</div>
	);
}

export default About;
```

About.jsx

```
function App() {
	return (
	<Router>
		<div>
		<nav>
			<ul>
				<li>
				<Link to="/">Home</Link>
```

```
                        </li>
                        <li>
                        <Link to="/about">About</Link>
                        </li>
                  </ul>
            </nav>
            <Routes>
                  <Route path="/" element={<Home />} />
                  <Route path="/about" element={<About />} />
            </Routes>
            </div>
      </Router>
      );
}

export default App;
```

OUTPUT :

localhost:5173/about

- Home
- About

The is About Page

APIs

What is an API?

APIs are mechanisms that enable two software components to communicate with each other using a set of definitions and protocols.

API stands for Application Programming Interface. In the context of APIs, the word Application refers to any software with a distinct function. Interface can be thought of as a contract of service between two applications. This contract defines how the two communicate with each other using requests and responses. Their API documentation contains information on how developers are to structure those requests and responses.

Fetching data from random APIs

Here, we will be using https://randomuser.me/ to implement this. This is a free and open source API for generating random user data.

How to use:

You can use AJAX to call the Random User Generator API and will receive a randomly generated user in return. We will do this using the fetch() method:

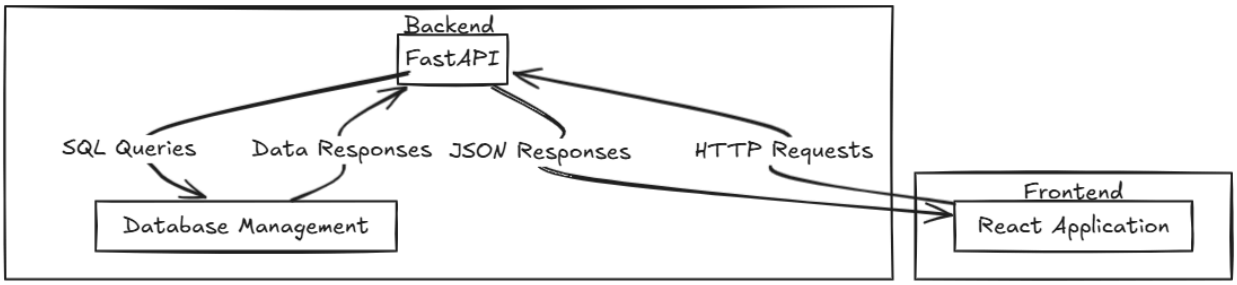
Code:

