Index

Experiment 1 : JavaScript Essentials

Interaction: alert, prompt, confirm

1. alert

```
alert(“Hello");
```

2. prompt:

```
result = prompt(title, [default]);
```

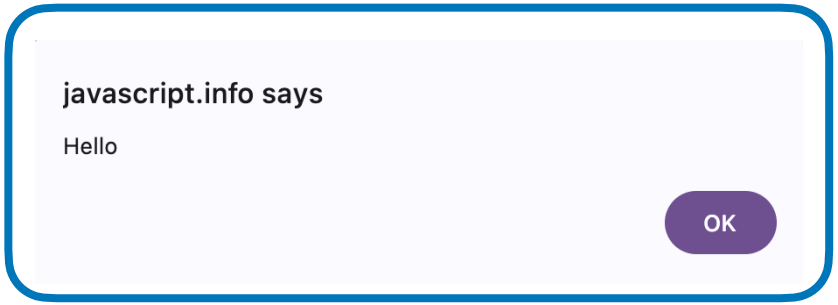
```
let age = prompt('How old are you?’,100);
```

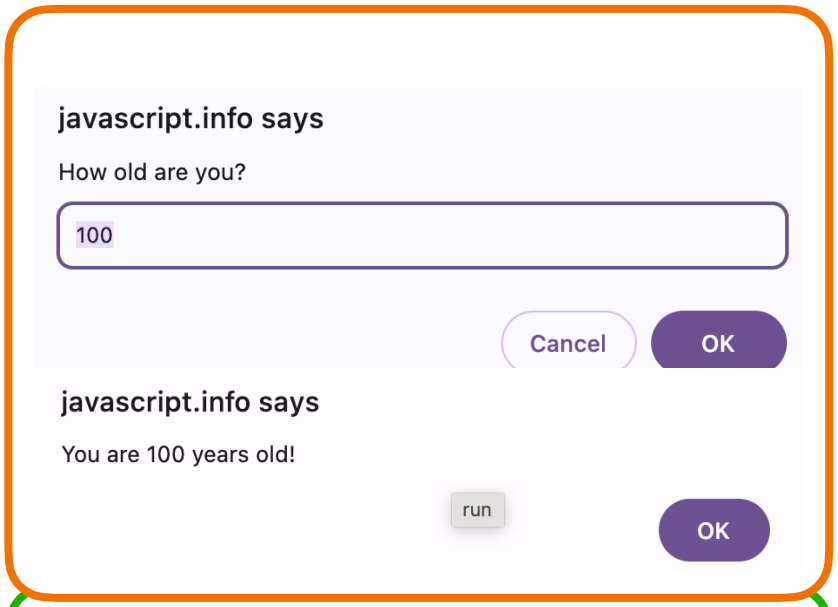
```
alert(`You are ${age} years old!`);
```

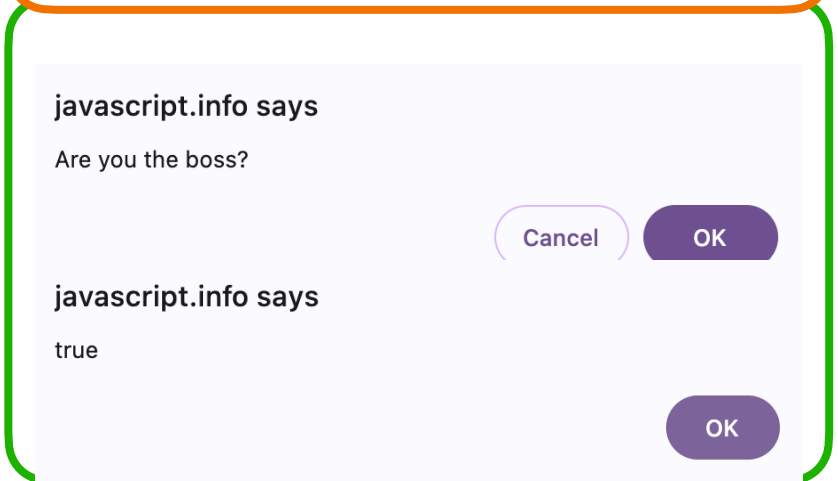
3. confirm:

```
let isBoss = confirm("Are you the boss?”);
```

```
alert( isBoss ); // true if OK is pressed
```







4. `console.log("Hello World"); //to print out`

Variables: let, var, const

```
let user = 'John';

let age = 25;

const myBirthday = '18.04.1982';

const COLOR_RED = "#F00";

let user = 'John', age = 25, message = 'Hello';
```

If we don't use "use strict"; then it is possible to create a variable by mere assignment of a value without using let.

```
num = 5; // the variable "num" is created if it didn't exist
```

- **Data Types:**
  - Dynamically typed
  - Eight Types:
    1. Number : Both integer and floating point

2. BigInt : represent integer values larger than (2^{53}-1)
3. String : Double or Single quotes.Unlike C no character type
4. Boolean : Two values: true and false.
5. null : represents “nothing”, “empty” or “value unknown”.
6. undefined : value is not assigned.
7. symbol : unique identifiers for objects
8. objects : It is the only non-primitive data type.

- **Type Conversion:**

```
typeof value // to identify type of value
String(1)    // type convert integer to String
```

strings are compared letter-by-letter. 'Glow' > ‘Glee'

```
alert( '2' > 1 ); // true, string '2' becomes a number 2
alert( '01' == 1 ); // true, string '01' becomes a number 1

alert( true == 1 ); // true
alert( false == 0 ); // true
```

```
alert( 0 === false ); // false, because the types are different
```

Conditional Operator ?

```
let result = condition ? value1 : value2;
```

The condition is evaluated: if it's true then value1 is returned, otherwise – value2.

```
Example: let accessAllowed = (age > 18) ? true : false;
```

=	Assignment operator
==	Equality test
===	Strict equality test without type conversion
!=	Not equals to

Function Declaration	Function Expression	Arrow Function

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

	Functions are values.They can be assigned, copied or declared in any place of the code.	

	Callback Functions:
	`function ask(question, yes, no) {`
	`  if (confirm(question)) yes()`
	`  else no();`
	`}`
	`function showOk() {`
	`  alert( "You agreed." );`
	`}`
	`function showCancel() {`
	`  alert( “canceled execution.");`
	`}`
	`ask("Do you agree?", showOk, showCancel);`
		Callback Functions:
		`function ask(question, yes, no)`
		`{`
		`  if (confirm(question)) yes();`
		`  else no();`
		`}`
		`ask("Do you agree?",`
		`  () => alert("You agreed."),`
		`  () => alert(“canceled execution.")`
		`);`

Objects:

- Used to store keyed collections of various data and more complex entities.
- An object can be created with figure brackets {…} with an optional list of *properties*. A property is a “key: value” pair, where key is a string (also called a “property name”), and value can be anything.

```
let user = new Object(); // "object constructor" syntax
let user = {};  // "object literal" syntax

let user = {     // an object
    name: "John",
    age: 30
  };
```

```
let user = {};

// set
user["likes birds"] = true;

// get
alert(user["likes birds"]); // true

delete user["likes birds”];
```

✓

```
let user = {
   name: "John",
   age: 30
 };

let key = prompt("What do you want to know about the user?", "name");

 // access by variable
 alert( user[key] );
 // John (if enter "name")
```

✗

```
let user = {
  name: "John",
  age: 30
};


let key = "name";
alert( user.key ) // undefined
```

Run time calculations can be done using square brackets but not using dot notations.

Additional operators:

To delete a property delete obj.prop

To check if a property with the given key exists: "key" in obj

To iterate over an object: for (let key in obj) loop.

Object references and copying (Copy by value and Copy by reference)

When an object variable is copied, the reference is copied, but the object itself is not duplicated.

```
let user = { name: 'John' };

let admin = user;

admin.name = 'Pete'; // changed by the "admin" reference

alert(user.name); // 'Pete', changes are seen from the "user" reference

let a = {};
let b = a; // copy the reference
let c = {};

alert( a == b ); // true, both variables reference the same object
alert( a === b ); // true
alert( a === c ); // false
```

Object Cloning (“shallow copy” (nested objects are copied by reference))

```
 let user = {
  name: "John",
  age: 30
};

let clone = Object.assign({}, user);

alert(clone.name); // John
alert(clone.age); // 30
```

Deep Cloning using structuredClone (nested objects are copied by value)

```
let user = {
  name: "John",
  sizes: {
    height: 182,
    width: 50
  }
};

let clone = structuredClone(user);
alert( user.sizes === clone.sizes ); // false, different objects
// user and clone are totally unrelated now
user.sizes.width = 60;    // change a property from one place
alert(clone.sizes.width); // 50, not related
```

Object Methods

```
let user = {
   name: "John",
   age: 30
};

user.sayHi = function() {
  alert("Hello!");
};

user.sayHi(); // Hello!
```

```
let user = {
    // ...
};

// first, declare
function sayHi() {
  alert("Hello!");
}

// then add as a method
user.sayHi = sayHi;

user.sayHi(); // Hello!
```

```
user = {
    sayHi: function() {
      alert("Hello");
    }
};
```

```
// method shorthand looks better,right?
user = {
    sayHi() {
      alert("Hello");
    }
};
```

```
let user = {
    name: "John",
    age: 30,

    sayHi() {
      // "this" is the
"current object"
      alert(this.name);
    }

  };

  user.sayHi(); // John
```

```
let user = {
    name: "John",
    age: 30,

    sayHi() {
      alert(user.name); //
"user" instead of "this"
    }

  };
```

```
let user = { name: "John" };
let admin = { name: "Admin" };

function sayHi() {
  alert( this.name );
}

// use the same function in two objects
user.f = sayHi;
admin.f = sayHi;

user.f(); // John  (this == user)
admin.f(); // Admin  (this == admin)

admin['f'](); // Admin (dot or square
brackets access method – doesn't matter)
```

```
let user = {
    name: "John",
    age: 30,

    sayHi() {
      alert( user.name ); // leads to an error
    }
  };

  let admin = user;
  user = null; // overwrite to make things obvious

  admin.sayHi(); // TypeError: Cannot read property
'name' of null
```

```
let user = {
    firstName: "Ilya",
    sayHi() {
      let arrow = () =>
alert(this.firstName);
      arrow();
    }
  };

  user.sayHi(); // Ilya
```

Constructor, operator “new"

Constructor functions:

1. They are named with capital letter first.
2. They should be executed only with "new" operator.

```
function User(name) {
    this.name = name;
    this.isAdmin = false;
  }

  let user = new User("Jack");

  alert(user.name);
// Jack

  alert(user.isAdmin);
// false
```

```
function User(name) {
    this.name = name;

    this.sayHi = function() {
      alert( "My name is: " +
this.name );
    };
  }

 let john = new User("John");

 john.sayHi(); //My name is:John
```

```
function BigUser() {

    this.name = "John";

    return { name: "Godzilla" };
   // <-- returns this object
  }

  alert( new BigUser().name );
// Godzilla, got that object
```

```
function SmallUser() {

    this.name = "John";

    return; // <-- returns this
  }

  alert( new SmallUser().name );
// John
```

String Methods	String Methods	Number Methods	Symbol methods
charAt(index) – Returns the character at the specified index. **at(index) -** unlike charAt, it allows negative index	**repeat(index)-** returns a string with a number of copies of a string	**toFixed(digits)** – Formats the number with a fixed number of digits after the decimal.	**toString()** – Converts the symbol to a string.
indexOf(substring) – Returns the index of the first occurrence of the substring.	**trim()** – Removes whitespace from both ends of a string. **trimStart(), trimEnd()**	**toString()** – Converts the number to a string.	**description** – Returns the description of the symbol.
toUpperCase() – Converts the string to uppercase. **toLowerCase()** – Converts the string to lowercase.	**replace(search, replacement)** – Replaces a substring with another value. **replaceAll(),**	**toExponential(fractionDigits)** – Converts the number to exponential notation.	BigInt Methods
slice(start, end) – Extracts a part of a string. **substring(start, end)** – Extracts characters from the string.	**split(separator)** – Splits the string into an array.	**toPrecision(significantDigits)** – Formats the number to a specific length.	**toString()** – Converts the BigInt to a string.
String Property	Boolean Methods	**isFinite(number)** – Checks if the number is finite.	**toLocaleString()** – Returns a locale-sensitive string representation.
length - Length of the string	**toString()** – Converts the boolean value to a string ("true" or "false").	**isNaN(value)** – Determines if the value is NaN (Not-a-Number).	**valueOf()** – Returns the primitive value of the BigInt.

A Primitive as an Object:

This is a paradox in JavaScript. Primitives must be as fast and lightweight as possible. Which means it should not contain methods etc. At the same time having methods helps us to do various things. So JavaScript provide methods to Primitives through object wrappers. These “object wrapper” that provides the extra functionality is created, and then is destroyed

For more concepts of Java script like Classes, Error handling, Promises, async/await, generators, advanced iteration, Modules refer to https://javascript.info/

Experiment 2 : Browser: Document, Events, Interfaces

HTML Styling using CSS – Inline and Internal

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Document</title>
    <style>
        p {
            color: aquamarine;
            background-color: blue;
        }
    </style>


</head>
<body style="background-color: bisque;">
    <h1 style="background-color: powderblue;">Hello World</h1>
    <p>Good Morning Everyone</p>
</body>
</html>
```

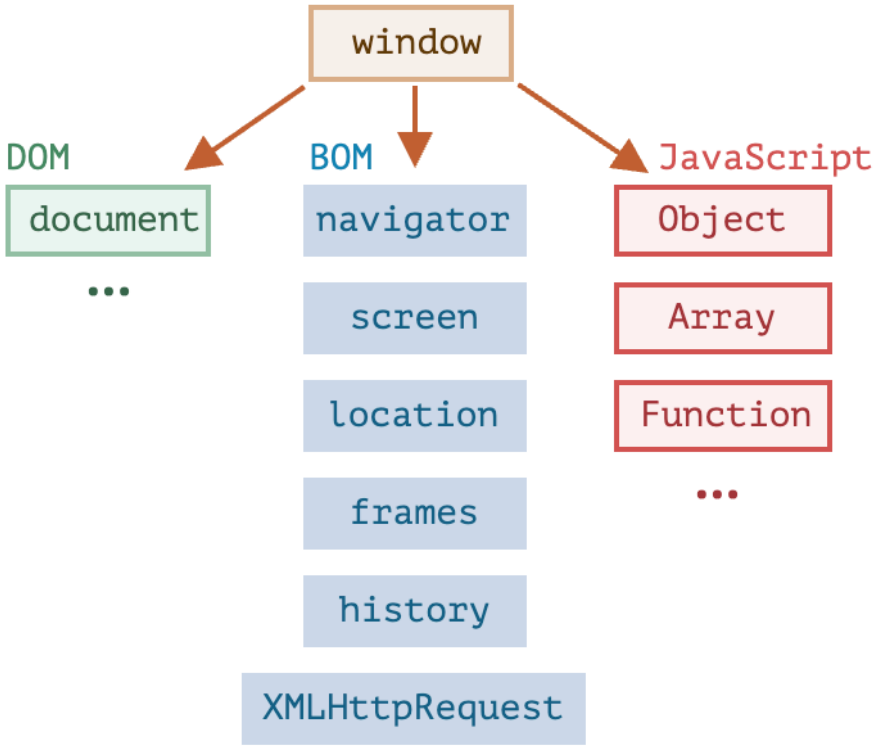
HTML Styling using CSS – External

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Document</title>
    <link rel="stylesheet" href="mycss1.css">
</head>
<body style="background-color: bisque;">
    <h1 style="background-color: powderblue;">Hello World</h1>
    <p>Good Morning Everyone</p>
</body>
</html>
```

mycss1.css

```
p {
   color:black;
   background-color:chartreuse;
}
```

Browser Environment



window object:

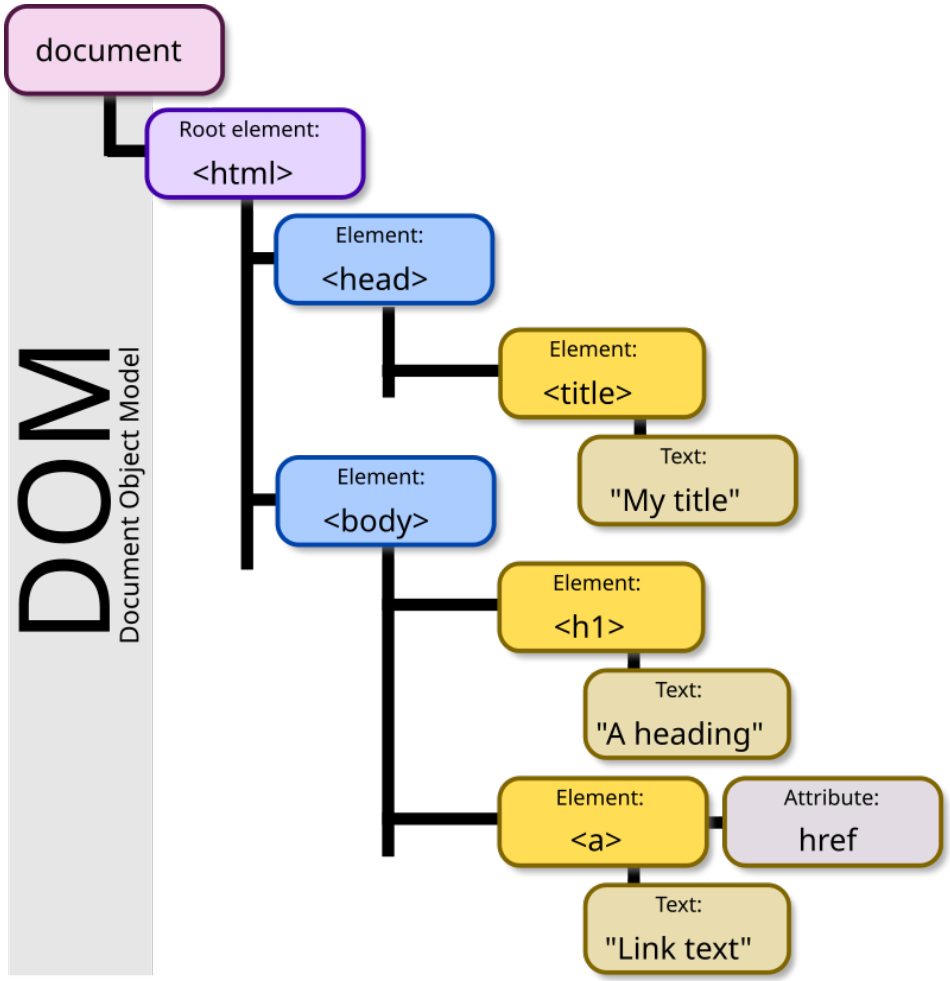
"root" object window has two roles:

It is a global object for JavaScript code.

