# Component States and Life Cycle Methods

## Introduction to Component States

The state is an instance of React Component Class can be defined as an object of a set of observable properties that control the behavior of the component. In other words, the State of a component is an object that holds some information that may change over the lifetime of the component. t **stores a component's dynamic data** and determines the component's behaviour. Because state is dynamic, it enables a component to keep track of changing information in between renders and for it to be dynamic and interactive.

## Defining the state

State can only be accessed and modified inside the component and directly by the component. Must first have some **initial state**, should define the state in the constructor of the component's class.

```
class MyClass extends React.Component {
        constructor(props)  {
        super(props);  this.state = { attribute : "value" };    }
}
```

The state object can contain as many properties as you like

```
class MyClass extends React.Component {
        constructor(props)  {
        super(props);   this.state = { attribute1 : "value1", attribute2 : "value2",
attribute3 : "value3"};    }            }
```

## Changing the state

State should **never be updated explicitly**. Must use **setstate().** It takes a single parameter and expects an object which should contain the set of values to be updated. Once the update is done the method implicitly calls the render() method to repaint the page.

```
this.setState({attributen : "valuen", attribute2 : "newvalue1", })
```

## Differences between state and props

- States are mutable and Props are immutable
- States can be used in Class Components, Functional components with the use of React Hooks (useState and other methods) while Props don't have this limitation.
- Props are set by the parent component. State is generally updated by event handlers

**Example code 1: Changing the state based on the click on the button**

```
<body>
<div id = "root"> </div>
<script type = "text/babel">
    class Hello extends React.Component {
        constructor(props)      {
                super(props)
                this.state = {name:"sindhu",address:"nagarbavi",phno:"9876554"     }
                //this.updatestate = this.updatestate.bind(this) // if the function has no =>
        }
        render(){
                return (<div><h1>hello {this.state.name}</h1>
                <p> u r from {this.state.address} and ua contact number is
                {this.state.phno}</p>
                <button onClick = {this.updatestate}>click here to change state</button>
                </div> )
        }
        updatestate =()=>       {
                this.setState({name:"pai"})         }
    }
  ReactDOM.render(<Hello/>, document.getElementById("root"))
</script> </body>
```

**Example code 2:**Generate the Digital Clock using ReactJS

```
<script type = "text/babel">
        class Clock extends React.Component{
        constructor(){
                super()
                this.state = {time: new Date()}
                //this.tick = this.tick.bind(this)
        }
        //tick()
        tick = () => {          this.setState({time:new Date()})          }
        render(){   setInterval(this.tick,1000)
                    return (<h1>{this.state.time.toLocaleTimeString()}</h1>)
        }
       }
        ReactDOM.render(<Clock/>, document.getElementById("root"))
</script>
```

**toLocaleTimeString():** Returns the time portion of a Date object as a string, using locale conventions.

**Few points to think:**

- What happens if you use setInterval directly inside class outside all functions?
- Calling the setInterval in render() is not a good idea as conventionally render function is used only to render the component . Then which is the best place to have this setInterval function call?

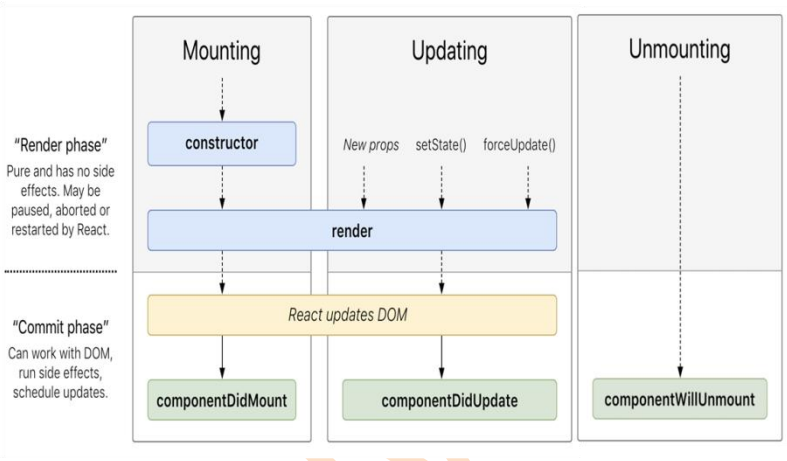
## Life Cycle Methods

The series of events that happen from the starting of a React component to its ending. Every component in React should go through the following lifecycle of events.