```
const apiUrl = 'https://randomuser.me/';
// Make a GET request
fetch(apiUrl)
    .then(response => {
        if (!response.ok) {
            throw new Error('Network response was not ok');
        }
        return response.json();
    })
    .then(data => {
        console.log(data);
    })
    .catch(error => {
        console.error('Error:', error);
    });
```

The need for a backend

- A backend is crucial in web development as it serves as the backbone for managing, processing, and storing data securely.
- It handles business logic, user authentication, database interactions, and API integrations, ensuring smooth and efficient data flow between the frontend and the server.
- Additionally, a backend provides the necessary infrastructure to scale applications, manage resources, and deliver a consistent user experience across various platforms and devices.
- By separating frontend and backend concerns, developers can build more robust, maintainable, and scalable applications.

Overview of how it works



We run a server on the backend which is managed by FastAPI and expose API endpoints to which our frontend application (React) connects to.

Backend Server Basics

- A backend server is the core component of a web application that processes client requests, performs necessary computations, interacts with databases, and returns the appropriate responses.
- It serves as the intermediary between the user-facing frontend and the server-side databases or external services, ensuring secure, reliable, and efficient communication.
- A server is a centralized computer program that provides resources, services, or data to other computers, known as clients, over a network.
- It plays a vital role in hosting websites, managing databases, processing client requests, and ensuring secure communication.
- Servers can be specialized for different tasks, such as web servers for hosting websites, database servers for managing data, hardware accelerated servers for computationally intensive tasks like ML and application servers for running software applications.
- They operate continuously to ensure high availability and reliability, forming the backbone of internet and enterprise network infrastructure.

Types of APIs

SOAP APIs

- SOAP stands for Simple Objects Access Protocol.
- Client and server exchange messages using HTTP for transport and XML for data specification.
- It can only transport XML data and is tied to the HTTP(S) protocol, so it is not flexible.
- It's heavy as there is a large amount of mandatory metadata tied to every request and as a result human readability of a SOAP request/response suffers.

RPC APIs

- RPC stands for remote procedure call, clients call predefined procedures on the server using a message.
- Parameters are packed into the message and sent to the server by the OS.
- The whole process repeats as the server sends its response.

Websocket APIs

- WebSocket uses TCP to establish a full duplex connection between client and server app.
- WebSocket offers high reliability and it is very lightweight as they generally are written at a very low level.
- This is highly useful for apps that required realtime streaming.

REST APIs

- REST is less of a protocol and more of an API architectural guideline.
- REST stands for representational state transfer.
- It emphasizes resources such as JSON, HTML or XML data and a bunch of hypermedia links that can be followed to change the state of the system.
- A client must know at least one identifier for a type of resources that it looks for such as the identifier of at least one note in a notes app, using which it can acquire the identifiers of all notes.

GraphQL

- GraphQL is more of a data query and manipulation language than an API, but it is used commonly in conjunction with WebSocket.
- It makes it very simple to prevent under and over fetching commonly seen in REST APIs.
- It only has Query and Mutation requests.
- Suppose you only need a part of the data, you can define GraphQL endpoints such that each request only fetches the information that is absolutely necessary.

Webhooks

- They work like hooks in React, acting when any event occurs either in the frontend or the backend.
- They run asynchronously, this makes them ideal for automatic tasks that have to occur in the background.

Rest APIs

- REST stands for Representational State Transfer.

- REST defines a set of functions like GET, PUT, DELETE, etc. that clients can use to access server data.
- Clients and servers exchange data using HTTP.
- The main feature of REST API is **statelessness**. Statelessness means that servers do not save client data between requests.
- Client requests to the server are similar to URLs you type in your browser to visit a website.
- The response from the server is plain data, without the typical graphical rendering of a web page.

STATUS CODES & HEADERS

When an API responds to a request, it usually contains 3 main components:
The status code: It is a three-digit number sent by the server to indicate the result of the request. Common status codes include:

- **2xx OK:** They are used to indicate success of requested operation
- **3xx Redirection:** They are used to indicate that requested resource has been moved
- **4xx Client Errors:** The request was invalid or malformed.
- **5xx Server Errors:** Errors occurred at the server side while processing the request.

Headers: These provide additional information about the response. They include metadata such as content type, length, server information, etd. Examples of headers include:

- **Content-Type:** Indicates the media type of the response body (e.g., application/json, text/html).
- **Content-Length:** Specifies the size of the response body in bytes.

BODY

The response contains the actual data sent back from the server. Depending on the request and the response, the body can include:

- **Data:** For successful GET requests, this could be the requested resource (e.g., a JSON object, file, etc).
- **Confirmation Messages:** For POST or PUT requests, this might include a confirmation message or the details of the created/updated resource.
- **Error Details:** In the case of errors, the body might contain a description of what went wrong or instructions on how to correct it.

API RESPONSES

```
HTTP/1.1 200 OK //status code
Content-Type: application/json //header
Content-Length: 123 //header
//Body
{ "id": 1,
"name": "COSC",
"email":"COSC@gmail.com"
```

```
}
```

OpenAPI Standards and Specification in FastAPI

- FastAPI is built on the OpenAPI standard, which is a widely used specification for defining RESTful APIs.
- OpenAPI provides a structured way to describe your API's endpoints, request and response formats, authentication methods, and more.
- This standardization ensures that your API is easily understandable and usable by other developers and tools.
- In FastAPI, the OpenAPI specification is automatically generated based on your defined routes, models, and parameters.
- This automatic generation not only saves time but also ensures that your API documentation is always up-to-date with the actual implementation, making it easier to maintain and expand your API.

Fundamentals of Python for FastAPI

Python Basics:
1.Data types
2.Variables
3.Operators and Expressions, Precedence and Associativity
4.Control flow Statements
    a.If-elif-else
    b.for and while loops
    c.Comprehensions (list and dictionary)
5.Modules and Packages
    a.Importing and exporting
6.File handling
7.Error handling
8.OOP concepts
9. Working with APIs and network requests
- The requests(now being dethroned httpx due to async support) library
- Handling JSON data with json module

10. Functions and Decorators
Type Hints / Type Annotations.

FastAPI

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python based on standard Python type hints. Important features:
1.Based on Open Standards
2.Automatic Docs
3.Modern Python
4.Editor Support.
5.Validation

6. Security and Authentication
7. Dependency Injection
8. Unlimited Plug-Ins and Community support
9. Concise

FastAPI is fully compatible with (and based on) Starlette (a lightweight Asynchronous Server Gateway Interface (ASGI) framework/toolkit, which is ideal for building async web services in Python) and Pydantic (widely used data validation library for Python)

Async Operations in FastAPI

- Asynchronous programming is a core feature in FastAPI, allowing you to handle many requests simultaneously without blocking the server.
- By defining routes with async def, FastAPI can pause operations with await, freeing the server to handle other requests in the meantime.
- This is especially useful for I/O-bound operations like database queries or external API calls, where waiting for a response could otherwise hold up your entire application.
- Using async operations ensures that your FastAPI application remains highly performant and scalable, capable of serving many users concurrently without slowdowns.

Installation

FastAPI needs Starlette and PyDantic as dependencies. It installs them if not already installed. To install FastAPI, we get it from the Python Package Index (PyPI) using pip (Python's package-management tool):