It represents the "browser window" and provides methods to control it.

```
function sayHi()
{
   alert("Hi");
}

window.sayHi()
```

Document Object Model (DOM):



Document Object Model (**DOM**) is a cross-platform and language-independent interface that treats an HTML or XML document as a tree structure wherein each node is an object representing a part of the document.

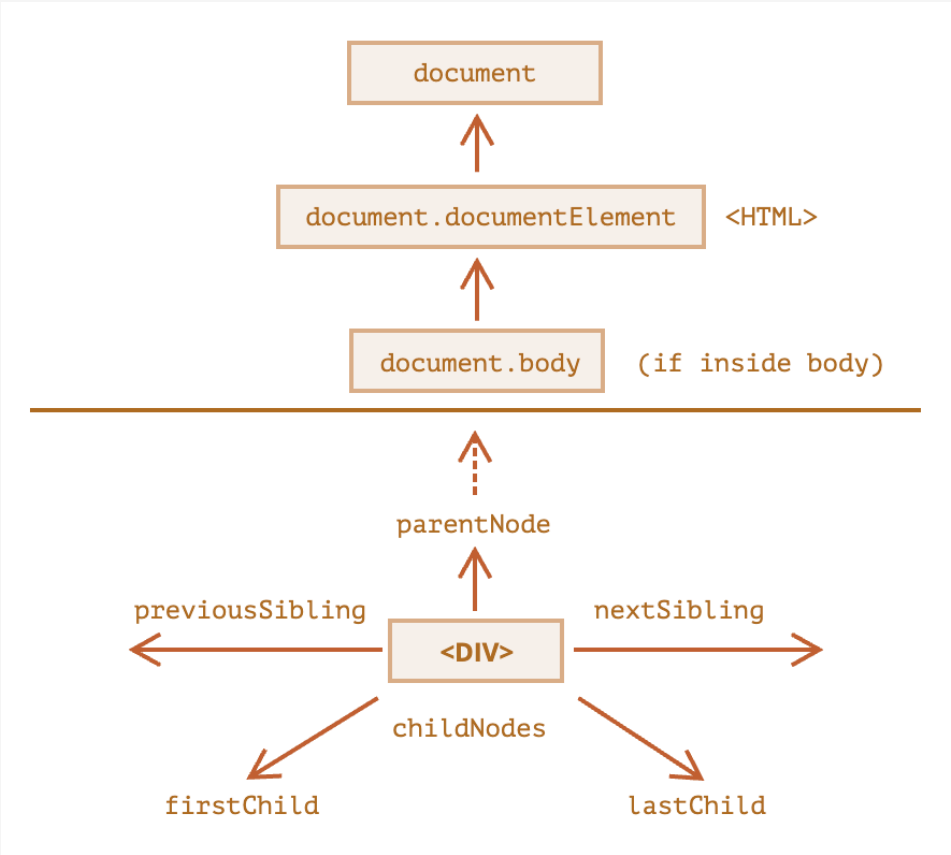
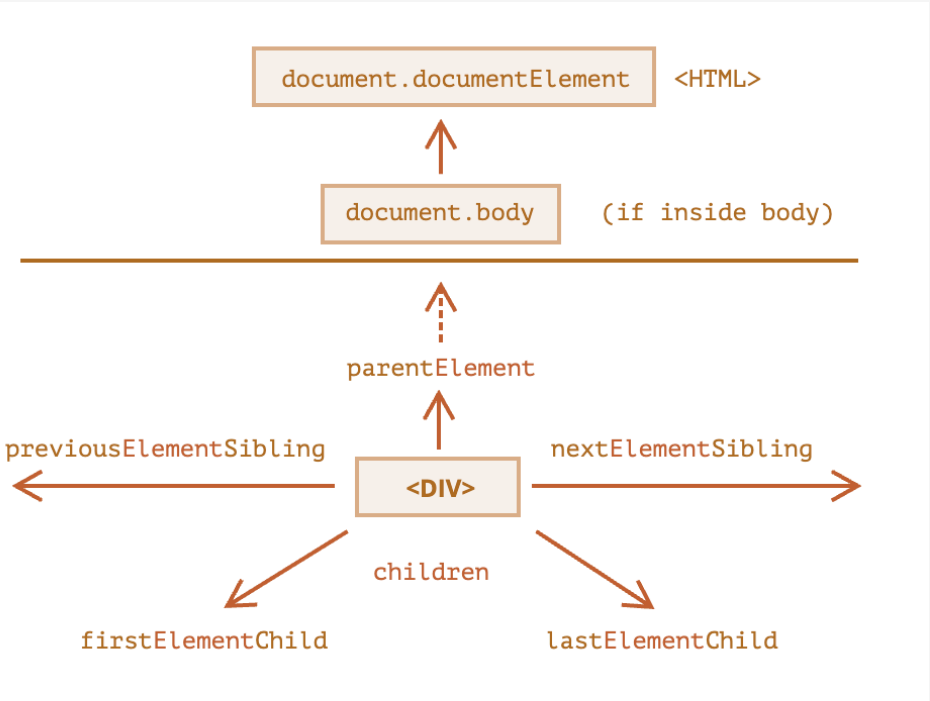
The DOM represents a document with a logical tree.

Each branch of the tree ends in a node, and each node contains objects.

DOM methods allow programmatic access to the tree; with them one can change the structure, style or content of a document.

Nodes can have event handlers (also known as event listeners) attached to them. Once an event is triggered, the event handlers get executed.

Walking the DOM

Travel between DOM Nodes	Element only Navigation
	

Modify the DOM elements using ids, tagnames and classnames in javascript

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Document</title>
</head>
<body>
    <h1 style="background-color: aquamarine;">Good Morning KLH</h1>
    <p id="para1">Hello World</p>
    <p class="myclass1">Hello Y23</p>
    <p class="myclass1">Hello Y22</p>

    <script>
        alert("Hello World")
        document.getElementById("para1").innerHTML = "Hello KLH"
        document.getElementsByTagName("h1")[0].innerHTML="Good Morning Hyderabad"
        document.getElementsByClassName("myclass1")[1].innerHTML="Hello Y21"
    </script>

</body>
</html>
```

setTimeout

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Document</title>
</head>
<body>
    <h1 style="background-color: aquamarine;">Good Morning KLH</h1>
    <script>
        function sayHi(){
            alert("Hi");
        }

        window.sayHi()

        document.body.style.background="red";

        setTimeout(() => document.body.style.background = "", 15000)

    </script>

</body>
</html>
```

Introduction to browser events

Event Handlers:

- To react on events we can assign a *handler* – a function that runs in case of an event.
- Handlers are a way to run JavaScript code in case of user actions.

Event handlers using HTML attribute:

```
<input value="Click me" onclick="alert('Click!')" type="button">
```

On mouse click, the code inside onclick runs.

Please note that inside onclick we use single quotes, because the attribute itself is in double quotes. If we forget that the code is inside the attribute and use double quotes inside, like this: onclick="alert("Click!")", then it won't work right.

```
<script>
    function countRabbits() {
      for(let i=1; i<=3; i++) {
        alert("Rabbit number " + i);
      }}
 </script>

 <input type="button" onclick="countRabbits()" value="Count rabbits!">
```

We can assign a handler using a DOM property on<event>.

```
<input id="elem" type="button" value="Click me">
<script>
  elem.onclick = function() {
    alert('Thank you');
  };
</script>
```

AddEventListener:

The fundamental problem of the aforementioned ways to assign handlers is that we *can't assign multiple handlers to one event*.

The syntax to add a handler:

```
element.addEventListener(event, handler, [options]);

element.removeEventListener(event, handler, [options]);
```

Multiple calls to `addEventListener` allow it to add multiple handlers, like this:

```
<input id="elem" type="button" value="Click me"/>

<script>
  function handler1() {
    alert('Thanks!');
  };

  function handler2() {
    alert('Thanks again!');
  }

  elem.onclick = () => alert("Hello");
  elem.addEventListener("click", handler1); // Thanks!
  elem.addEventListener("click", handler2); // Thanks again!
</script>
```

EventObject:

When an event happens, the browser creates an *event object*, puts details into it and passes it as an argument to the handler.

```
<input type="button" value="Click me" id="elem">

<script>
  elem.onclick = function(event) {
    // show event type, element and coordinates of the click
    alert(event.type + " at " + event.currentTarget);
    alert("Coordinates: " + event.clientX + ":" + event.clientY);
  };
</script>
```

Experiment 3 : React

React is a JavaScript library used to build user interfaces (UIs) for web and mobile apps. It allows us to easily create single page Apps (SPA).

Single Page Application in React improves user experience by dynamically updating the view and minimising server requests, while allowing for a faster, more interactive experience.

A key advantage of React is that it only re-renders those parts of the page that have changed, avoiding unnecessary re-rendering of unchanged DOM elements.

Components:

React code is made of entities called components. These components are modular and reusable. React applications typically consist of many layers of components. The components are rendered to a root element in the DOM using the React DOM library. When rendering a component, values are passed between components through *props* (short for "properties"). Values internal to a component are called its *state.*

In React, there are primarily two ways to declare components: **Functional Components** and **Class Components**.

Functional Components:

- These are simple JavaScript functions that return JSX (JavaScript XML) to define what the UI should look like.
- They are **stateless** (without React hooks) but can manage state and side effects using **hooks** like useState, useEffect, etc.

1.

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

2. With hooks:

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

  function Counter() {
    const [count, setCount] = useState(0);
    return (
      <div>
        <p>You clicked {count} times</p>
        <button onClick={() => setCount(count + 1)}>Click me</button>
      </div>
    );
  }
```

Class Components:

- Before React hooks, components that needed state or lifecycle methods were written as classes.
- These components extend `React.Component` and must have a `render` method that returns JSX.

```
class Welcome extends React.Component {
    render() {
      return <h1>Hello, {this.props.name}</h1>;
    }
  }
```

Class components have access to lifecycle methods like componentDidMount, shouldComponentUpdate, componentWillUnmount, etc.

Key Differences:

- **Functional Components** are generally preferred now due to their simplicity and the ability to use hooks.
- **Class Components** are still supported but are less commonly used in new React applications.

Other Types of Declarations:

- **Higher-Order Components (HOC)**: Functions that take a component and return a new component.
- **Render Props**: A technique for sharing code between components using a prop whose value is a function.

In modern React development, **functional components with hooks** are the standard approach.

In visual studio code install Simple React Snippets. It simplifies writing code in react. It generates boilerplate react components.

Emmet significantly speeds up the workflow for developers by allowing them to write JSX (JavaScript XML) code with just a few keystrokes. Inside settings lets type emmet. Edit emmet include Languages.

Item : javascript

Value : javascriptreact

Create React App is used for starter project. To do this a modern version of node to be installed. We can use npx to run create react app tool. To check if node is already installed and its version number run the below command in terminal.

```
node -v
```

The above command might result with the version number like v20.17.0. If it is not installed go to nodejs.org and download the current version and install it.

Create a folder named JavaScriptProjects using below terminal command

```
mkdir JavaScriptProjects
```

Run the below node command to create react app named **my-blog**

```
npx create-react-app my-blog
```

Open my-blog in visual studio code. You can see three folders: **node_modules**, **public**, **src**.

node_modules contains all project dependencies live including the react library. Any library we install later will also live in this folder.

public contains all public files live. These are public to the browser. Public folder contains **index.html** file. All of the react code is injected in to this file with id root.

src : This folder will contain the most of the react code we write. All the react components we write will reside in this folder. By default there is already **App.js** with some default component. We have **index.js** which takes all the react components and mount on the DOM at root id. There are some files which are not needed in the beginning. Those can be deleted like **App.test.js** and **reportWebVitals.js, setupTests.js.** Delete reportWebVitals import in the top and bottom of **index.js** file.

React.StrictMode in **index.js** gives warnings if there is any in react code.

package.json contains all packages. Usually node_modules will be too large. So when we copy a project, we usually avoid copying node_modules because of its size. But we can regenerate it based on all the packages mentioned in package.json using command **npm install** (Note: This command is needed only when we copy a project without packages of node_modules folder.)

Open a new terminal in visual studio code and run the following terminal command:

```
npm run start
```

You will see a local host address like **http://localhost:3000**

Experiment 4 : React Components

Components contain template and logic. Template contains JSX (syntax similar to HTML) code and logic contains javascript code. In **index.js** (part of **src** folder) we have only one component being rendered which is **App.** The **App.js** file contains a functional component which looks like a normal function that returns JSX code as shown below. In the background Babel (a compiler) converts the JSX code in to Javascript. Usually the function name is always capital and arrow functions can also be used if needed.

Emmet helps in creating div with classname easily. Type div.content and press tab (Note: content is the name of class), Emmet autofills and create div with class name as content as shown below.

App.js

```
import './App.css';

function App() {
  return (
    <div className="App">
      <div className="content">Hello World</div>
    </div>
  );
}
export default App;
```

index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';

const root =
ReactDOM.createRoot(document.getElementById('root
'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

One big difference between JSX and HTML is how we declare class. In JSX we use className, since class keyword is already reserved keyword in JavaScript. In the console, when you check the HTML code corresponding to className, we will have class.

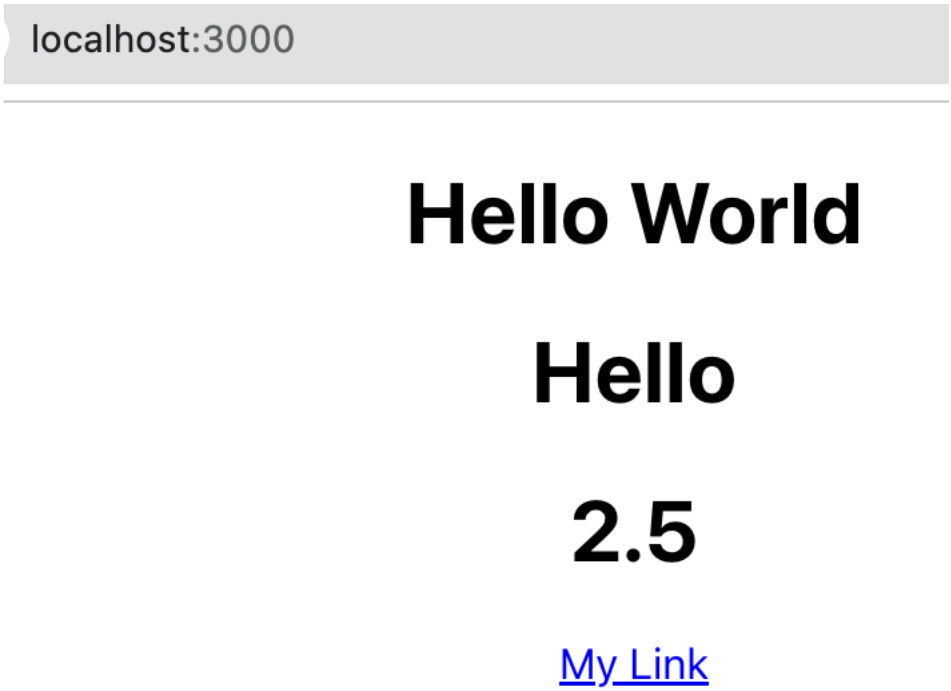
At the end of **App.js** we have export option and while importing in to other files like **index.js** use import App from ‘./App’;

Dynamic Values in Templates:

App.js

```
import './App.css';

function App() {

  const myStr="Hello";
  const a=2.5;
  const myLink="https://www.google.com";

  return (
    <div className="App">
     <div className="content">
      <h1>Hello World</h1>
      <h1>{myStr}</h1>
      <h1>{a}</h1>
      <a href={myLink}>My Link</a>

     </div>
    </div>
  );
}
export default App;
```

Output



Experiment 5 : Multiple Components

Components are structured as a tree called as component tree. **App.js** will be the root component. Now lets have **NavBar.js** as child component to **App.js** In **Navbar.js** use the shortcut sfc and press tab. It will create a stateless functional component. It generate an arrow function as shown below. Note this feature is available due to Simple React Snippets that we installed earlier.