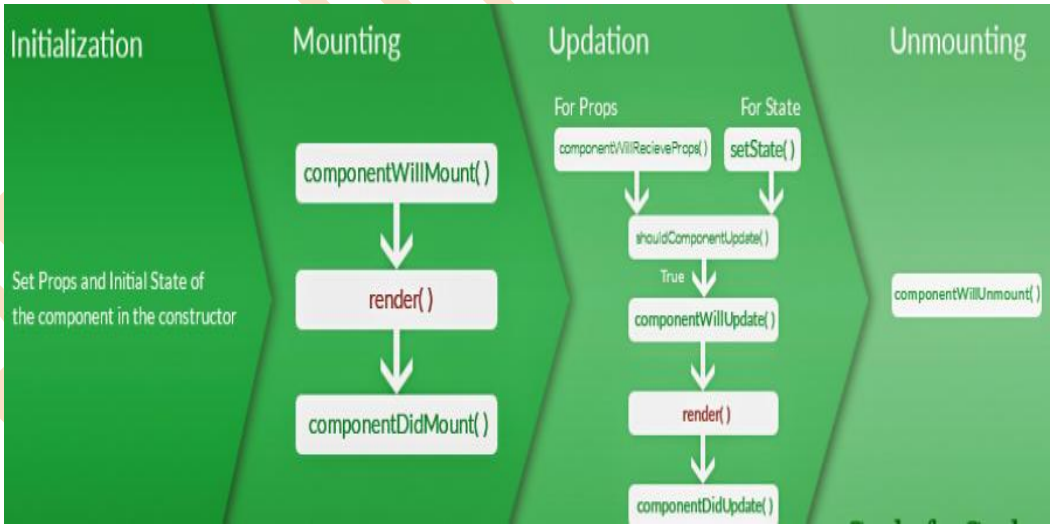
**Mounting - Birth of the Component**

**Updating- Growing of component**

**Unmounting- End of the component**




Functions defined in every phase is more clear with the below diagram




## Functions/Methods in detail

### 1. constructor() : Premounting

This function is called before the component is mounted. Implementation requires

calling of super() so that we can execute the constructor function that is inherited from React.Component while adding our own functionality. Supports Initializing the state and binding our component

```
        constructor() {
  super()
  this.state = {
    key: "value"
  }
}
```

It is possible to use the constructor to set an initial state that is dependent upon props. Otherwise, this.props it will be undefined in the constructor, which can lead to a major error in the application.

```
constructor(props) {
  super(props);
  this.state = {
    color: props.initialColor
  };
}
```

**2. componentWillMount()**

This is called only once in the component lifecycle, immediately before the component is rendered. Executed before rendering, on both the server and the client side. Suppose you want to keep the time and date of when the component was created in your component state, you could set this up in componentWillMount.

```
componentWillMount() {
  this.setState({ startDateTime: new Date(Date.now())});
}
```

**3. render()**

Most useful life cycle method as it is the only method that is required. Handles the rendering of component while accessing this.state and this.props

4. **componentDidMount()**

Function is called once only, but immediately after the render() method has taken place. That means that the HTML for the React component has been rendered into the DOM and can be accessed if necessary. This method is used to perform any DOM

manipulation of data-fetching that the component might need.

The best place to initiate API calls in order to fetch data from remote servers. Use setState which will cause another rendering but It will happen before the browser updates the UI. This is to ensure that the user won't see the intermediate state. AJAX requests and DOM or state updates should occur here. Also used for integration with other JavaScript frameworks like Node.js and any functions with late execution such as **setTimeout** or **setInterva**l

**5. componentWillReceiveProps()**

Allows us to match the incoming props against our current props and make logical. We get our current props by calling this.props and the new value is the nextProps argument passed to the method. It is invoked as soon as the props are updated before another render method is called.

6. **shouldComponentUpdate()**

Allows a component to exit the Update life cycle if there's no reason to use a replacement render. It may be a no-op that returns true. Means while updating the component, we'll re-render.

**7. componentWillUpdate()**

Called just before the rendering

**8. componentDidUpdate()**

Is invoked immediately after updating occurs. Not called for the initial render. Will not be invoked if **shouldcomponentUpdate() returns false**.