```
$ pip install fastapi
```

FastAPI basics

We write some python functions to run on each type of request for a specific route and FastAPI takes care of the rest, right from starting up a server to managing incoming requests, routing them to the right function and returning the response back to the client.

FastAPI uses decorators to make a Python Function act as an API Endpoint, it does all the heavy-lifting to manage the server and create the routing map. As FastAPI is built based on PyDantic, we will also explore a bit of validation here.

Getting Started

We will need a Python file with a FastAPI app instance and one or more route-handlers. For example, a file named server.py containing the route handlers.

Importing FastAPI and creating an application instance

We can then import the FastAPI class from the fastapi module and create an application instance using it.

```
//import the FastAPI class
from fastapi import FastAPI
app=FastAPI() //creates a new FastAPI application instance
```

FastAPI provides decorators that can be used to define our route-handlers for specific request methods and we can also pass in the route-URL as an argument to it to specify the endpoint corresponding to this function.

```
// use a decorator of the form app.<method>
@app.get("/")
def read_root():
    return{"message":"Hello, World!"}
```

The handler can return various forms of output, like strings, dictionaries, Pydantic schema objects, and more

Running the FastAPI Application with Uvicorn

To run your FastAPI application, you typically use Uvicorn, a fast ASGI server for Python. Installation To install Uvicorn, you can use pip

```
$ pip install uvicorn
```

After you've created your FastAPI application, you can run it using Uvicorn. For example, if your FastAPI app is defined in a file named main.py, you can start your server by running:

```
$ uvicorn main:app –reload
```

- main refers to the file name (without the .py extension).
- app refers to the FastAPI instance in your code.
- --reload tells Uvicorn to automatically reload the server when code changes, which is useful during development

This will spin-up a server on the localhost port 8000 (or another port if port 8000 is already in use). The routes on the server will be as per the route handlers defined in the server.py file.

We can check it by going to http://localhost:8000 on a web browser and we should see a JSON response with the object we returned from the function {"Hello": "World"}

To view the Interactive documentation for API routes, go to http://localhost:8000/docs on a web browser.

Route Parameters

Sometimes it is not possible to specify all the paths separately but it is possible to specify a pattern for the routes and in such cases, we can add parameters to our Route-URLs to create dynamic routes. There are 2 types of parameters:

Path parameters - They are a part of the endpoint-url
Example: /users/{id} represents a parameter id which can take any value.
Sample routes that match this pattern would be: /users/41, /users/103ab

Query parameters - They are added after the endpoint-url.
They follow the ?key1=value1&key2=value2 notation.
Example: /cars?color=blue&type=sedan

Example: Path and Query Parameters in FastAPI

// We can pass a path-string with {param} syntax to the decorator to add path parameter(s) as below merchant_id and item_id are Route Parameters @app.get("/items/{merchant_id}/{item_id}")
// The parameters of the function apart from the Route Parameters are used as the Query Parameters. It is best to define default values for Query Parameters as they are optional.

```
def read_item(merchant_id:int, item_id: int, q: Union[str, None] = None):
return {"item_id": item_id, "q": q}
```

When a suitable request is made to the endpoint, the handler received the Route parameters and the Query Parameters as arguments and we can use them to further process the data and generate the response.

We can pass a path-string with {param} syntax to the decorator to add path parameter(s) as below time_of_day and name are Route Parameters.

```
@app.get("/greet/{time_of_day}/{name}")
def read_path_params(time_of_day: str, name: str);
    return {"message":f"Good {time_of_day}, {name}!"}
```

```
http://127.0.0.1:8000/greet/morning/SaiKiran
```

Example: Query Parameters in FastAPI

When you declare other function parameters that are not part of the path parameters, they are automatically interpreted as "query" parameters.