```
const  = () => {
    return (  );
}

export default ;
```

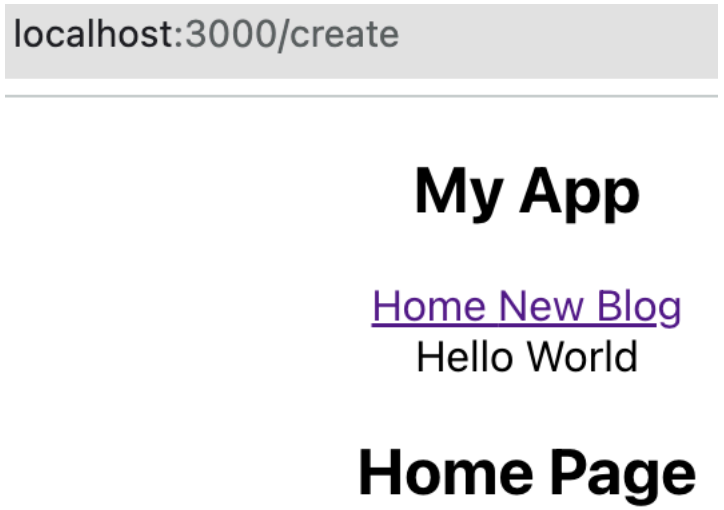
Navbar.js

```
const Navbar = () => {
    return (
        <nav className="navbar">
            <h1>My App </h1>
            <div className="links">
                <a href="/"> Home </a>
                <a href="/create"> New Blog </a>
            </div>
        </nav>
     );
}
export default Navbar;
```

Home.js

```
const Home = () => {

    return (

        <div className="home">

            <h2> Home Page</h2>

        </div>
     );
}
export default Home;
```

App.js	Output

`import './App.css';`
`import Navbar from './Navbar.js';`
`import Home from './Home.js';`
 	
`function App() {`
`  return (`
`  <div className="App">`
`  <Navbar/>`
`  <div className="content">Hello World</div>`
`  <Home />`
`  </div>`
`  );`
`}`
`export default App;`
	

Experiment 6 : Change Styling.

Delete **App.css** and change the content of **index.css** to below.

```
@import url('https://fonts.googleapis.com/css2?family=Quicksand:wght@300;400;500;600;700&display=swap');

/* base styles */
* {
  margin: 0;
  font-family: "Quicksand";
  color: #333;
}
.navbar {
  padding: 20px;
  display: flex;
  align-items: center;
  max-width: 600px;
  margin: 0 auto;
  border-bottom: 1px solid #f2f2f2;
}
.navbar h1 {
  color: #f1356d;
}
.navbar .links {
  margin-left: auto;
}
.navbar a {
  margin-left: 16px;
  text-decoration: none;
  padding: 6px;
```

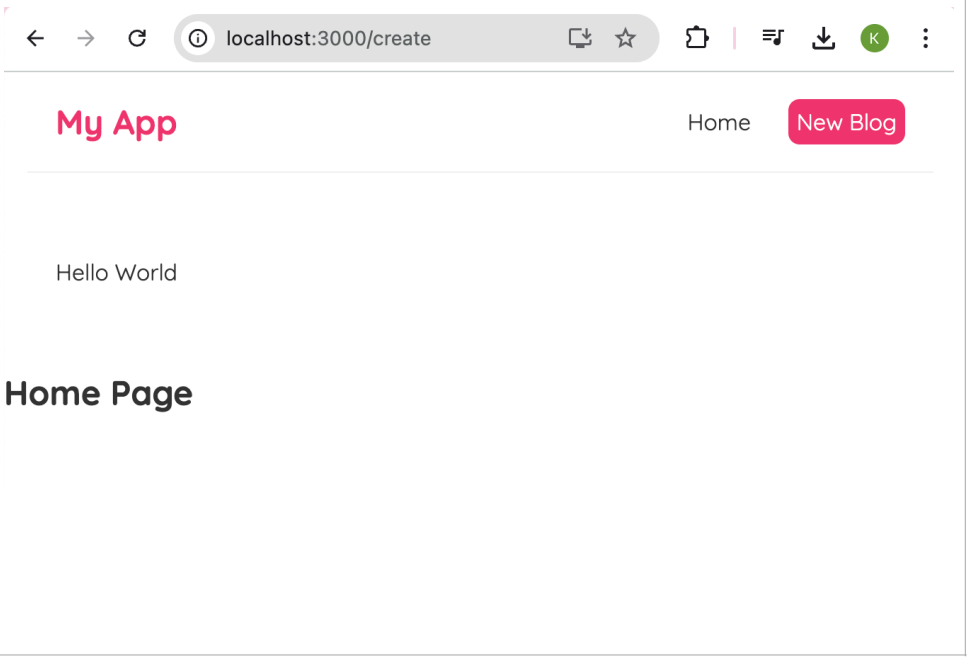
```
}
.navbar a:hover {
  color: #f1356d;
}
.content {
  max-width: 600px;
  margin: 40px auto;
  padding: 20px;
}
```

Inline styling in **Navbar.js** can be done as follows. Note it is not CSS, but it is JSX.

Navbar.js

```
const Navbar = () => {
    return (
        <nav className="navbar">
            <h1>My App </h1>
            <div className="links">
                <a href="/"> Home </a>
                <a href="/create" style={{
                    color:"white",
                    backgroundColor:'#f1356d',
                    borderRadius: '8px'
                }}> New Blog </a>
            </div>
        </nav>
    );
}

export default Navbar;
```

Output



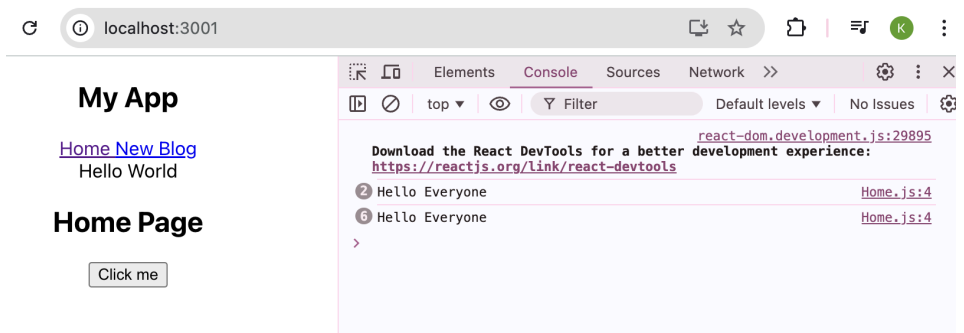
Experiment 7 : Click Events

In a website we have number of events like Hover events, click events, scroll events, form submission events, keyboard events etc. Here we look in to the specific case of Click Events.

Home.js

```
const Home = () => {
    const handleClick=() => {
        console.log("Hello Everyone")
    }

    return (
        <div className="home">
            <h2> Home Page</h2>
            <button onClick={handleClick}>Click
me</button>
        </div>
     );
}
export default Home;
```

Output



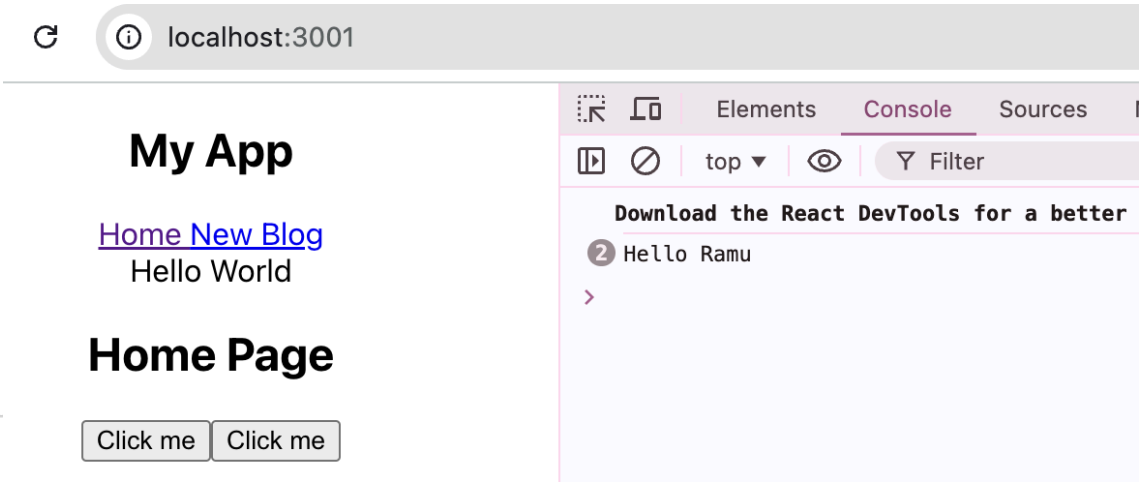
In the onClick, we should not write parenthesis like handleClick(). If we do so, it automatically executes without the click of button. Instead we should write as handleClick. Then the function gets executed only after the click.

If we want to give arguments to the handleClick function, we have to use arrow function as shown below. Only when the button pressed the arrow function is going to get executed. Also by using default events arguments, we can extract the property details as shown later.

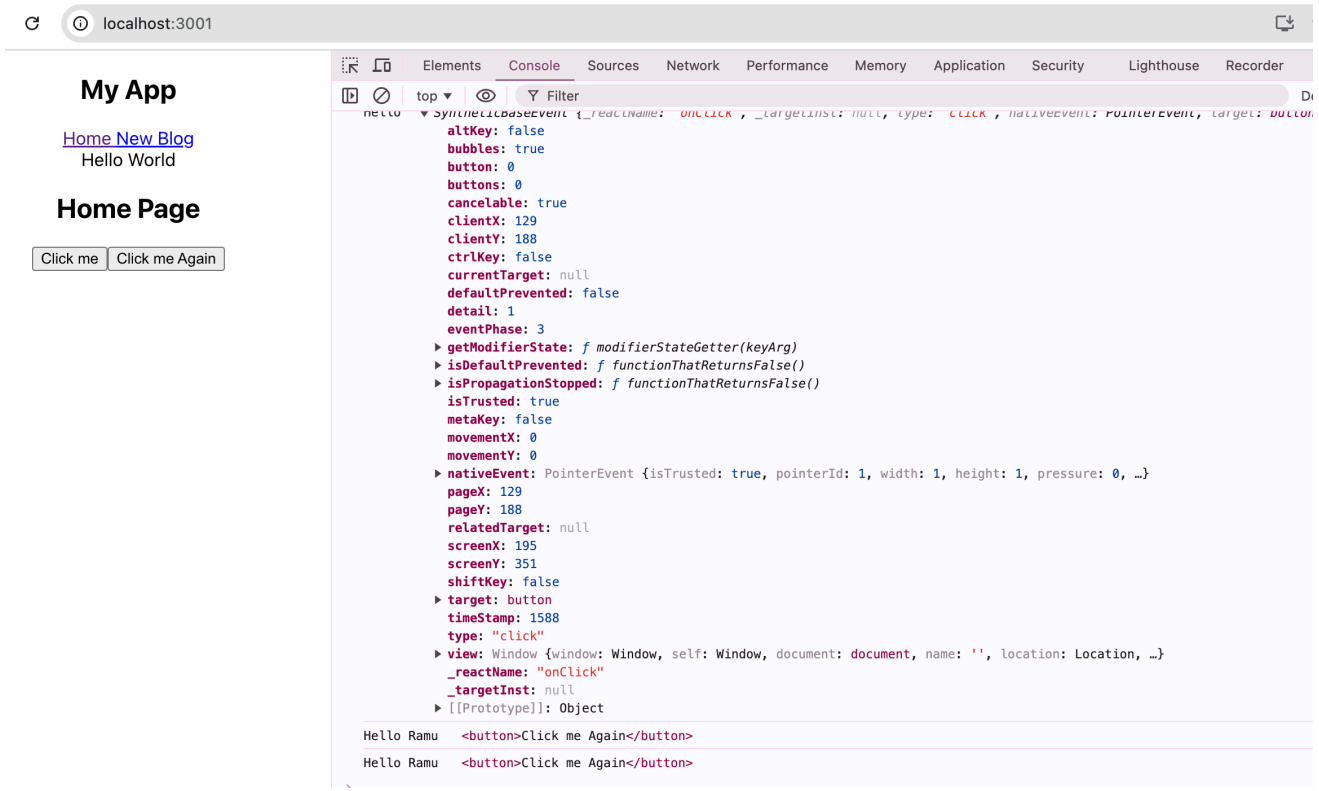
Home.js

```
const Home = () => {

    const handleClickAgain = (name) =>  {
        console.log("Hello "+name);
    }

    return (

        <div className="home">
            <h2> Home Page</h2>
            <button onClick={() => handleClickAgain('Ramu')}>Click me</button>
        </div>
     );
}
export default Home;
```

Output:



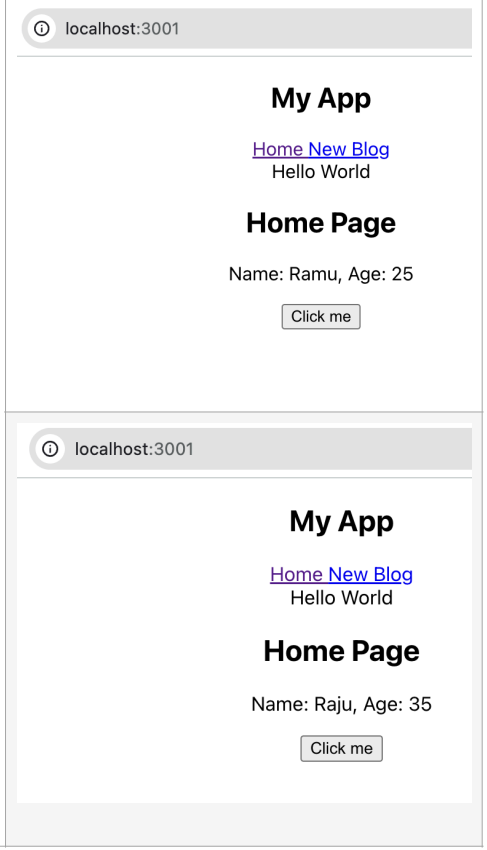
```jsx
const Home = () => {
    const handleClick = (e) =>  {
        console.log("Hello ",e);
    }
    const handleClickAgain = (name,e) =>  {
        console.log("Hello "+name,e.target);
    }
    return (
        <div className="home">
            <h2> Home Page</h2>
            <button onClick={handleClick}>Click me</button>
            <button onClick={(e) => handleClickAgain('Ramu',e)}>Click me Again</button>
        </div>
     );
}
export default Home;
```



Experiment 8 : useState Hook

When the variable defined is non-reactive it does not change its value even after updating in a clickHandle function. To get it updated and rendered everywhere, we should make the variable reactive, which means when ever its gets changed it re-renders all the page with updated value. To make them reactive we use something called as hook, more specifically useState hook as shown below.

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

const Home = () => {

    const [name,setName]=useState('Ramu');
    const [age,setAge]=useState('25');

    const handleClick = () =>  {
        console.log("Hello Everyone");
        setAge('35');
        setName("Raju");
    }
    return (
        <div className="home">
            <h2> Home Page</h2>
            <p> Name: {name}, Age: {age} </p>
            <button onClick={handleClick}>Click
me</button>
        </div>
     );
}
export default Home;
```

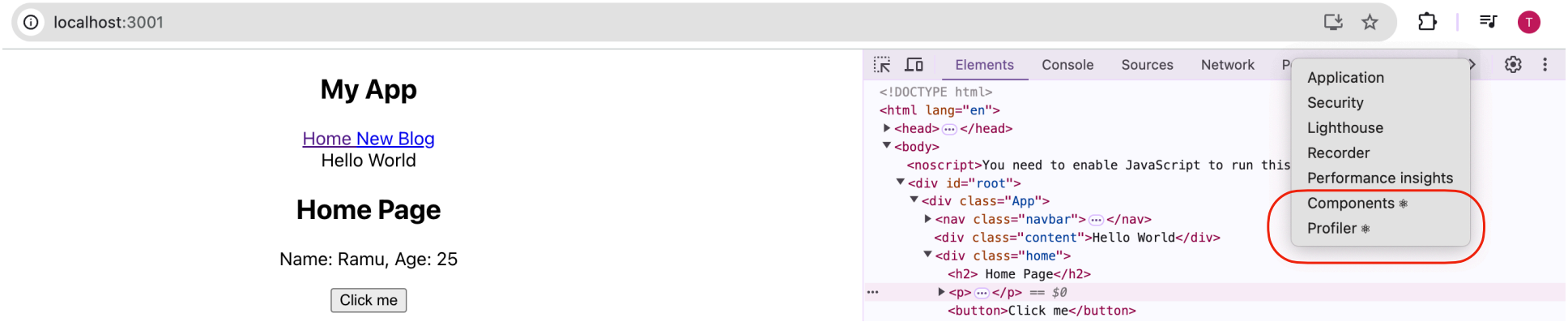


Experiment 9 : React Developer Tools

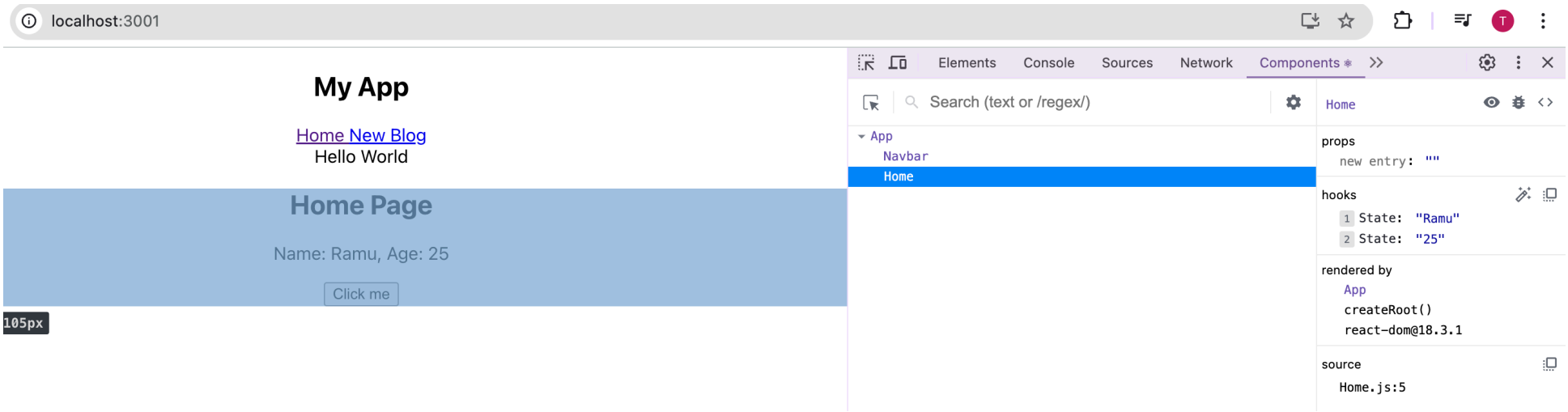
Search for React developer tools in Google and install the extension in chrome or Firefox. **React Developer Tools** is a browser extension that helps developers inspect and debug React applications.