9. **componentWillUnmount()**

The last function to be called immediately before the component is removed from the DOM. It is generally used to perform clean-up for any DOM-elements or timers created in componentWillMount.

```
componentWillUnmount() {
```

```
  clearInterval(this.interval);
}
```

**Example code 3: Sequence of execution of life cycle methods**

```
<script type = "text/babel">
        class Hello extends React.Component{
                constructor(props){
                        console.log("in constructor")
                        super(props)
                        this.state={ame:"sindhu",address:"nagarbavi",phno:"9876"}
                        //this.updatestate = this.updatestate.bind(this)
                        //this.fun1 = this.fun1.bind(this)
                }
                render(){
                        console.log("in render")
                        return (<div><h1>hello {this.state.name}</h1>
                                <p> u r from {this.state.address} and ua contact number is {this.state.phno}</p>
                        <button onClick = {this.updatestate}>click here to change state</button>
                        <button onClick = {this.fun1}> click here to delete the user</button></div>
                        )
                }
                componentDidUpdate(prevProps, prevState){
                                console.log("component did update")
                }
                UNSAFE_componentWillUpdate()  {
                                console.log("will update")
                }
                componentDidMount(){
                                console.log("did mount")
```

```
        }
        UNSAFE_componentWillMount()    {
                console.log("will mount")
        }
        fun1 = () =>    {
ReactDOM.unmountComponentAtNode(document.getElementById("root"))
        }
        componentWillUnmount(){
                console.log("will unmount")
        }
        shouldComponentUpdate(nextProps, nextState) {
            console.log("yes, in shouldCOmponentupdate")
            return true;
        }
        updatestate =()=>{
            console.log("in updatestate")
            this.setState({name:"pai"})    }
        }
        ReactDOM.render(<Hello/>, document.getElementById("root"))
</script>
```

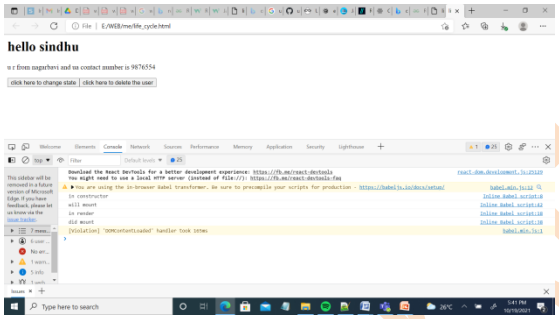
**ReactDOM.unmountComponentAtNode**(node to be deleted) : Used to delete the componenet from the DOM

**Note:** In React v16.3, componentWillMount(), componentWillReceiveProps(), and componentWillUpdate() are marked as unsafe legacy lifecycle methods for deprecation process. They have often been misused and may cause more problems with the upcoming async rendering. As safer alternatives for those methods, **getSnapshotBeforeUpdate() and getDerivedStateFromProps()** were newly added. Since the roles of the unsafe methods and the newly added methods may overlap, React prevents the unsafe methods from being called when the alternatives are defined and outputs a **warning** message in the browser. So,

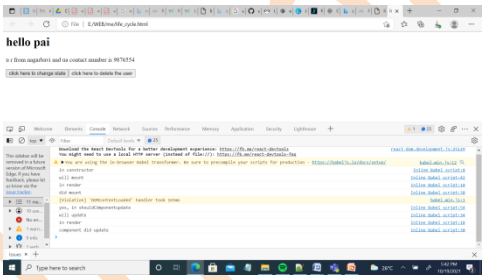
explicitly UNSAFE_ keyword is used in redefining these methods.

**Observe the output of the above code in console at every stage. Snapshot is attached in sequence.**

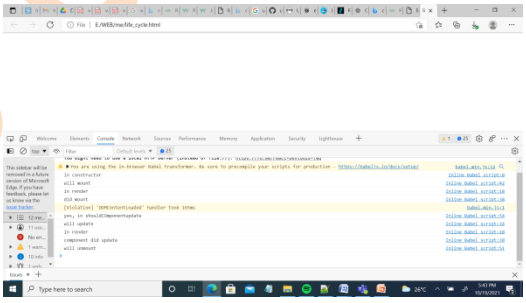
**stage 1: once the page is loaded on the browser**