```
@app.get("/greet/{time_of_day}/{name}")
def read_path_params(time_of_day: str, name: str);
    return {"message":f"Good {time_of_day}, {name}!"}
```

```
http://127.0.0.1:8000/greet?time_of_day=morning&name=SaiKiran
```

FastAPI Documentation

An advantage of using FastAPI is that it generates a nice documentation for each Route defined in the application and describes the Route-Parameters based on the type-hints used in our source-code (eg- item_id: int specifies that the parameter item_id is expected to be of type integer).If a client requests the API with a parameter with invalid data type value, FastAPI will automatically show the a clean, useful Error .

Head over to http://localhost:8000/docs to see our new handler's documentation. FastAPI dev server notices changes made to files and re-starts the server when needed to incorporate changes. So we don't have to manually restart the server each time.

There is also an Alternative documentation format supported by FastAPI, we can view that by going to http://localhost:8000/redoc

A bigger example

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from typing import List

app=FastAPI()

class Item(BaseModel):
    id: int
    name: str
    description: str = None
    price: float
    tax: float = None
```

```
items = []
```

```
@app.get("/")
def read_root():
    return {"message":"Hello, World"}
@app.get("/items",response_model=List[Item])
def read_items():
    return items
@app.get("/items/{item_id}",response_model=Item)
def read_item(item_id:int)
    for item in items:
        if item.id == item_id:
            return item
    raise HTTPException(status_code=404,detail="Item not found")
```

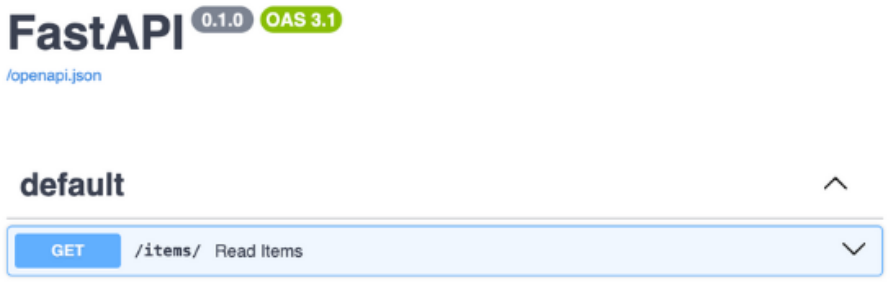
```
@app.post("/items",response_model=Item)
def create_item(item: Item):
    items.append(item)
    return item
@app.put("/items/{item_id}",response_model=Item)
def update_item(item_id: int, item: Item):
    for index, existing_item in enumerate(items):
        if existing_item.id==item_id:
            items[index]=item
            return item
    raise HTTPException(status_coe=404,detail="Item not found")
```

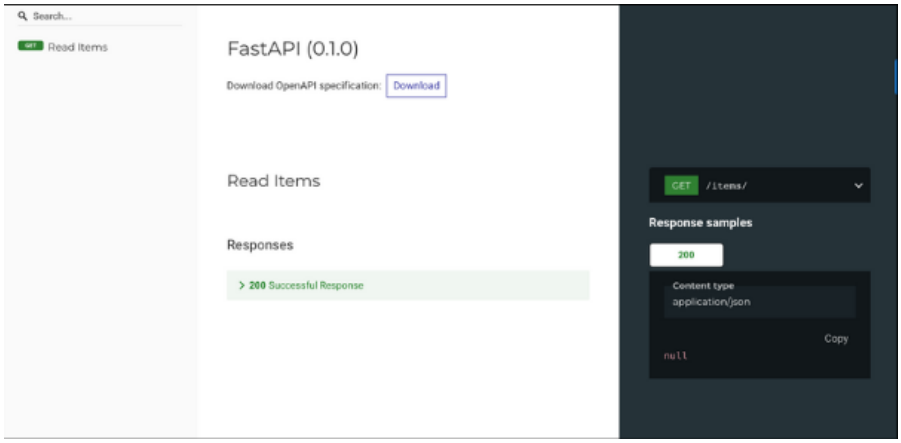
```
@app.delete("/items/{item_id}",response_model=Item)
def delete_item(item_id: int):
    for index,existing_item in enumerate(items):
        if existing_item.id == item_id:
            return items.pop(index)
      raise HTTPException(status_coe=404,detail="Item not found")
```

Swagger UI in FastAPI

FastAPI automatically generates an interactive API documentation interface using Swagger UI and Redoc

```python
from fastapi import FastAPI
app = FastAPI()
@app.get("/items/")
async def read_items():
return [{"item_id": 1, "name": "Item 1"}, {"item_id": 2, "name": "Item 2"}]
```





Pydantic

Pydantic is the most widely used validation library for Python. It is written in Rust and is fast and extensible, building upon Python's type hints, it is an easy to use robust way to validate data.