- **Component Tree Inspection:** It allows you to view the structure of your React app, showing the hierarchy of components. You can inspect the **props, state, and hooks** of any component in the tree, making it easier to understand data flow.
- **Performance Monitoring:** It includes a **Profiler** that helps you measure the performance of your React app. You can see which components render slowly and optimize them.
- **Debugging Hooks:** You can view the values of React hooks (like `useState` or `useEffect`) directly in the developer tools, which is useful when debugging functional components.
- **Updating Component State**: You can directly modify the state or props of a component to test changes in real time, without editing your code.
- **Context Inspection**: It helps you inspect and debug **React Context** values, making it easy to track context-related data flow.

Right click on the output webpage. Click on inspect and click on right arrows to see two new attributes called as **components** and **profiler.**

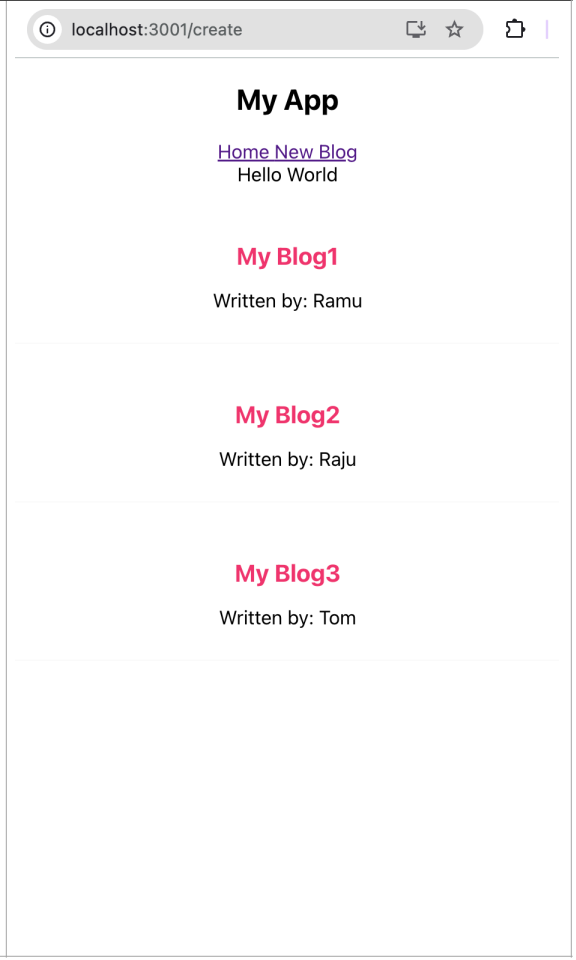


If you click on components it gives the component tree. If you hover on Home as shown below, it highlights the corresponding section in the output webpage. Also it displays the various properties like props, hooks and state values.



Experiment 10 : Outputting Lists

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

const Home = () => {

    const [blogs,setBlogs]=useState([
        {title:"My Blog1", body:"Body of Blog1", author:"Ramu", id:1},
        {title:"My Blog2", body:"Body of Blog2", author:"Raju", id:2},
        {title:"My Blog3", body:"Body of Blog3", author:"Tom", id:3}
    ]);
    return (
        <div className="home">
         {blogs.map((blog) => (
            <div className="blog-preview" key={blog.id}>
            <h2>{blog.title}</h2>
            <p>Written by: {blog.author}</p>

            </div>
         ))

         }
        </div>
     );
}
export default Home;
```



Place the below styling in **index.css**

```
.blog-preview {
  padding: 10px 16px;
  margin: 20px 0;
  border-bottom: 1px solid #fafafa;
}
.blog-preview:hover {
  box-shadow: 1px 3px 5px rgba(0,0,0,0.1);
}
.blog-preview h2 {
  font-size: 20px;
  color: #f1356d;
  margin-bottom: 8px;
}
```

When we want to create templates of similar type, it is best to store the properties as objects in an array. We can use javascript map to iterate over each object to create multiple templates. Manually each template could have been hardcoded, but the disadvantage is it will be tedious and does not allow to modify if the data is modified, deleted or increased in number. So instead of hardcoding we can use map to run over all objects irrespective of the count and it automatically updates if there is any modification or any deletion.

Experiment 11 : Props

Components can be reused in to other components, thereby lot of repetitive code can be avoided. Between the components data can be shared using props. Props are a way to send the data from parent component to child component.

Home.js

```
import { useState } from "react";
import BlogList from './BlogList';

const Home = () => {

    const [blogs,setBlogs]=useState([
       {title:"My Blog1", body:"Body of Blog1", author:"Ramu", id:1},
       {title:"My Blog2", body:"Body of Blog2", author:"Raju", id:2},
       {title:"My Blog3", body:"Body of Blog3", author:"Tom", id:3}
    ]);
    return (
        <div className="home">
         < BlogList blogs={blogs} title="All Blogs!" />
        </div>
     );
}
export default Home;
```

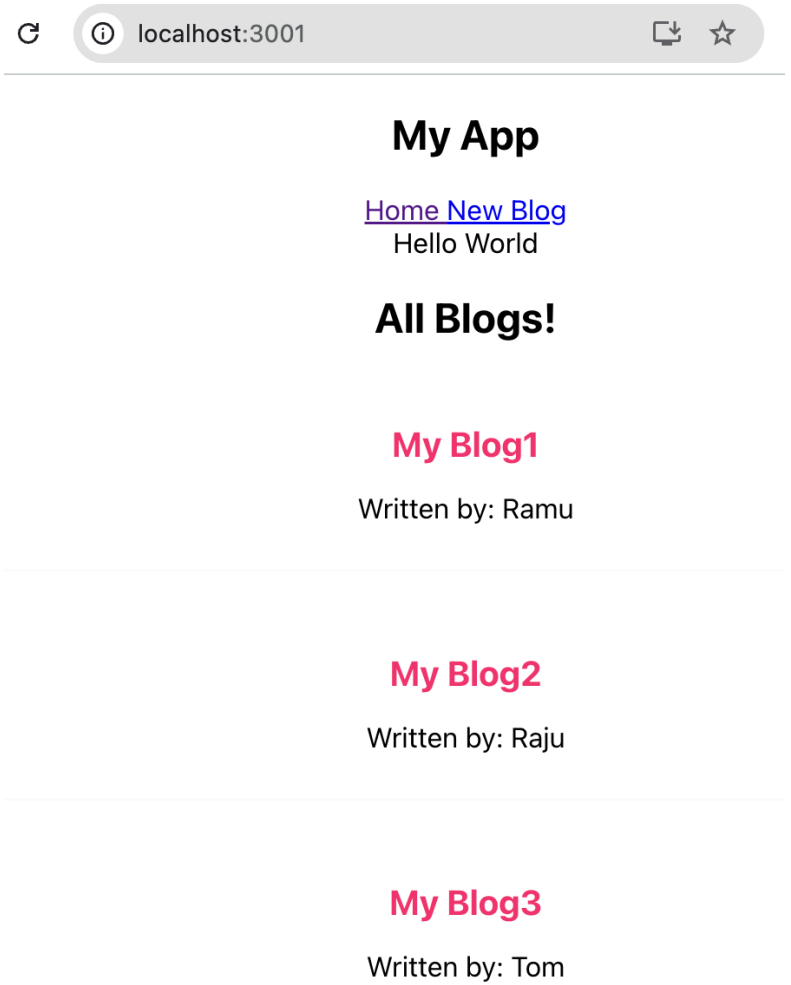
BlogList.js

```
const BlogList = (props) => {
    const blogs=props.blogs;
    const title=props.title;

    console.log(props,blogs)

    return (
    <div className="blog-list">
       <h2>{title}</h2>
       {blogs.map((blog) => (
    <div className="blog-preview" key={blog.id}>
        <h2>{blog.title}</h2>
        <p>Written by: {blog.author}</p>

           </div>
         ))

         }
        </div>
      );
}
export default BlogList;
```

```
const BlogList = (props) => {
    const blogs=props.blogs;
    const title=props.title;
```

The above code can also be alternatively written as shown below. Both the above and below code works.

```
const BlogList = ({blogs, title}) => {
```

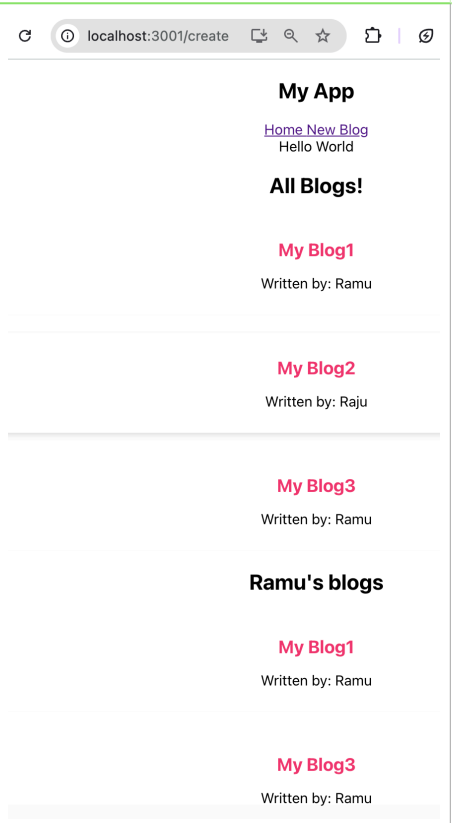


Experiment 12 : Reusing Components

Home.js

```
import { useState } from "react";
import BlogList from './BlogList';

const Home = () => {

    const [blogs,setBlogs]=useState([
        {title:"My Blog1", body:"Body of Blog1", author:"Ramu",
id:1},
        {title:"My Blog2", body:"Body of Blog2", author:"Raju",
id:2},
        {title:"My Blog3", body:"Body of Blog3", author:"Ramu", id:3}
    ]);
    return (
        <div className="home">
         < BlogList blogs={blogs} title="All Blogs!" />
         < BlogList blogs={blogs.filter((blog) => blog.author ===
'Ramu')} title="Ramu's blogs" />
        </div>
     );
}
export default Home;
```

Output.js



By having a separate javascript file for BlogList, we can reuse it multiple times. In the above example based on the filter condition it generates different outputs. Such a filter option can be used in Search bar to filter the content.

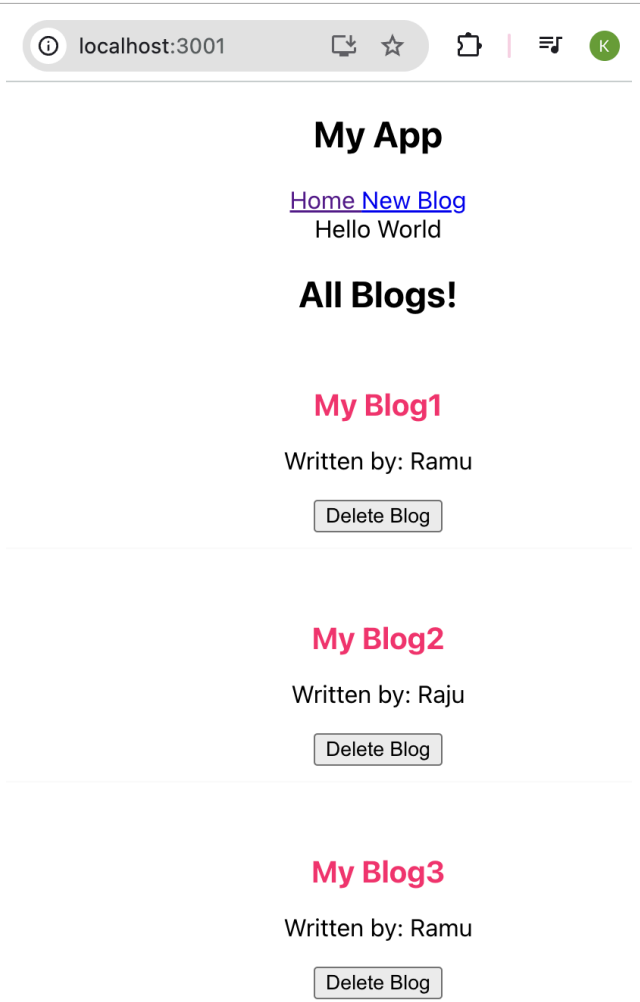
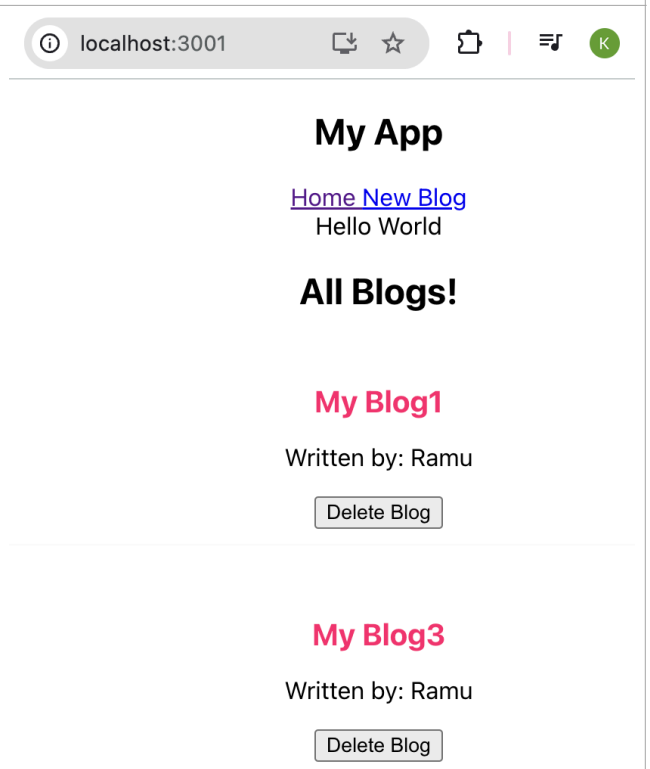
Experiment 13 : Functions as Props

Home.js

```
import { useState } from "react";
import BlogList from './BlogList';

const Home = () => {

    const [blogs,setBlogs]=useState([
        {title:"My Blog1", body:"Body of Blog1",
author:"Ramu", id:1},
        {title:"My Blog2", body:"Body of Blog2",
author:"Raju", id:2},
        {title:"My Blog3", body:"Body of Blog3",
author:"Ramu", id:3}
    ]);
    const handleDelete = (id) => {
        const newBlogs=blogs.filter(blog =>
blog.id !== id);
        setBlogs(newBlogs);
    }
    return (
        <div className="home">
         < BlogList blogs={blogs} title="All
Blogs!" handleDelete={handleDelete} />
        </div>
     );
}
export default Home;
```

BlogList.js

```
const BlogList = ({blogs, title, handleDelete}) =>
{

    return (
        <div className="blog-list">
            <h2>{title}</h2>
            {blogs.map((blog) => (
            <div className="blog-preview"
key={blog.id}>
            <h2>{blog.title}</h2>
            <p>Written by: {blog.author}</p>

            <button onClick={() =>
handleDelete(blog.id)}> Delete Blog</button>

            </div>
         ))

         }
        </div>
      );
}

export default BlogList;
```

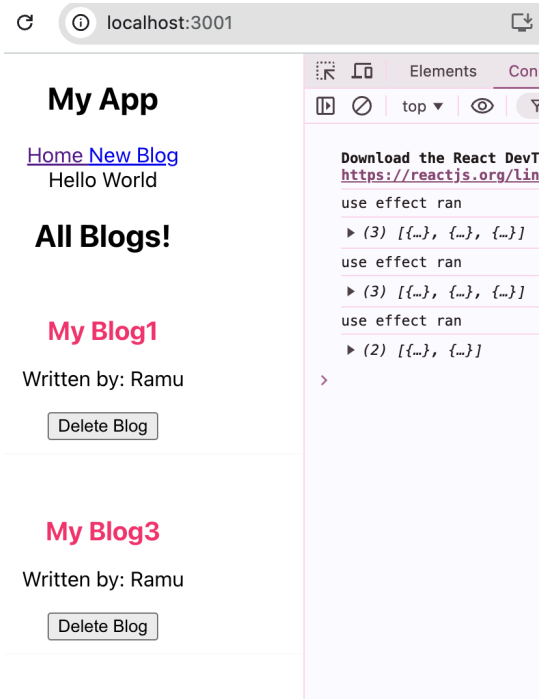
	Outputs
Functions can also be send as props. handleDelete function is sent as props. This functions is executed in BlogList.js file. Also setBlogs handleDelete function creates a new array without deleted entries. However when we render we see the original output that shows all entries.	 

Experiment 14 : useEffect Hook and Dependencies

Home.js

```
import { useState, useEffect } from "react";
import BlogList from './BlogList';

const Home = () => {

    const [blogs,setBlogs]=useState([
        {title:"My Blog1", body:"Body of Blog1", author:"Ramu", id:1},
        {title:"My Blog2", body:"Body of Blog2", author:"Raju", id:2},
        {title:"My Blog3", body:"Body of Blog3", author:"Ramu", id:3}
    ]);

    const handleDelete = (id) => {
        const newBlogs=blogs.filter(blog => blog.id !== id);
        setBlogs(newBlogs);
    }

    useEffect(() => {
        console.log('use effect ran');
        console.log(blogs);
    });
    return (
        <div className="home">
         < BlogList blogs={blogs} title="All Blogs!"
handleDelete={handleDelete}/>
        </div>
     );
}
export default Home;
```

Output



Explanation:

Every time we render it useEffect Hook gets executed that can be seen in above output.

Even if there is a state change, useEffect hook will be executed.

Using empty dependency array like as shown below will make the useEffect Hook to run the function only in the first initial render. Then after even if it changes state it will not run.