**Stage 2: when the first button is clicked**




**Stage 3: When the second button is clicked.**



**Example code 4: Digital clock using life cycle method**

```
<body>
  <div id = "root"> </div>
  <script type = "text/babel">
```

```
class Clock extends React.Component{
        constructor()   {
                super()
                this.state = {time: new Date()}
                this.tick = this.tick.bind(this)
                console.log("in constructor")
        }
        tick(){
                console.log("in tick")
                this.setState({time:new Date()})
        }
        render(){
                console.log("in render")
                return (<h1>{this.state.time.toLocaleTimeString()}</h1>)
        }
        componentDidMount(){
                console.log("in did mount")
                setInterval(this.tick,1000) // Observe that added here in didmount
method of lifecycle
        }
    }
    ReactDOM.render(<Clock/>, document.getElementById("root"))
  </script>
</body>
```

## References


- ReactJS | State in React - GeeksforGeeks
- React Component Mounting And Unmounting - Learn.co
- Component Lifecycle | Build with React JS
- Rule | DeepScan

# Stateless Components

## Introduction

There exist components which do not use state. Such components just print out what is given to them via props, or they just render the same thing. For performance reasons and for clarity of code, it is recommended that such components (those that have only render() ) are written as functions rather than classes: a function that takes in props and just renders based on it. It's as if the component's view is a pure function of its props, and it is stateless. The render() function itself can be the component. If a component does not depend on props, it can be written as a simple function whose name is the component name.

Stateless components are those components which don't have any state at all, which means you can't use this.setState inside these components.It has no lifecycle, so it is not possible to use lifecycle methods. When react renders our stateless component, all that it needs to do is just call the stateless component and pass down the props.

**Note: A functional component is always a stateless component, but the class component can be stateless or stateful.**

**When would you use a stateless component??**

- When you just need to present the props
- When you don't need a state, or any internal variables
- When creating element that does not need to be interactive
- When you want reusable code

**When would you use a stateful component?**

- When building element that accepts user input
- Element that is interactive on page
- When dependent on state for rendering, such as, fetching data before rendering
- When dependent on any data that cannot be passed down as props

**Ways of creation of stateless components:**

- The first is the ES2015 arrow function style with only the return value as an expression. There are no curly braces, and no statements, just a JSX expression

```
…
    const IssueRow = (props) => ( …)
…
```

- second style, a little less concise, is needed when the function is not a single expression. The main difference is the use of curly braces to indicate that there's going to be a return value, rather than the expression within the round braces being an implicit return of that expression's result.

```
…
    function IssueTable(props) {
        …
    }
…
```

**Coding Example 1:Simple code illustration of stateless components**

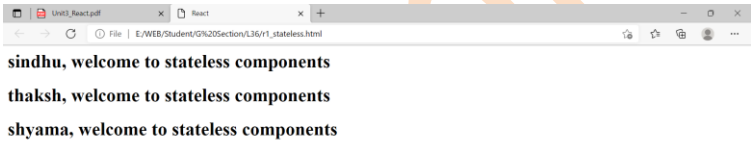
```
<div id = "root"></div>
<script type = "text/babel">
    function Sample(props)
    {       return <h1> welcome to stateless components</h1>                    }
    ReactDOM.render(<Sample/>, document.getElementById("root")
</script>
```

**Coding Example 2:Usage of props in stateless components**

```
<body>
<div id="root"></div>
<script type = "text/babel">
    function Sample(props)
    {
```

```
            return <h1> {props.name}, welcome to stateless components</h1>
        }

        ReactDOM.render(<div><Sample name = "sindhu"/>
            <Sample name = "thaksh"/>
            <Sample name = "shyama"/>
            </div>
        ,document.getElementById("root"))
    </script>
    </body>
```

**Output:**




**What if we change the rendering code as below? Is it possible to access the content of tag/componenet inside the stateless componenet creation code? If yes, how?**