One of the primary ways of defining schema in Pydantic is via models. Models are simply classes which inherit from pydantic.BaseModel and define fields as annotated attributes.

Models share many similarities with Python's dataclasses.

Basic Usage

```
from pydantic import BaseModel, Field // import the base class
class User(BaseModel): // create our model using a class definition
   id: int // defining fields with type hints and additionally
   name: str = Field('Jane Doe') // a default value (similar to Python
dataclasses)
```

Then, we can instantiate an object of this model (class) with the values for each field and two things can happen (broadly):

- The fields satisfy validation constraints and we successfully create an instance of the model which is guaranteed to follow the defined schema.
- Eg: user = User(id='123')

- Or, the validation process results in a ValidationError showing that the input data is not and can not be coerced to the schema provided by the Model and object creation fails.
- Eg: user = User(id='1a')REST APIs adhere to six key principles that guide their design and functionality:
- The model's instance is guaranteed to have data that adheres to the Model Schema but the input itself may or may not be so, this is a subtle difference but it is important to understand.
- The data from the model's instance can be converted to a dictionary using the model_dump() method on the instance. This is different from dict(user) in terms that dic() doesn't recursively convert nested models to dictionaries while .model_dump() does.

Examples

```
from datetime import datetime
from typing import Tuple
from pydantic import BaseModel
class Delivery(BaseModel):
   timestamp: datetime
   dimensions: Tuple[int, int]
m = Delivery(timestamp= '2020-01-02T03:04:05Z' , dimensions=['10' , '20'])
print(repr(m.timestamp))
#> datetime.datetime(2020, 1, 2, 3, 4, 5, tzinfo=TzInfo(UTC))
print(m.dimensions)
#> (10, 20)
```

Binary I/O in Python

- Binary I/O (also called buffered I/O) expects bytes-like objects and produces bytes objects. No encoding, decoding, or newline translation is performed. This category of streams can be used for all kinds of non-text data, and also when manual control over the handling of text data is desired.
- The easiest way to create a binary stream is with open() with 'b' in the mode string:

```
f = open("myfile.jpg" , "rb")
```

- In-memory binary streams are also available as BytesIO objects:
- f = io.BytesIO(b"\x00\x01") # some initial binary data:
- Buffered I/O streams provide a higher-level interface to an I/O device than raw I/O does

open() vs io.BytesIO()

- Using open opens a file on your hard drive. Depending on what mode you use, you can read or write (or both) from the disk.
- A BytesIO object isn't associated with any real file on the disk. It's just a chunk of memory that behaves like a file does.
- It has the same API as a file object returned from open (with mode r+b, allowing reading and writing of binary data).
- BytesIO (and it's close sibling StringIO which is always in text mode) can be useful when you need to pass data to or from an API that expect to be given a file object, but where you'd prefer to pass the data directly.
- You can load your input data you have into the BytesIO before giving it to the library.
- After it returns, you can get any data the library wrote to the file from the BytesIO using the getvalue() method. (Usually you'd only need to do one of those, of course.).

An example use case is dealing with Image file data

```
from io import BytesIO
from PIL import Image

# Create an image using PIL
image = Image.new('RGB' , (100, 100), color = 'red')

# Create a BytesIO object to hold the image data
buffer = BytesIO()

# Save the image to the buffer
image.save(buffer, format= 'JPEG')

# Move the cursor to the beginning of the buffer
buffer.seek(0)

# Read the binary data (this would be the JPEG file content
image_data = buffer.read()
print(image_data[:10]) # Print the first 10 bytes of the image data
```

An example with FastAPI

Let's say we want to send the image file data over the network but reading and writing to a file on the hard disk is not very efficient and would need us to clear any temporary files so created, in such use cases, the BytesIO buffers are helpful.

```
from io import BytesIO
from fastapi import FastAPI, Response

app = FastAPI()

@app.get("/image/")
async def get_image():
  # Create an image using PIL
  image = Image.new('RGB' , (100, 100), color= 'blue')

  # Create a BytesIO object to hold the image data
  buffer = BytesIO()
  image.save(buffer, format= 'PNG')
  buffer.seek(0)

# Return the image data as a response
return Response(content=buffer.getvalue(), media_type= "image/png")
```

CRUD Operations in FastAPI

Create: Add a new resource.

Read: Retrieve existing resources

Update: Modify existing resources.

Delete: Remove existing resources.

Use Cases:

-Managing items in an inventory system.

. -Handling user accounts in an application.

-Working with blog posts or articles in a content management system.