```
useEffect(() => {
        console.log('use effect ran');
        console.log(blogs);
    }, []);
```

However if we want the useEffect to be triggered based on change in some specified state we could instead mention the variable in the dependency array. Based on its state it will render. In the below example whenever name changes (here due to a button click), useEffect hook gets executed.

```
const [name, setName]=useState('mario')

    useEffect(() => {
        console.log('use effect ran');
        console.log(blogs);
    },[name]);

    return (
        <div className="home">
         < BlogList blogs={blogs} title="All Blogs!" handleDelete={handleDelete} />
         <button onClick={() => setName('luigi')}>Set Name</button>
         <p>{name}</p>
        </div>
     );
```

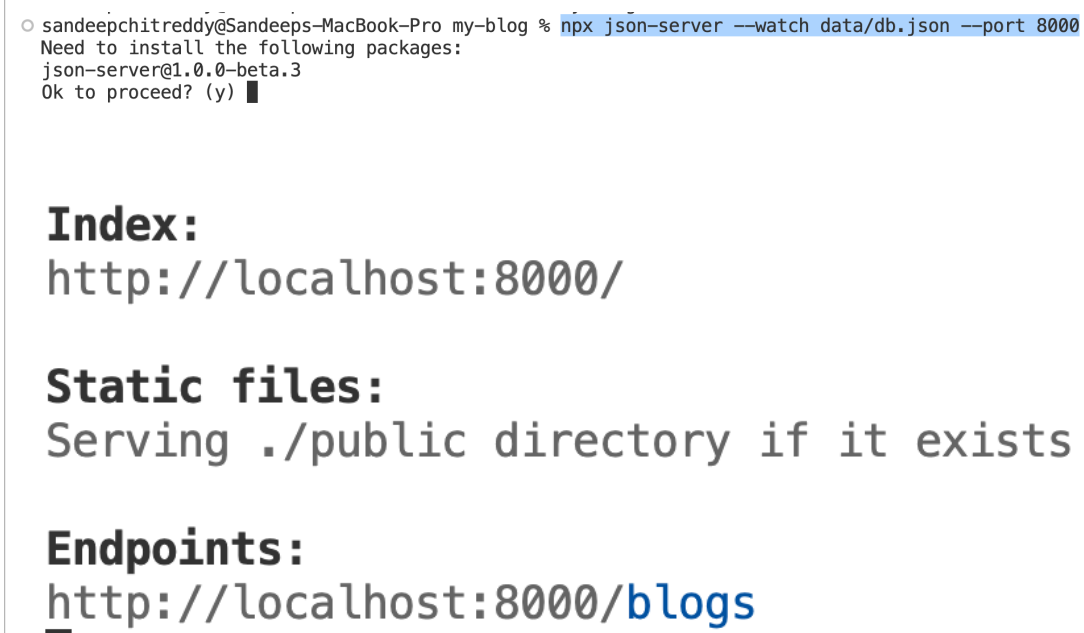
Experiment 15 : JSON Server

Usually in a web application data is not stored directly in the javascript files. It is instead stored in database and through api endpoints it would be accessed. An another way is to use a package called JSON server which is like a fake API just using JSON file. This JSON file contains the data. Create a folder called **data** inside the project and right click to create a new file called **db.json**. Top level property in the JSON file is called as a resource and it creates endpoints to interact with it. We can add, delete, insert and get items from this endpoint. To get the end point link we run the following terminal command.

npx json-server --watch data/db.json --port 8000

First it will ask to install JSON server as shown below. Press y to install it. Then you will see the following outputs

JSON Server Installation



Endpoint Output

Experiment 16 : Fetching Data with useEffect from JSON server

Home.js

```
import { useState, useEffect } from "react";
import BlogList from './BlogList';

const Home = () => {

    const [blogs,setBlogs]=useState(null);

    useEffect(() => {
       fetch('http://localhost:8000/blogs')
       .then(res => {
            return res.json();
       })
       .then(data => {
        setBlogs(data);
       })
    },[]);

    return (
        <div className="home">
         { blogs && <BlogList blogs={blogs}
title="All Blogs!"/> }
        </div>
     );
}
export default Home;
```

BlogList.js

```
const BlogList = ({blogs, title}) => {
    return (
        <div className="blog-list">
            <h2>{title}</h2>
            {blogs.map((blog) => (
            <div className="blog-preview"
key={blog.id}>
            <h2>{blog.title}</h2>
            <p>Written by: {blog.author}</p>
            </div>
         ))
         }
        </div>
      );
}
export default BlogList;
```

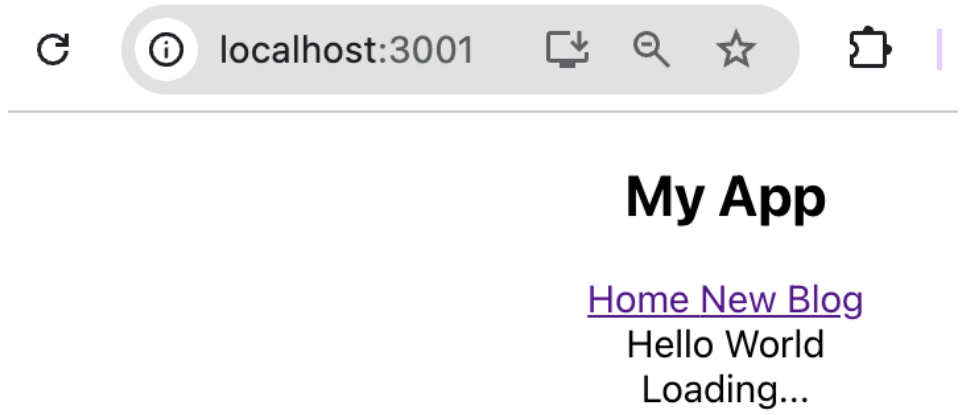
db.json

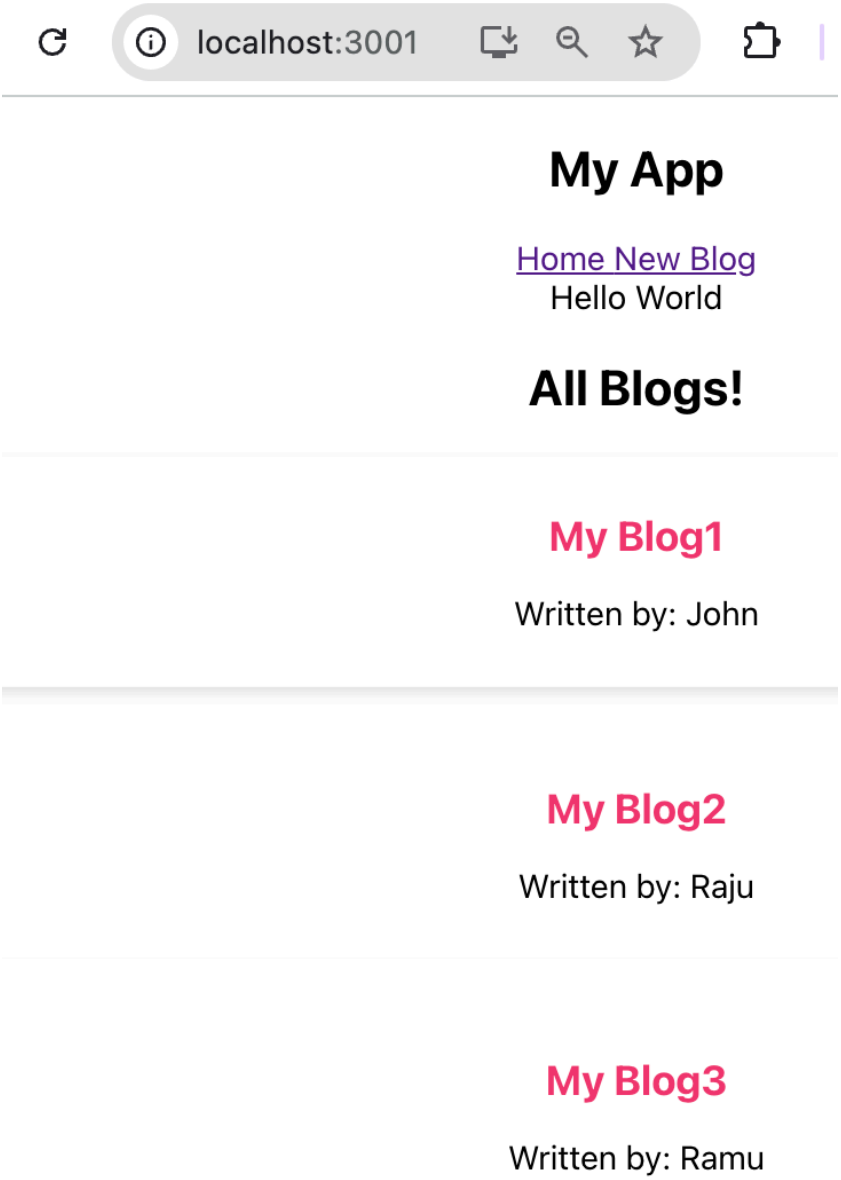
```
{
    "blogs" :[
        {"title":"My Blog1", "body":"Body of Blog1",
"author":"John", "id":1},
        {"title":"My Blog2", "body":"Body of Blog2",
"author":"Raju", "id":2},
        {"title":"My Blog3", "body":"Body of Blog3",
"author":"Ramu", "id":3}
    ]
}
```

Experiment 17 : Conditional Loading Message

```
import { useState, useEffect } from "react";
import BlogList from './BlogList';

const Home = () => {
    const [blogs,setBlogs]=useState(null);
    const [isPending, setIsPending]=useState(true);

    useEffect(() => {
       setTimeout( () => {
        fetch('http://localhost:8000/blogs')
       .then(res => {
            return res.json();
       })
       .then(data => {
        setBlogs(data);
        setIsPending(false);
       })
       }, 5000);
    },[]);

    return (
        <div className="home">
         { isPending && <div>Loading... </div> }
         { blogs && <BlogList blogs={blogs} title="All Blogs!"/> }
        </div>
     );
}
export default Home;
```

```
const BlogList = ({blogs,
title}) => {
    return (
        <div className="blog-
list">
            <h2>{title}</h2>
            {blogs.map((blog)
=> (
            <div
className="blog-preview"
key={blog.id}>
            <h2>{blog.title}</
h2>
            <p>Written by:
{blog.author}</p>
            </div>
          ))
          }
        </div>
      );
}
export default BlogList;
```





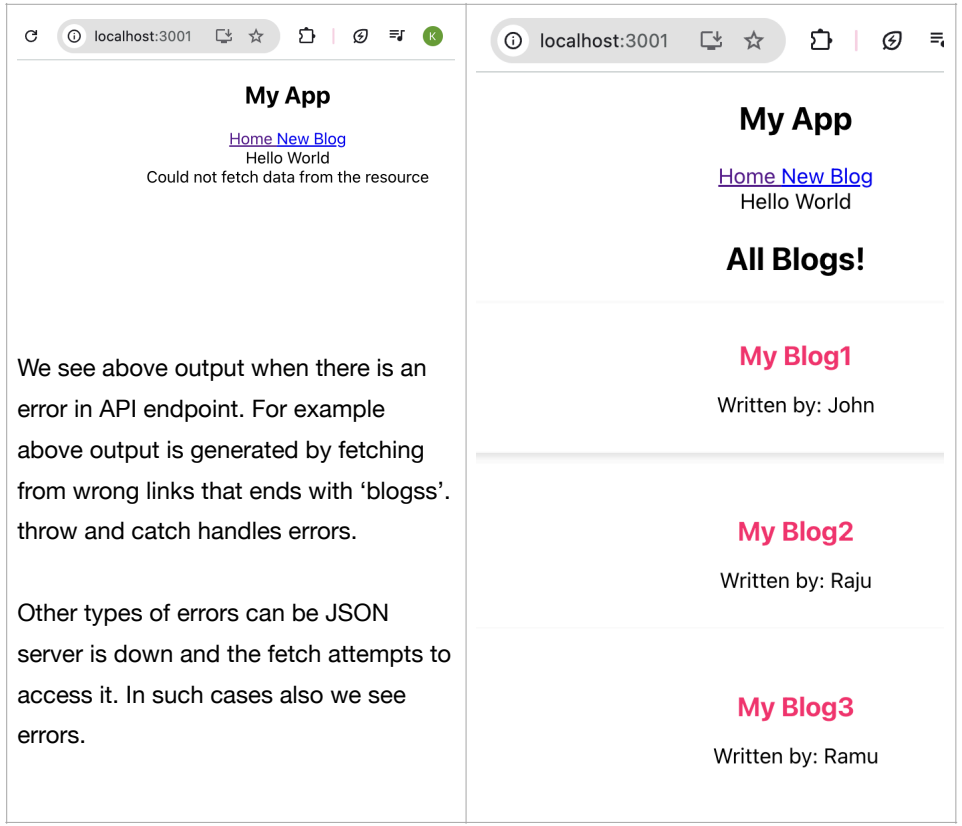
Output Observations:

Usually when the data is being fetched from database, there will not be any output. Insect situation we would like to print Loading as shown above. This can be achieved by keeping state information os isPending variable. Based on conditional formatting and the state information we can display wither Loading or the actual data fetched from the database.

Experiment 18 : Handling Fetch Errors

```
import { useState, useEffect } from "react";
import BlogList from './BlogList';

const Home = () => {

    const [blogs,setBlogs]=useState(null);
    const [isPending, setIsPending]=useState(true);
    const [error,setError]=useState(null);

    useEffect(() => {
       setTimeout( () => {
        fetch('http://localhost:8000/blogs')
       .then(res => {
            if (!res.ok){
                throw Error("Could not fetch data from the resource");
            }
            return res.json();
       })
       .then(data => {
        setBlogs(data);
        setIsPending(false);
       })
       .catch(err => {
        setIsPending(false)
        setError(err.message)
       })
       }, 5000);
    },[]);

    return (
```

```
        <div className="home">
          { error && <div> {error} </div>}
          { isPending && <div>Loading... </div> }
          { blogs && <BlogList blogs={blogs} title="All Blogs!"/> }
        </div>
      );
}
export default Home;
```



Experiment 19 : Custom Hooks

Home.js

```
import BlogList from './BlogList';
import useFetch from './useFetch';

const Home = () => {

    const {data, isPending, error} =
useFetch('http://localhost:8000/blogs')

    return (
        <div className="home">
         { error && <div> {error} </div>}
         { isPending && <div>Loading... </div> }
         { data && <BlogList blogs={data}
title="All Blogs!"/> }
        </div>
     );
}
export default Home;
```

Custom Hooks should always start with the word ‘use’ like in this example useFetch. By using a separate JS code for fetch, it can be reused with different URL endpoints.

Output of this chapters code is same as earlier chapters.

useFetch.js

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

const useFetch = (url) => {

    const [data,setData]=useState(null);
    const [isPending, setIsPending]=useState(true);
    const [error,setError]=useState(null);

    useEffect(() => {
       setTimeout( () => {
        fetch(url)
       .then(res => {
            if (!res.ok){
                throw Error("Could not fetch");
            }
            return res.json();
       })
       .then(val => {
        setData(val);
        setIsPending(false);
       })
       .catch(err => {
        setIsPending(false)
        setError(err.message)
       })
       }, 5000);
    },[url]);
    return {data, isPending, error}
}
export default useFetch
```

Experiment 20 : The React Router

React router is used to navigate between multiple pages. In a typical non-react websites, browser make a request and then sever responds using HTML pages. For any routing, we usually have a separate response from Server. But react applications does not work like that. Browser handles the routing between pages. It starts in the similar way of making a request from browser to server. In response along with HTML files, server send compiled react javascript files which controls our react application. From this point onwards react and react router takes complete control of react application. Initially HTML page is empty, then react injects the content dynamically using react components. It means we are having less number of times requesting server to provide the response and therefore the response time is quick.