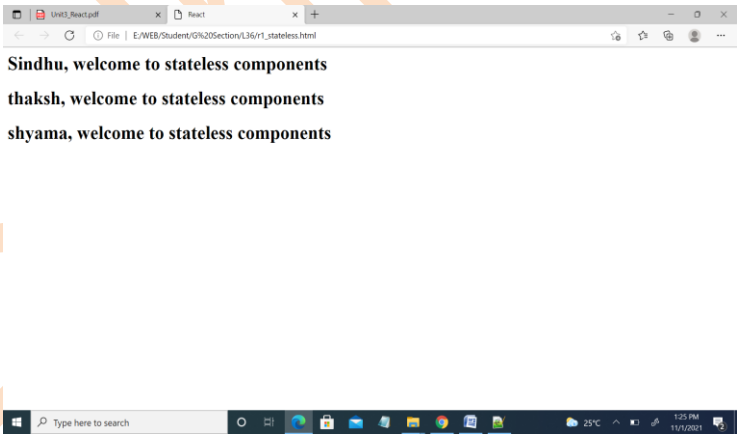
```
ReactDOM.render(<div><Sample>Sindhu</Sample>
            <Sample>thaksh</Sample>
            <Sample>shyama</Sample>
            </div>
        ,document.getElementById("root"))
```

**Answer: Possible using props.children**

**Coding Example 3:Usage of props.children in stateless components**

```
<body>
        <div id="root"></div>
        <script type = "text/babel">
                function Sample(props)
                {
                return <h1> {props.children}, welcome to stateless components</h1>
                }
                ReactDOM.render(<div><Sample>Sindhu</Sample>
                        <Sample>thaksh</Sample>
                        <Sample>shyama</Sample>
                        </div>
                ,document.getElementById("root"))
        </script>
</body>
```

**Output:**




## Nesting of stateless Components

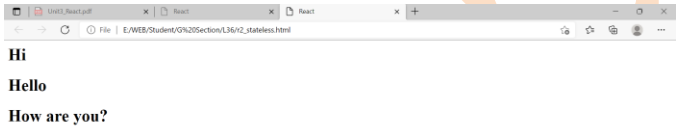
Consider the below code to understand the concept of nesting.

**Coding Example 1:**

```
<div id="root"></div>
<script type = "text/babel">
```

```
function Sample(props){
        return <Sample1>{props.text}</Sample1> }
function Sample1(props){
        return <h1>{props.children}</h1>  }
ReactDOM.render(<div><Sample     text= "Hi"/>
        <Sample text = "Hello"/>
        <Sample text = "How are you?"/>
        </div>
,document.getElementById("root"))
```

**Output:**




If we render the component with below code, what changes must be done in the above code? Think!!

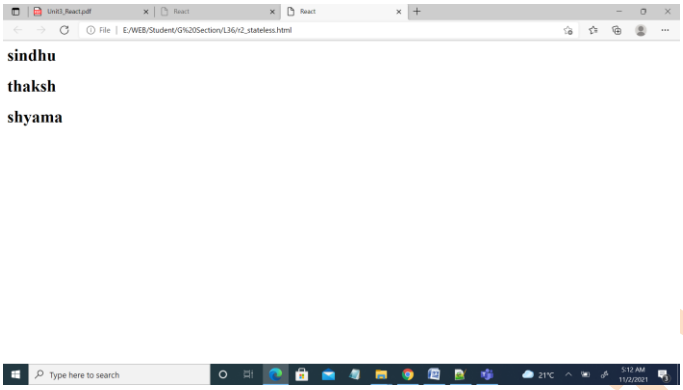
**Coding Example 2:**

```
ReactDOM.render(<div><Sample>sindhu</Sample>
        <Sample>thaksh</Sample>
        <Sample>shyama</Sample>
        </div>
,document.getElementById("root"))
```

**Answer: Only change is as below**

```
function Sample(props)
{       return <Sample1>{props.children}</Sample1>      }
```

**Output:**

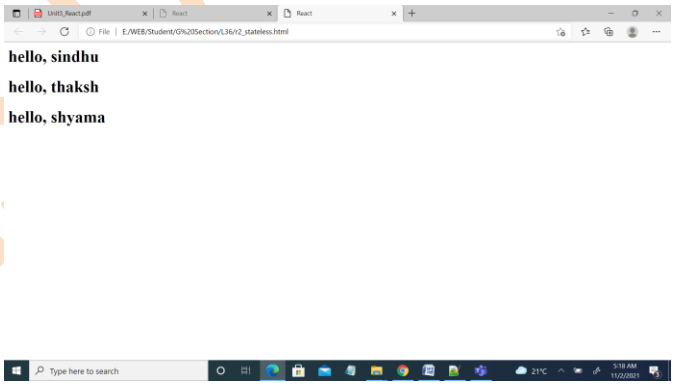



**Coding Example 3: Usage of variable within the component and no change in rendering the component**

```
function Sample(props)
    {       var greeting = "hello,"
            return <Sample1>{greeting} {props.children}</Sample1>
    }
```