Setting Up the Backend

FastAPI and SQLite

```
from fastapi import FastAPI
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
app = FastAPI()
Base = declarative_base()
engine = create_engine("sqlite:///./test.db")
SessionLocal = sessionmaker(bind=engine)
```

SQLite is a lightweight, disk-based database that doesn't require a separate server.

Database Model and Pydantic Schema

SQLAlchemy: Defines the database schema.

Pydantic: Validates and parses the data.

```
class Item(Base):
   __tablename__ = "items"
   id = Column(Integer, primary_key=True)
   title = Column(String)
   description = Column(String)

class ItemCreate(BaseModel):
   title: str
   description: str
```

Setting Up the Frontend

Initialize the React Project

```
npm create vite@latest . -- --template react
npm install
```

Install Axios

```
npm install axios
```

Create state variables

```
// Inside App.jsx
const [items, setItems] = useState([]);
const [title, setTitle] = useState("");
```

```
const [description, setDescription] = useState("");
const [editId, setEditId] = useState(null);
```

Create Operation in FastAPI

The "Create" operation uses the POST method to add new resources to the server. For example, you might add a new item to a database by sending the necessary data as a JSON object. FastAPI uses Pydantic models to validate and structure this data before processing it.

Backend

```
@app.post("/items/", response_model=ItemResponse)
def create_item(item: ItemCreate):
    db = SessionLocal()
    db_item = Item(**item.dict())
    db.add(db_item)
    db.commit()
    db.refresh(db_item)
    return db_item
```

```
const createItem = async () => {
    await axios.post("http://127.0.0.1:8000/items/", {
        title,
        description,
    });
    setTitle("");
    setDescription("");
    fetchItems();
};
```

```
<form onSubmit={handleSubmit}>
    <input
      type="text"
      value={title}
      onChange={(e) => setTitle(e.target.value)}
    />
    <input
      type="text"
      value={description}
      onChange={(e) => setDescription(e.target.value)}
```

```
    />
    <button type="submit">Create</button>
</form>
```

Read Operation in FastAPI

The "Read" operation relies on the GET method, allowing you to retrieve one or more resources. You can fetch all available items or filter them based on specific criteria, such as an item ID. Path parameters are often used to dynamically retrieve specific resources.

Backend

```
@app.get("/items/", response_model=List[ItemResponse])
def read_items():
    db = SessionLocal()
    items = db.query(Item).all()
    db.close()
    return items
```

```
const [items, setItems] = useState([]);
const fetchItems = async () => {
    const response = await axios.get("http://127.0.0.1:8000/items/");
    setItems(response.data);
};
useEffect(() => {
    fetchItems();
}, []);
```

```
<ul>
    {items.map((item) => (
        <li
            key={item.id}
            <div className=" flex flex-row items-center gap-2">
                <strong>{item.title}</strong>: {item.description}
            </div>
        </li>
    ))}
</ul>
```

```
@app.get("/items/")
async def read_items():
   return [{"item_id": 1, "name": "Item 1"}, {"item_id": 2, "name":
 "Item 2"}]
@app.get("/items/{item_id}")
async def read_item(item_id: int):
   return {"item_id": item_id, "name": "Item Name"}
```

Update Operation in FastAPI

When you need to modify existing data, the "Update" operation comes into play. Using the PUT or PATCH methods, you can either fully replace a resource or update only specific fields. This operation also leverages Pydantic models to ensure data consistency.

Backend

```
@app.put("/items/{item_id}", response_model=ItemResponse)
def update_item(item_id: int, item: ItemCreate):
   db = SessionLocal()
   db_item = db.query(Item).filter(Item.id == item_id).first()
   if db_item is None:
      raise HTTPException(status_code=404, detail="Item not found")
   db_item.title = item.title
   db_item.description = item.description
   db.commit()
   db.refresh(db_item)
   db.close()
   return db_item
```

```
{/* Add this to item */}
<button
     onClick={() => {
         setEditId(item.id);
         setTitle(item.title);
         setDescription(item.description);
      }}>
         Edit
 </button>
```

```
const updateItem = async () => {
```

```
    await axios.put(`http://127.0.0.1:8000/items/${editId}`, {
        title,
        description,
    });
    setTitle("");
    setDescription("");
    setEditId(null);
    fetchItems();
};
```

Delete Operation in FastAPI

Finally, the "Delete" operation removes a resource from the server using the DELETE method. This operation typically requires specifying the resource by an identifier, such as an item ID. It's essential to handle this operation carefully, potentially implementing soft deletes to preserve data integrity.