First we need to install react router using below command.

npm install react-router-dom@5 (5 can be changed to newer versions)

Run the below command to connect the JSON server to fetch data

npx json-server --watch data/db.json --port 8000

Run **npm start** to see the result in http://localhost:3000

Experiment 20 : Exact Match Routes

When we have multiple routes we place them inside switch which allows only one of them to route based on the path. If we don't use keyword 'exact' in route "/create" url will not be rendered even if we request it, instead it only renders "/"

App.js

```
import './App.css';
import Navbar from './Navbar.js';
import Home from './Home.js';
import { BrowserRouter as Router, Route, Switch} from 'react-router-dom';
import Create from './Create.js';

function App() {
  return (
    <Router>
    <div className="App">
      <Navbar/>
      <div className="content">Hello World</div>
      <Switch>
        <Route exact path="/">
        <Home />
        </Route>
        <Route path="/create">
        <Create />
        </Route>
      </Switch>
    </div>
    </Router>
  );}
export default App;
```

Create.js

```
const Create = () => {
    return (

<div className="create">
<h2> Add a new blog </h2>
</div>

     );
}

export default Create;
```

All other files like Home.js, useFetch.js has teh same content as previous experiments.

Experiment 21 : Router Links

In non-react applications when ever we invoke a new url, the request goes to the server and responds with an HTML file. In react, Link component directs to the new url without sending it to server. Unlike anchor tags (<a>), Link component does not have href, instead it has "to" attribute through which it direct to specified URL. As there is no server communication, usually the response would be faster. Change the code in **Navbar.js** as follows.

```
import {Link} from 'react-router-dom';
const Navbar = () => {
    return (
        <nav className="navbar">
            <h1>My App </h1>
            <div className="links">
                <Link to="/"> Home </Link>
                <Link to="/create"> New Blog </Link>
            </div>
        </nav>
     );
}

export default Navbar;
```

Experiment 22 : Router Parameters

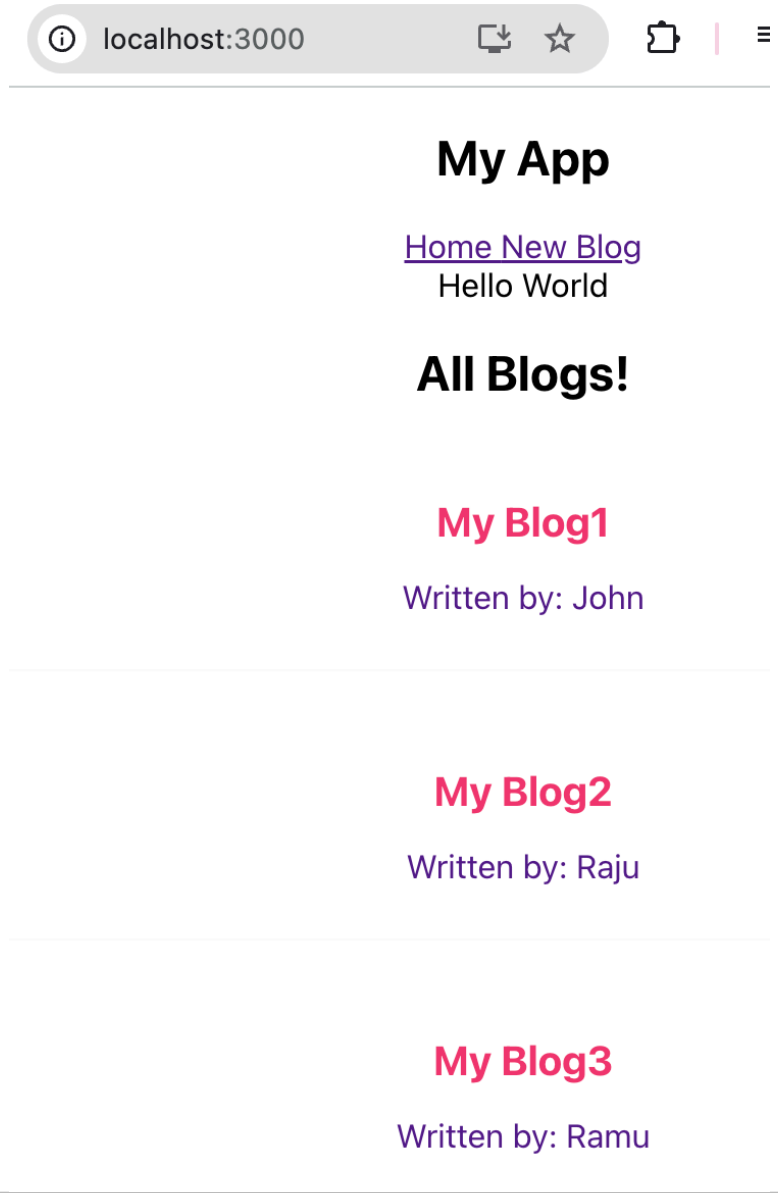
BlogList.js

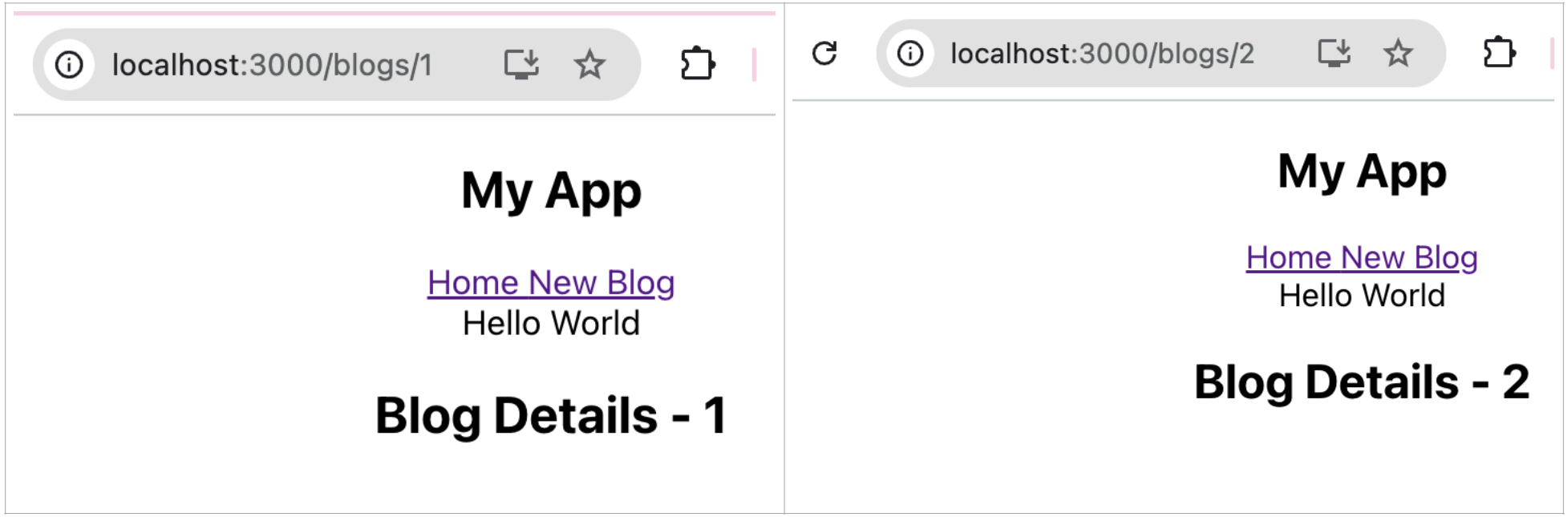
```
import {Link} from 'react-router-dom';
const BlogList = ({blogs, title}) => {
    return (
        <div className="blog-list">
            <h2>{title}</h2>
            {blogs.map((blog) => (
            <div className="blog-preview" key={blog.id}
>

            <Link to={`/blogs/${blog.id}`}>
                <h2>{blog.title}</h2>
                <p>Written by: {blog.author}</p>
            </Link>

            </div>
         ))
         }
        </div>
      );
}
export default BlogList;
```

BlogDetails.js

```
import { useParams } from "react-router-
dom";
const BlogDetails = () => {
    const {id} = useParams();
    return (
        <div className="blog-details">
            <h2> Blog Details - {id} </h2>
        </div>
     );
}
export default BlogDetails;
```

```
import './App.css';
import Navbar from './Navbar.js';
import Home from './Home.js';
import { BrowserRouter as Router, Route, Switch} from
'react-router-dom';
import Create from './Create.js';
import BlogDetails from './BlogDetails.js';

function App() {
  return (
    <Router>
    <div className="App">
      <Navbar/>
      <div className="content">Hello World</div>
      <Switch>
        <Route exact path="/">
        <Home />
        </Route>
        <Route path="/create">
        <Create />
        </Route>
        <Route path="/blogs/:id">
        <BlogDetails/>
        </Route>
      </Switch>
    </div>
    </Router>
  );
}
export default App;
```





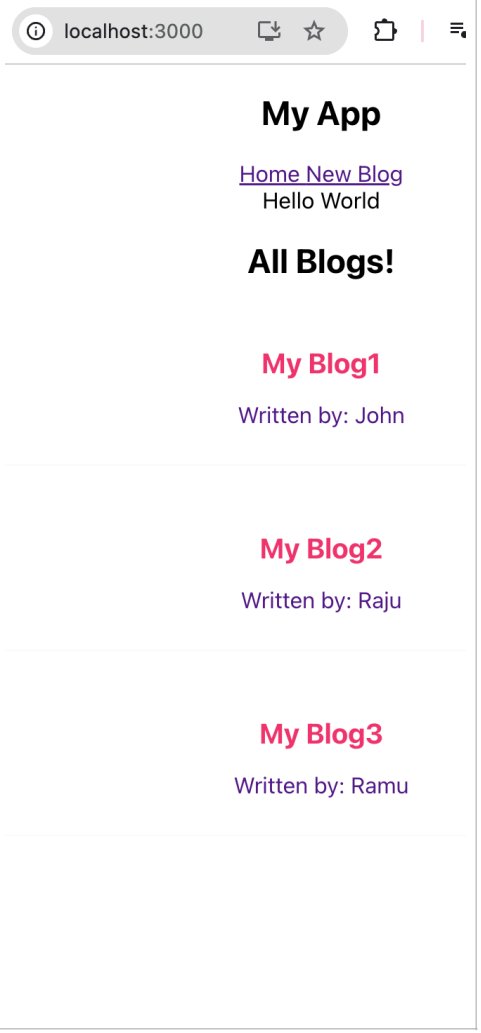
In **BlogList.js** we used Link component to create url for each blog and directed to a particular URL using an id. useParams allows us to grab the route parameters from the routing url. The parameter id is used in **BlogDetails.js** to construct the template.

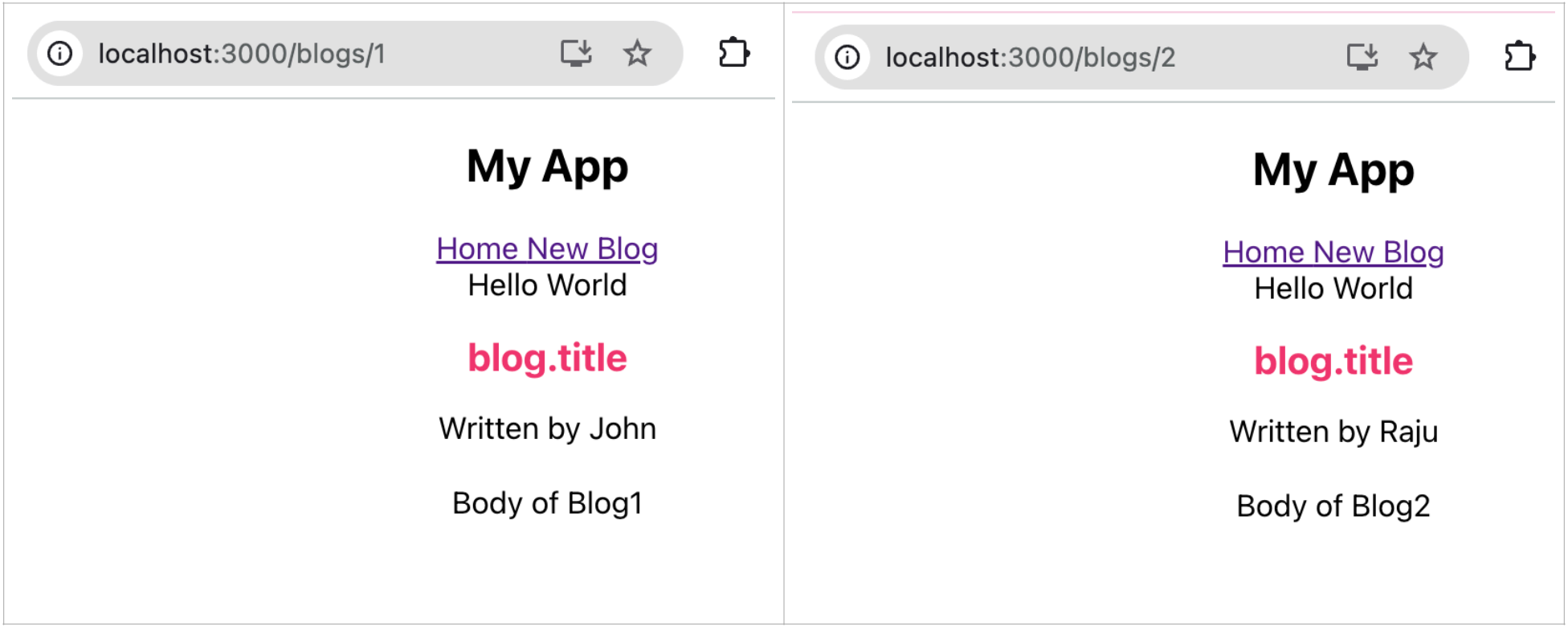
Experiment 22 : Reusing Custom Hooks

BlogDetails.js

```
import { useParams } from "react-router-dom";
import useFetch from "./useFetch";
const BlogDetails = () => {
    const {id} = useParams();

    const {data:blog, error, isPending}=useFetch('http://
localhost:8000/blogs/'+id)

    return (
        <div className="blog-details">
            {isPending && <div> Loading... </div>}
            {error && <div>{error} </div>}
            {blog && (
                <article>
                    <h2> blog.title</h2>
                    <p> Written by {blog.author} </p>
                    <div> {blog.body} </div>
                </article>
            )}


        </div>
     );
}
export default BlogDetails;
```

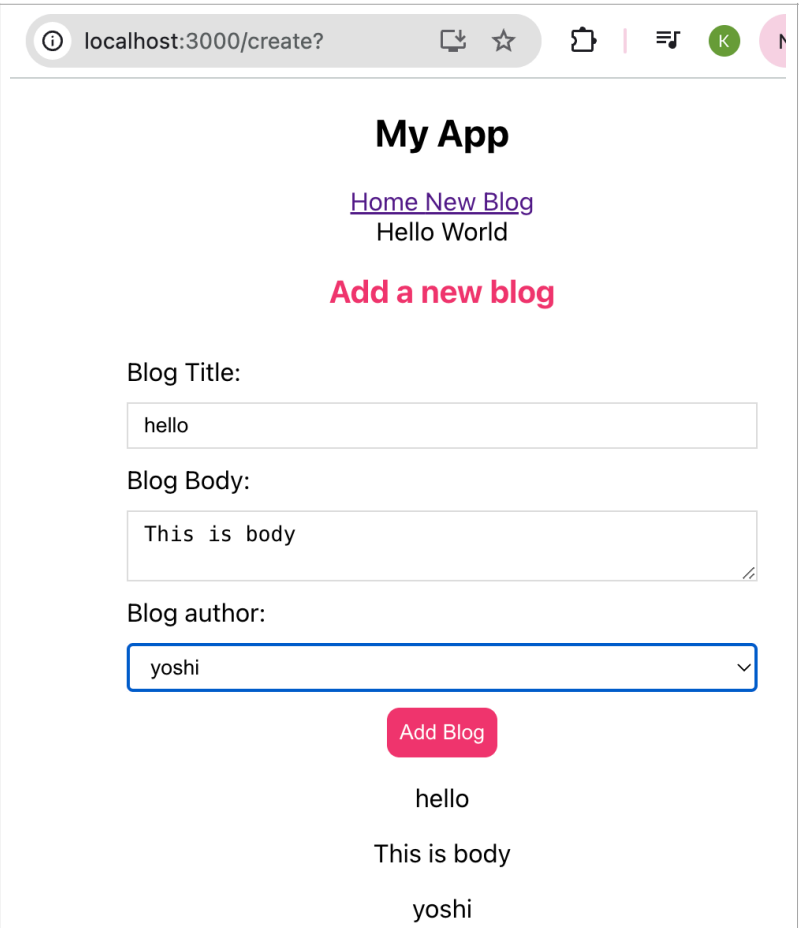
Output




localhost:3000/blogs/1
My App
Home New Blog
Hello World
blog.title
Written by John
Body of Blog1
localhost:3000/blogs/2
My App
Home New Blog
Hello World
blog.title
Written by Raju
Body of Blog2

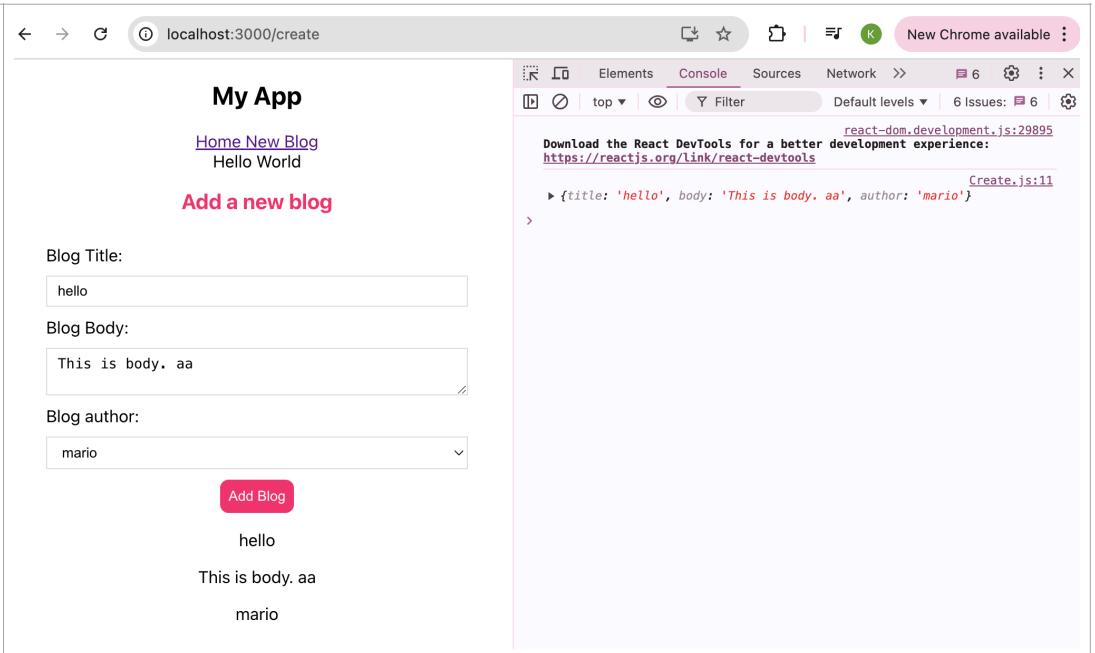
Experiment 22 : Controlled Inputs (Forms)

```
import {useState} from 'react';
const Create = () => {

    const [title,setTitle]=useState('hello');
    const [body,setBody]=useState('This is body');
    const [author,setAuthor]=useState('Raju');
    return (
        <div className="create">
            <h2> Add a new blog </h2>
            <form>
                <label>Blog Title:</label>
                <input type="text"
                required
                value={title}
                onChange={(e) => setTitle(e.target.value)}/>

                <label>Blog Body:</label>
                <textarea required value={body}
                onChange={(e) => setBody(e.target.value)}
                ></textarea>

                <label>Blog author:</label>
                <select value={author}
                    onChange={(e) => setAuthor(e.target.value)}>
                    <option value="mario"> mario</option>
                    <option value="yoshi"> yoshi</option>
                </select>
                <button>Add Blog</button>
                <p>{title}</p>
                <p>{body}</p>
                <p>{author}</p>
            </form>
        </div>
    );}
export default Create;
```



Experiment 23 : Submit Events

```
import {useState} from 'react';
const Create = () => {
    const [title,setTitle]=useState('hello');
    const [body,setBody]=useState('This is body');
    const [author,setAuthor]=useState('Raju');

    const handleSubmit=(e) => {
        e.preventDefault();
        const blog={title, body, author};
        console.log(blog)
    }
    return (
        <div className="create">
            <h2> Add a new blog </h2>
            <form onSubmit={handleSubmit}>
                <label>Blog Title:</label>
                <input
                type="text"
                required
                value={title}
                onChange={(e) => setTitle(e.target.value)}/>

                <label>Blog Body:</label>
                <textarea required
                value={body}
                onChange={(e) => setBody(e.target.value)}
                ></textarea>

                <label>Blog author:</label>
                <select
                    value={author}
                    onChange={(e) => setAuthor(e.target.value)}>
                    <option value="mario"> mario</option>
                    <option value="yoshi"> yoshi</option>
                </select>
                <button>Add Blog</button>
                <p>{title}</p>
                <p>{body}</p>
                <p>{author}</p>
            </form>
        </div>

     );}
export default Create;
```



```
import {useState} from 'react';
const Create = () => {

    const [title,setTitle]=useState('hello');
    const [body,setBody]=useState('This is body');
    const [author,setAuthor]=useState('Raju');
    const [isPending,setIsPending]=useState(false);

    const handleSubmit=(e) => {
        e.preventDefault();
        const blog={title, body, author};

        setIsPending(true);

        fetch('http://localhost:8000/blogs',{
        method:'POST',
        headers: {"Content-Type": "application/json"},
        body: JSON.stringify(blog)
        }).then(() => {
            console.log("New Blog Added");
            setIsPending(false);
        })
    }

    return (
```

```
    <div className="create">
            <h2> Add a new blog </h2>
            <form onSubmit={handleSubmit}>
                <label>Blog Title:</label>
                <input
                type="text"
                required
                value={title}
                onChange={(e) => setTitle(e.target.value)}
                />
                <label>Blog Body:</label>
                <textarea required
                value={body}
                onChange={(e) => setBody(e.target.value)}
                ></textarea>

                <label>Blog author:</label>
                <select
                    value={author}
                    onChange={(e) =>
setAuthor(e.target.value)}>
                    <option value="mario"> mario</option>
                    <option value="yoshi"> yoshi</option>
                </select>
                {!isPending && <button>Add Blog</button> }
                {isPending && <button disabled>Adding
Blog...</button> }
                <p>{title}</p>
                <p>{body}</p>
                <p>{author}</p>
            </form>
        </div>

    );
}
export default Create;
```

Experiment 24 Programatic Redirects using useHistory

Create.js

```
import {useState} from 'react';
import { useHistory } from 'react-router-dom';


const Create = () => {

    const [title,setTitle]=useState('hello');
    const [body,setBody]=useState('This is body');
    const [author,setAuthor]=useState('Raju');
    const [isPending,setIsPending]=useState(false);
    const history=useHistory();

    const handleSubmit=(e) => {
        e.preventDefault();
        const blog={title, body, author};

        setIsPending(true);

        fetch('http://localhost:8000/blogs',{
        method:'POST',
        headers: {"Content-Type": "application/json"},
        body: JSON.stringify(blog)
        }).then(() => {
            console.log("New Blog Added");
            setIsPending(false);
            // history.go(-1);
            history.push('/');
        })
    }
```

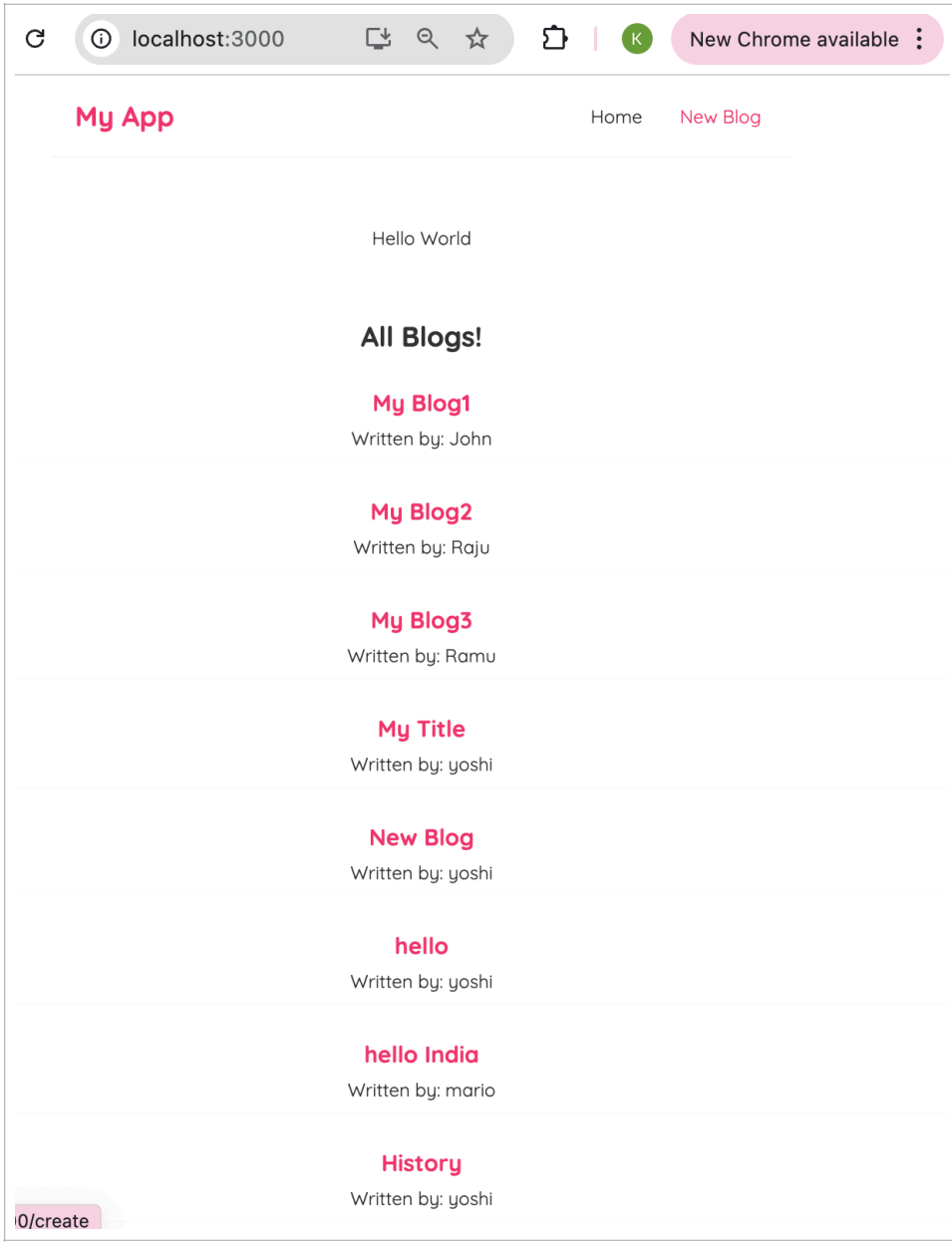
```
 return (
        <div className="create">
            <h2> Add a new blog </h2>
            <form onSubmit={handleSubmit}>
                <label>Blog Title:</label>
                <input
                type="text"
                required
                value={title}
                onChange={(e) => setTitle(e.target.value)}
                />

                <label>Blog Body:</label>
                <textarea required
                value={body}
                onChange={(e) => setBody(e.target.value)}

                ></textarea>

                <label>Blog author:</label>
                <select
                    value={author}
                    onChange={(e) => setAuthor(e.target.value)}>
                    <option value="mario"> mario</option>
                    <option value="yoshi"> yoshi</option>
                </select>
                {!isPending && <button>Add Blog</button> }
                {isPending && <button disabled>Adding Blog...</
button> }
                <p>{title}</p>
                <p>{body}</p>
                <p>{author}</p>
            </form>
        </div>

    );
}
export default Create;
```

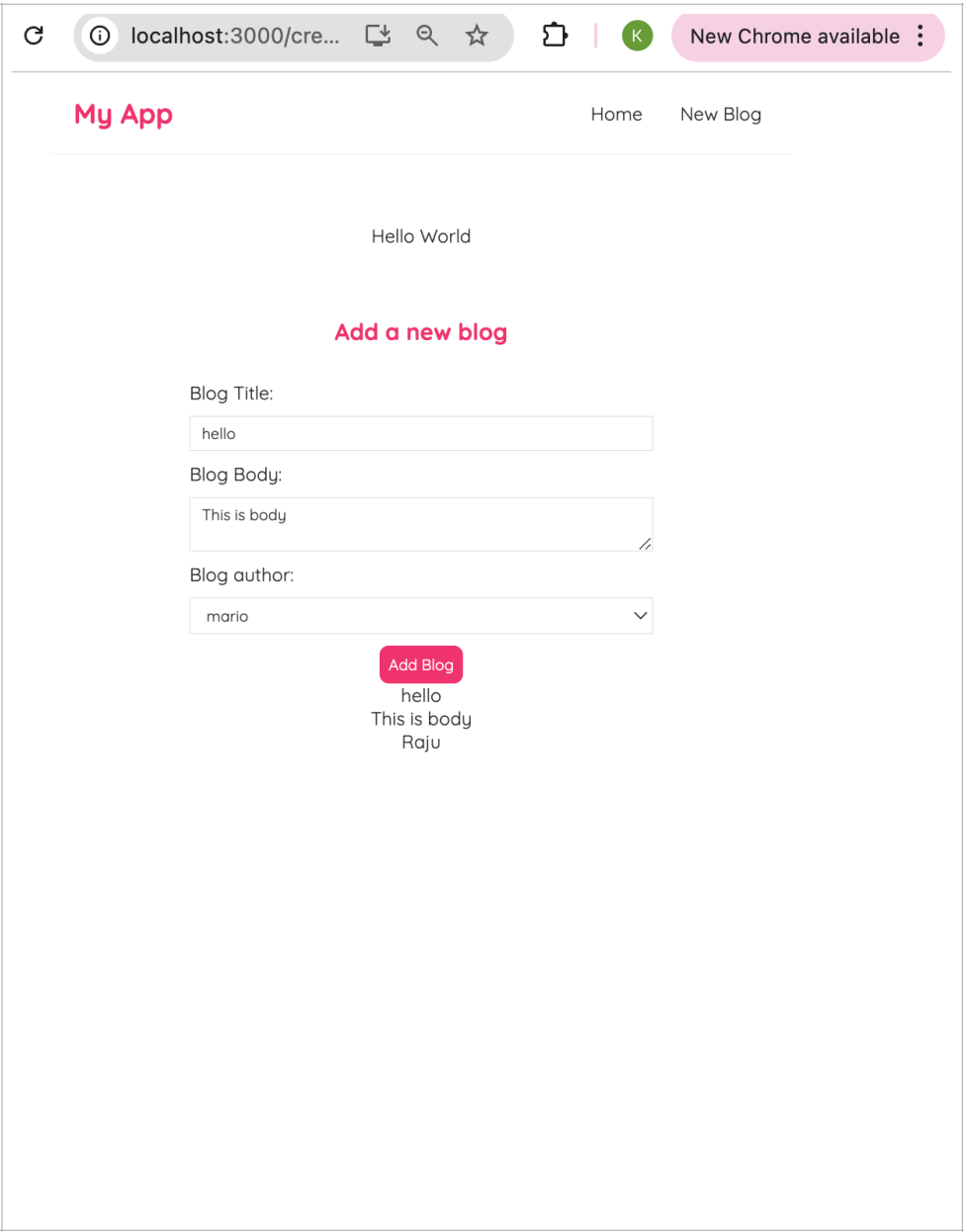
index.css

```css
@import url('https://fonts.googleapis.com/css2?family=Quicksand:wght@300;400;500;600;700&display=swap');

/* base styles */
* {
  margin: 0;
  font-family: "Quicksand";
  color: #333;
}
.navbar {
  padding: 20px;
  display: flex;
  align-items: center;
  max-width: 600px;
  margin: 0 auto;
  border-bottom: 1px solid #f2f2f2;
}
.navbar h1 {
  color: #f1356d;
}
.navbar .links {
  margin-left: auto;
}
.navbar a {
  margin-left: 16px;
  text-decoration: none;
  padding: 6px;
}
.navbar a:hover {
  color: #f1356d;
}
.content {
  max-width: 600px;
  margin: 40px auto;
  padding: 20px;
}
```

```css
/* blog previews / list */

.blog-preview {
  padding: 10px 16px;
  margin: 20px 0;
  border-bottom: 1px solid #fafafa;
}
.blog-preview:hover {
  box-shadow: 1px 3px 5px rgba(0,0,0,0.1);
}
.blog-preview h2 {
  font-size: 20px;
  color: #f1356d;
  margin-bottom: 8px;
}
.blog-preview a{
  text-decoration: none;
}

.blog-details h2{
  font-size: 20px;
  color: #f1356d;
  margin-bottom: 10px;
}

.blog-details div{
  margin: 20px 0;
}

.blog-details button {
  background: #f1356d;
  color: #fff;
  border: 0;
  padding: 8px;
  border-radius: 8px;
  cursor: pointer;
}
```