Backend

```
@app.delete("/items/{item_id}")
def delete_item(item_id: int):
    db = SessionLocal()
    db_item = db.query(Item).filter(Item.id == item_id).first()
    if db_item is None:
        raise HTTPException(status_code=404, detail="Item not found")
    db.delete(db_item)
    db.commit()
    db.close()
    return {"ok": True}
```

```
{/* Add this to item */}
<button onClick={()=> deleteItem(item.id)}>
      Delete
</button>

const deleteItem = async (id) => {
    await axios.delete(`http://127.0.0.1:8000/items/${id}`);
    fetchItems();
};
```

```
@app.delete("/items/{item_id}")
async def delete_item(item_id: int):
```

```
    return {"message": f"Item {item_id} deleted"}
```

BackgroundTasks() in FastAPI

FastAPI allows you to execute tasks in the background using the BackgroundTasks class. This feature is particularly useful for tasks that are time-consuming and don't need to block the response to the client. For example, tasks like sending an email, logging data, or processing files can be done in the background. When a request is received, the response is sent immediately to the client, while the background task continues to run. This non-blocking behavior enhances the performance and responsiveness of your API. You simply add tasks to the BackgroundTasks object, and FastAPI takes care of running them after the main response is completed.

BackgroundTasks Example

```
from fastapi import FastAPI, BackgroundTasks

app = FastAPI()

def write_log(message: str):
    with open("log.txt", "a") as log:
        log.write(message + "\n")

@app.post("/send-notification/")
async def send_notification(background_tasks: BackgroundTasks):
    background_tasks.add_task(write_log, "Notification sent")
    return {"message": "Notification sent, logging in the background"}
```

Middleware

- A middleware runs between the reception of all requests and the execution of the function that actually runs at the given route's functionality.
- Middlewares can be used to perform many things including but not limited to handling authentication, body parsing, CORS handling etc.
- Multiple middlewares can be chained together to perform multiple tasks sequentially.

Why Middleware?

- Middlewares allow programmers to reduce repetitive code and allow programmers to process all incoming requests uniformly without much effort.
- The need also arises due to the fact that data formats that are efficient to send across a network are not necessarily human readable or easily malleable in the backend.

How to use Middlewares in FastAPI?

```
from fastapi import FastAPI, Request
app = FastAPI()
@app.middleware("http")
   async def authenticate_user(request: Request, call_next):
      ... # Authentication Logic
      ...
      response = await call_next(request)
      return response
```

Classes for handling requests

When working with FastAPI, you will often interact with HTTP requests. Two important classes for handling requests are Request and HTTPConnection. While they share some similarities, each has its specific use cases and functionalities.

Request Class in FastAPI

The Request class is designed for handling HTTP requests in FastAPI. It inherits from HTTPConnection, adding additional functionality specific to HTTP requests.

Importing Requests:

```
from fastapi import Request
```

Example:

```
from fastapi import FastAPI, Request
app = FastAPI()
@app.get("/items/{item_id}")
async def read_item(item_id: int, request: Request):
   headers = request.headers
   query_params = request.query_params
   body = await request.body()
   return {
      "item_id": item_id,
      "headers": headers,
      "query_params": query_params,
      "body":body.decode()
}
```

HTTPConnection Class in FastAPI

The HTTPConnection class is a more general class that can be used for both HTTP and WebSocket connections. It provides functionalities common to both connection types.

Importing HTTPConnection:

```
From fastapi.requests import HTTPConnection
```

Example

```
from fastapi import FastAPI
from fastapi.requests import HTTPConnection

app = FastAPI()

@app.get("/info")
async def get_info(conn: HTTPConnection):
   url = conn.url
   headers = conn.headers
   return {"url": url, "headers": headers}
```

Dependencies

- In FastAPI, managing dependencies and security is crucial for building robust applications.
- Two key functions for handling these aspects are Depends() and Security().
- Both functions allow you to declare dependencies, but they cater to different needs.

Depends()

- The Depends() function is used to declare dependencies in FastAPI. It enables you to inject reusable components or functions into your route handlers.
- Import Depends from fastapi and use it to specify a callable that FastAPI will invoke.
- Allows you to declare a dependency that is automatically resolved by FastAPI.
- By default, dependencies are cached within a single request. Set use_cache to False if you need the dependency to be re-evaluated if used multiple times in the same request.

```
from typing import Annotated
from fastapi import Depends, FastAPI
app = FastAPI()
async def common_parameters(q: str | None = None, skip: int = 0, limit: int =
100):
   return {"q": q, "skip": skip, "limit": limit}
@app.get("/items/")
async def read_items(commons: Annotated[dict, Depends(common_parameters)]):
   return commons
```

In this example, common_parameters is a dependency injected into the read_items endpoint. It provides query parameters and pagination options

Security():
The Security() function is a specialized form of Depends() used for handling security-related dependencies, such as OAuth2 scopes.

Import Security from fastapi and use it similarly to Depends(), but with added functionality for OAuth2 scopes.
Allows you to declare dependencies that also involve security aspects like authorization and authentication.

You can specify OAuth2 scopes required for accessing certain endpoints.
These scopes are integrated into the OpenAPI specification and the API documentation

```
from typing import Annotated
from fastapi import Security, FastAPI
from .db import User
from .security import get_current_active_user

app = FastAPI()

@app.get("/users/me/items/")
async def read_own_items(
   current_user: Annotated[User, Security(get_current_active_user, scopes=
["items"])]
):
return [{"item_id": "Foo" , "owner": current_user.username}]
```

Exporting and Integrating an ML model with FastAPI

Basics of Linear Regression:

Linear regression is a statistical method that models the relationship between a dependent variable (target) and one or more independent variables (features) by fitting a linear equation to the observed data.
The simplest form is simple linear regression, which uses one independent variable to predict a dependent variable. The equation of a simple linear regression line is:

$y=mx+c$

where:
y is the dependent variable (target)
x is the independent variable (feature)
m is the slope of the line (coefficient)
c is the y-intercept (constant)

The goal of linear regression is to find the values of m m and c c that minimize the error between the predicted values and the actual values.

Building Our Model

We'll use the car price dataset, where the price of a car is predicted based on its year of manufacture.

```
import pandas as pd
from sklearn.linear_model import LinearRegression
import pickle

data = {
   'year': [2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009],
   'price': [2000, 2100, 2150, 2200, 2300, 2400, 2450, 2500, 2600, 2700]
}

df = pd.DataFrame(data)

X = df[['year']]
y = df['price']

model = LinearRegression()
model.fit(X, y)

with open('car_price_model.pkl', 'wb') as f:
   pickle.dump(model, f)
```

The Request Body:

The data sent from the client side to the API is called a request body. The data sent from the API to the client is called a response body.

To define our request body, we'll use BaseModel from the pydantic module and define the format of the data we'll send to the API. We'll create a class that inherits BaseModel and define the features as the attributes of that class along with their type hints. pydantic ensures that these type hints are validated during runtime, generating an error when the data is invalid.

```
from pydantic import BaseModel

class RequestBody(BaseModel):
   year: int
```

The Endpoint

Now that we have a request body, all that's left to do is to add an endpoint that'll predict the car price and return it as a response:

```
from fastapi import FastAPI
import pickle
import uvicorn

app = FastAPI()


with open('car_price_model.pkl', 'rb') as f:
   model = pickle.load(f)

@app.post('/predict')
def predict(data: RequestBody):

   test_data = [[data.year]]

   predicted_price = model.predict(test_data)[0]

   return {'predicted_price': predicted_price}
```

```
if __name__ == "__main__":
   uvicorn.run(app, host="127.0.0.1", port=8000)
```

Testing the API using the axios library in react:

Introduction to Axios:

Axios is a popular JavaScript library used to make HTTP requests from the browser. It works in both the browser and Node.js environments. It supports promises, making it easy to handle asynchronous operations. Axios provides a simple API for sending HTTP requests and handling responses, and it can be used for making GET, POST, PUT, DELETE, and other types of HTTP requests.

Steps to Set Up and Use Axios in a React App:

Install Axios:
First, we need to install Axios in our React project.

```
npm install axios
```

Set Up FastAPI Backend:

Ensure your FastAPI backend is running on the port 8000

Create a React Component for Prediction:

Create a new file CarPricePredictor.js in the src directory:

```
import React, { useState } from 'react';
import axios from 'axios';

const CarPricePredictor = () => {
 const [year, setYear] = useState('');
 const [price, setPrice] = useState(null);

 const handlePredict = async () => {
   try {
     const response = await axios.post('http://127.0.0.1:8000/predict', { year:
parseInt(year) });
     setPrice(response.data.predicted_price);
```

```
    } catch (error) {
      console.error('There was an error predicting the car price!', error);
    }
  };

  return (
    <div>
      <h1>Car Price Predictor</h1>
      <input
        type="number"
        placeholder="Enter year"
        value={year}
        onChange={(e) => setYear(e.target.value)}
      />
      <button onClick={handlePredict}>Predict Price</button>
      {price !== null && (
        <div>
          <h2>Predicted Price: {price}</h2>
        </div>
      )}
    </div>
  );
};

export default CarPricePredictor;
```

Update App Component and run the app