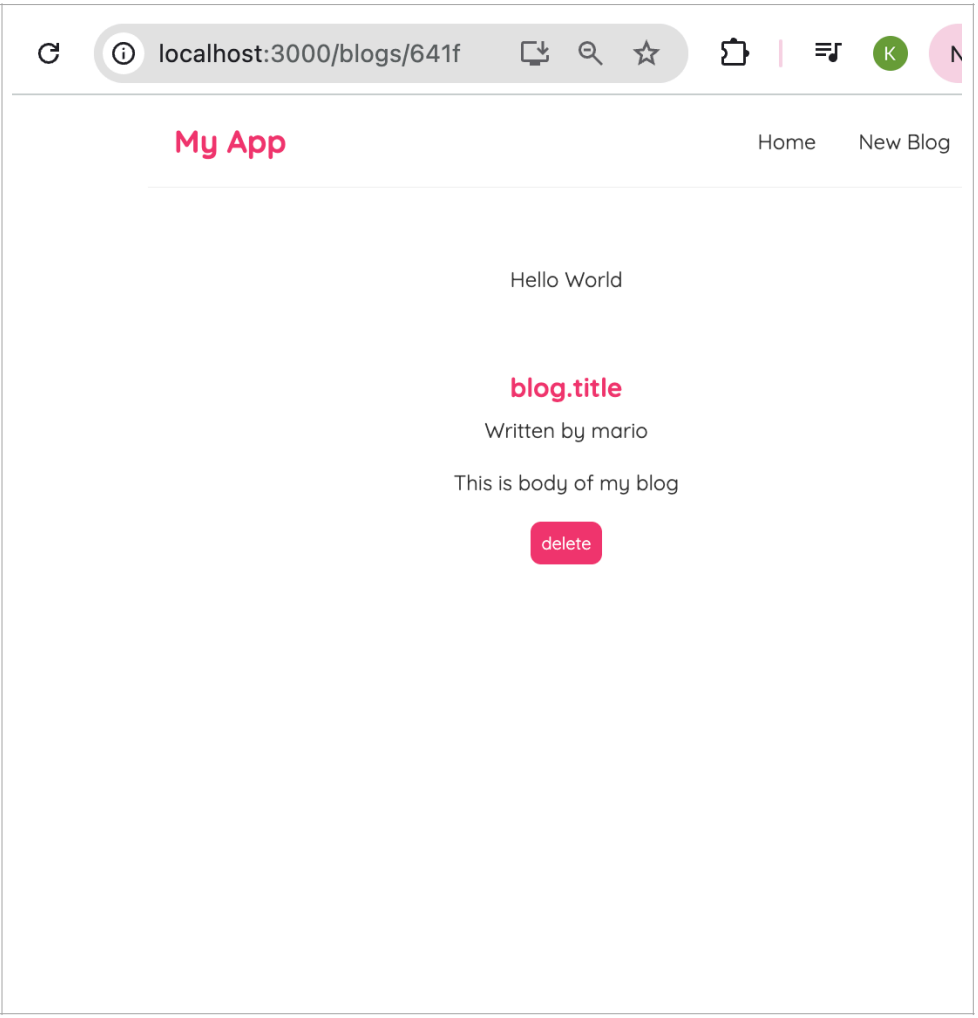
```css
/* create new blog form */
.create {
  max-width: 400px;
  margin: 0 auto;
  text-align: center;
}
.create label {
  text-align: left;
  display: block;
}
.create h2 {
  font-size: 20px;
  color: #f1356d;
  margin-bottom: 30px;
}
.create input, .create
textarea, .create select {
  width: 100%;
  padding: 6px 10px;
  margin: 10px 0;
  border: 1px solid #ddd;
  box-sizing: border-box;
  display: block;
}
.create button {
  background: #f1356d;
  color: #fff;
  border: 0;
  padding: 8px;
  border-radius: 8px;
  cursor: pointer;
}
```


localhost:3000
New Chrome available
My App
Home
New Blog
Hello World
All Blogs!
My Blog1
Written by: John
My Blog2
Written by: Raju
My Blog3
Written by: Ramu
My Title
Written by: yoshi
New Blog
Written by: yoshi
hello
Written by: yoshi
hello India
Written by: mario
History
Written by: yoshi
0/create


localhost:3000/cre...
New Chrome available
My App
Home
New Blog
Hello World
Add a new blog
Blog Title:
hello
Blog Body:
This is body
Blog author:
mario
Add Blog
hello
This is body
Raju

Experiment 25 Delete Blogs

```
import { useParams } from "react-router-dom";
import useFetch from "./useFetch";
import { useHistory } from "react-router-dom";
const BlogDetails = () => {
    const {id} = useParams();
    const history=useHistory();

    const {data:blog, error, isPending}=useFetch('http://
localhost:8000/blogs/'+id)

    const handleClick = () => {
        fetch('http://localhost:8000/blogs/'+blog.id,{
            method:'DELETE'
        }).then(()=>{
            history.push('/')
        })
    }

    return (
        <div className="blog-details">
            {isPending && <div> Loading... </div>}
            {error && <div>{error} </div>}
            {blog && (
                <article>
                    <h2> blog.title</h2>
                    <p> Written by {blog.author} </p>
                    <div> {blog.body} </div>
                    <button onClick={handleClick}>delete</button>
                </article>
            )}



        </div>
    );
}
export default BlogDetails;
```



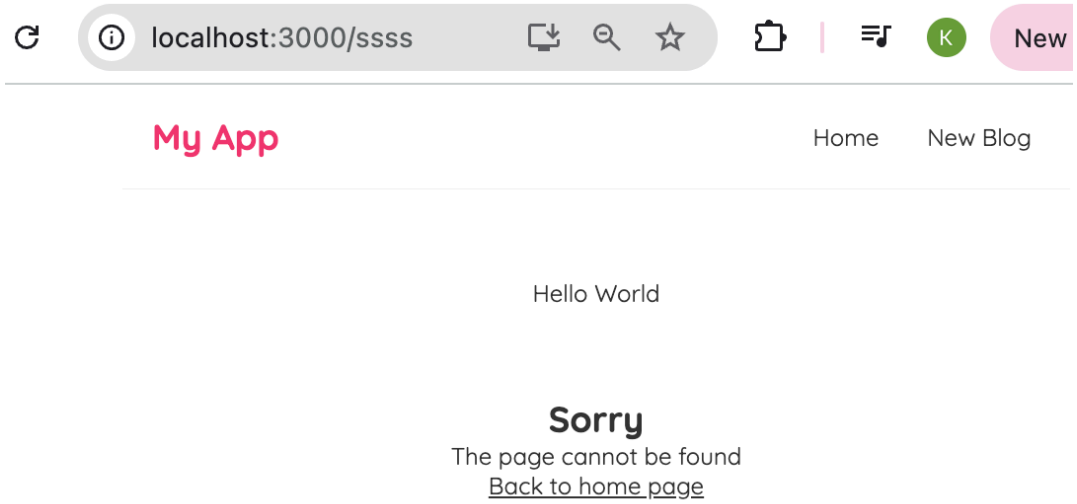
Experiment 25 Page Not Found (404)

App.js

```
import './App.css';
import Navbar from './Navbar.js';
import Home from './Home.js';
import { BrowserRouter as Router, Route, Switch}
from 'react-router-dom';
import Create from './Create.js';
import BlogDetails from './BlogDetails.js';
import Notfound from './Notfound.js';

function App() {
  return (
    <Router>
    <div className="App">
      <Navbar/>
      <div className="content">Hello World</div>
      <Switch>
        <Route exact path="/">
        <Home />
        </Route>
        <Route path="/create">
        <Create />
        </Route>
        <Route path="/blogs/:id">
        <BlogDetails/>
        </Route>
        <Route path="*">
        <Notfound />
        </Route>

      </Switch>
    </div>
    </Router>
  );
}
export default App;
```

Notfound.js

```
import { Link } from "react-router-dom/cjs/react-router-dom.min";

const Notfound = () => {
    return (
        <div className="not-found">
            <h2> Sorry</h2>
            <p>The page cannot be found </p>
            <Link to="/">Back to home page</Link>
        </div>
     );
}

export default Notfound
```



Experiment 26 Node Basics

Javascript usually runs in browser which contains V8 engine. This engine written in C++ can run javascript code. But if we want to run javascript in a computer, we can do so using node. Node internally contains V8 engine.

You can install node. If its already installed you can check the version using following terminal command:

```
node -v
```

If you type just node in terminal you can type and test the javascript commands.

```
node
```

Press **ctrl+c** two times to come out of the node in terminal.

Create directory to place all the .js files as follows.

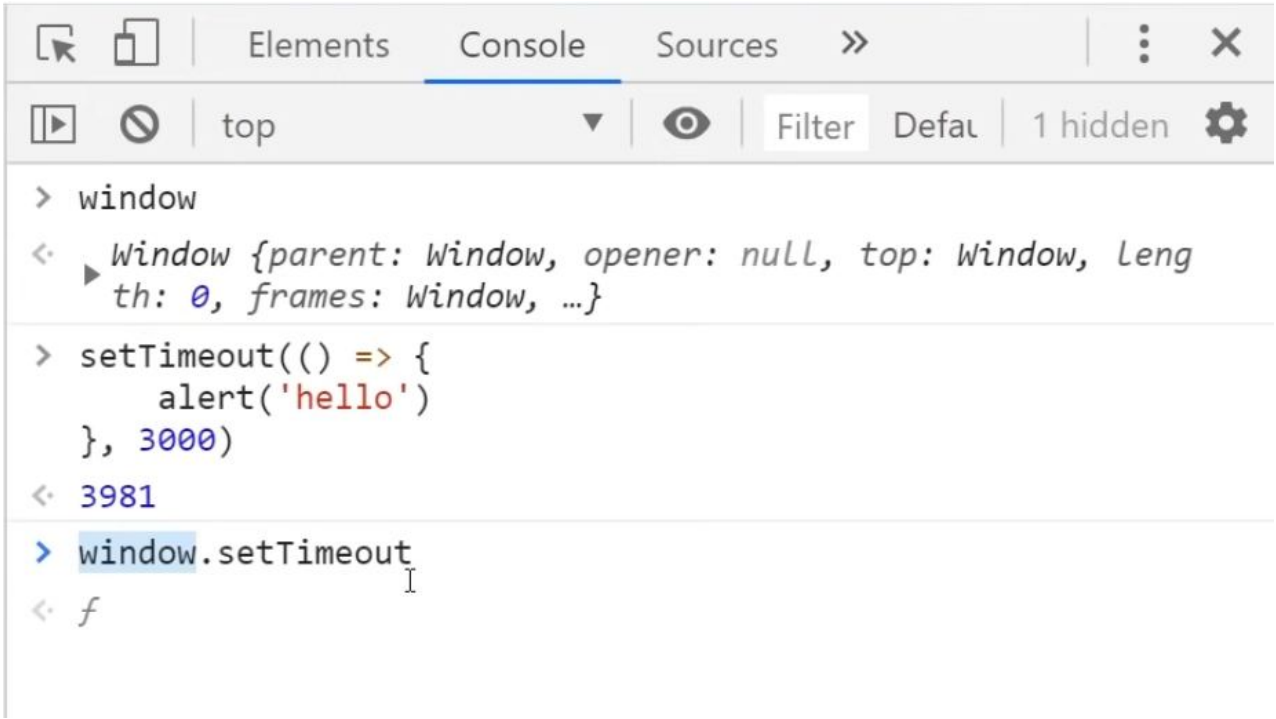
```
mkdir node-crash-course
cd node-crash-course
```

Open visual studio code and open the **node-crash-course** directory and right click to create **test.js** file. Type the following javascript commands and run in terminal using following terminal command.

Terminal Command	test.js

`node test.js`	`const myName = 'mario';`
	
	`console.log(myName);`

In browser window is the global object, but in node window is not global object instead global is.



In node we can type **console.log(global)** to access all methods which are part of window. Just like how window is not needed to retyped explicitly, we don't need to type global when we access their methods like as shown below.

test.js	Terminal Output

setTimeout(()=> {
  console.log('in the timeout');
  clearInterval(int);
},3000)
 	
const int = setInterval(()=> {
  console.log('in the interval');
}, 1000);
	in the interval
	in the interval
	in the timeout

```
console.log(__dirname);
console.log(__filename);
```

Above commands give the absolute path of the current folder and the later one gives folder path with file name appended as well.

Output in mac:

```
/Users/sandeepchitreddy/MERN/node-crash-course
/Users/sandeepchitreddy/MERN/node-crash-course/test.js
```

Global object in node is different from global object in window of browser. Some of the things we cannot access in node like document.queryselector etc. document is only part of browser window object and not node global object. But any ways we don't need them as node is used for backend and not for browser.

Experiment 26 Node Basics: Modules and Require

module.js	people.js	Terminal command & Output

`const xyz = require('./people.js')`
 	
`console.log(xyz)`
	`const people = ['ram','sita','raju'];`
	
	`console.log(people);`
	
	`module.exports = people;`
		node module
		
		['ram', 'sita', 'raju']
		['ram', 'sita', 'raju']

module.js	people.js

`const {people, ages} = require('./people.js')`
 	
`console.log(people, ages);`
	`const people = ['ram','sita','raju'];`
	`const ages = [20, 25, 30, 35];`
	
	`console.log(people);`
	
	`module.exports = { people, ages };`

module.js	Output

`const os = require('os');`
 	
`console.log(os.platform(), os.homedir());`
	`In MAC: darwin /Users/sandeepchitreddy`

Apart from os module, one other important module is filesystem that will be discussed in next section.

Experiment 27 Node Basics: File System

In this section we will see how we can read, write, delete text files and create directories from the computer using node. Create a text file with arbitrary content in it and name it as **blog1.txt.**

Read textiles:

files.js

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

fs.readFile('./blog1.txt',(err, data) => {

    if(err){
        console.log(err)
    }
    console.log(data.toString());

});

console.log('last line');
```

blog1.txt

```
Hello World
Hello India
```

Output:

```
last line
Hello World
Hello India
```

As you can see first, the last line got printed as it takes time to read data from file, and the javascript does not wait to complete it. In the mean time javascript prints other sections of the code.

Write in to text files:

Writing content to a new file can be done using following javascript code. Method writeFile that is part of file system is used to write content in to text file. If the file does not exist, it creates a new file and write it.

```
const fs = require('fs');
fs.writeFile('./blog1.txt','Hello Hyderabad',() => {
    console.log('file was written')
})
```

Creating directories:

Below code creates a directory named as assets. If the directory already exists it throws error and prints err.

```
const fs = require('fs');
fs.mkdir('./assets',(err) => {
    if (err){
        console.log('err');
    }
    console.log('folder created');
})
```

Below code checks if the assets folder already exits. If it exists it does not executes other below statements.

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

if(!fs.existsSync('./assets')){
    fs.mkdir('./assets',(err) => {
        if (err){
            console.log('err');
        }
        console.log('folder created');
    })
}
```

`fs.rmdir` can be used to remove directory

Deleting files:

If there is a file named deleteme.txt, it deletes it. If it does not exists it does not run the rest of code.

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

if(fs.existsSync('./deleteme.txt')){
    fs.unlink('./deleteme.txt',(err) => {
        if (err){
            console.log('err');
        }
        console.log('file deleted');
    })
}
```

Experiment 27 Node Basics: Streams and Buffers.

Streams: start using data before it has finished loading.

When the file is about to loaded is big and takes time, in such cases we use streams to load a part of data. Small chunks of data called as buffers are sent when the buffer is fully filled. Example is Netflix where it streams a part of video and can be played instead of waiting for whole data to be loaded.

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

const readStream = fs.createReadStream('./blog1.txt',{encoding:'utf8'});
const writeStream = fs.createWriteStream('./blog2.txt');

readStream.on('data', (chunk) => {
    console.log('\n... New Chunk...\n')
    console.log(chunk);

    writeStream.write('\n... New Chunk...\n');
    writeStream.write(chunk);
});
```

There is shorter way to do the above code using pipe methods.

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

const readStream = fs.createReadStream('./blog1.txt',{encoding:'utf8'});
const writeStream = fs.createWriteStream('./blog2.txt');

readStream.pipe(writeStream);
```

Experiment 28 Node Basics: Create Server

Unlike PHP, where the server is typically set up through Apache or a similar web server, in Node.js, you need to explicitly create a server. This server listens for requests from browsers or clients, handling them programmatically within your application.

We import http using require. http object has createServer as one of the method. This method returns a call back function. This call back function takes request and response object. Request object carrys the information about URL, request tyep like GET or POST etc. Response object will have proper response from the server to browser.

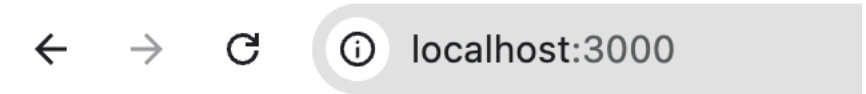
```
const http = require('http');

const server= http.createServer((req,res) => {
    console.log('request made')
})

server.listen(3000,'localhost', () => {
    console.log('listening for requests on port 3000');
});
```

```
sandeepchitreddy@Sandeeps-MacBook-Pro node-crash-course % node server
listening for requests on port 3000
request made
```

Type localhost:3000 in browser you will see the output ‘request made’ as shown above which means the server is listening to the browser’s request.

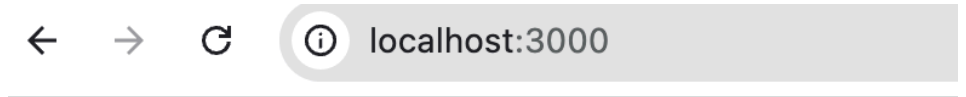
Response Object:

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

const server= http.createServer((req,res) => {
    console.log(req.url,req.method)

    res.setHeader('Content-Type','text/plain');
    res.write('Hello World');
    res.end()

})

server.listen(3000,'localhost', () => {
    console.log('listening for requests on port
3000');
});
```



Hello World

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

const server= http.createServer((req,res) => {
    console.log(req.url,req.method)

    res.setHeader('Content-Type','text/html');
    res.write('<h2> Hello World</h2>');
    res.end()

})

server.listen(3000,'localhost', () => {
    console.log('listening for requests on port
3000');
});
```



Hello World

References:

https://www.youtube.com/playlist?list=PL4cUxeGkcC9gZD-Tvwfod2gaISzfRiP9d

React Course Material of Net Ninjas: https://github.com/iamshaunjp/Complete-React-Tutorial