\

**Output:**



## Stateful vs Stateless Components

| Stateful Components | Stateless Components |
|---|---|
| Also known as container or smart components. | Also known as presentational or dumb components. |
| Have a state | Do not have a state |
| Can render both props and state | Can render only props |
| Props and state are rendered like *{this.props.name}* and *{this.state.name}* respectively. | Props are displayed like *{props.name}* |
| A stateful component is always a *class* component. | A Stateless component can be either a functional or class component. |

# Refs and Keys in React

## Introduction to Refs

Ref provides a way **to access DOM nodes or React elements** created in the render method. It is an **attribute which makes it possible to store a reference to particular DOM nodes or React elements.**

According to React.js documentation some of the best cases for using refs are:

- ➢ managing focus
- ➢ text selection
- ➢ media playback
- ➢ triggering animations
- ➢ integrating with third-party DOM libraries

Usually props are the way for parent components to interact with their children. However, in some cases you might need **to modify a child without re-rendering it with new props**. That's exactly when refs attribute comes to use.

## Need of Refs

Consider the below code to fulfil the requirement of clicking on the plus button must increment the value of input text box and clicking on the minus button must decrement the value of text box.

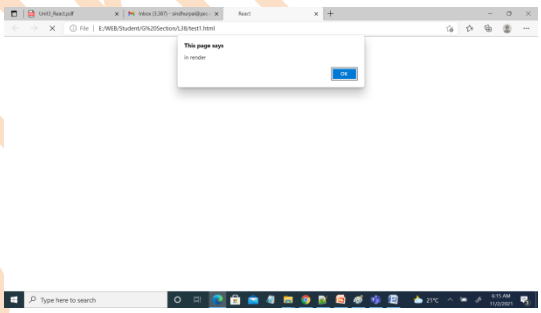
**Coding Example 1:**

```
<div id="root"></div>
        <script type = "text/babel">
        class My_component extends React.Component
        {       constructor()
                {       super();        this.state = {val:0}            }
                render()
                {       alert("in render");
                        return (<div>
```

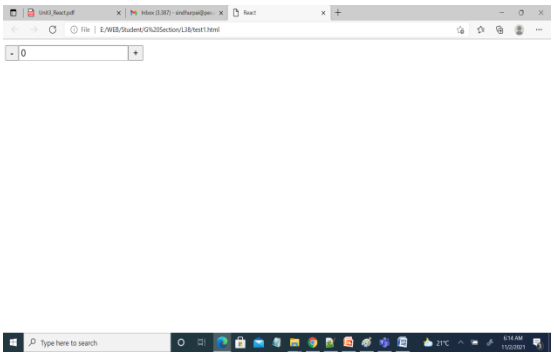
```
                    <button onClick = {this.decrement}>-</button>
                    <input type = "text" value = {this.state.val}/>
                    <button onClick = {this.increment}>+</button>
                    </div>)
                }
                increment=()=>
                {       this.setState({val: this.state.val+1}  )   }
                decrement=()=>
                {       this.setState({val: this.state.val-1})    }
            }
            ReactDOM.render(<My_component/>,
document.getElementById("root"))
        </script>
```

**Outputs:**

**Case 1: Rendered the page on the browser**



**Case 2:OK is clicked**

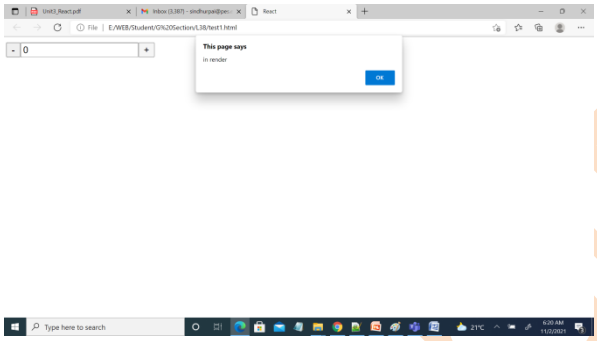


There are two problems in above code.

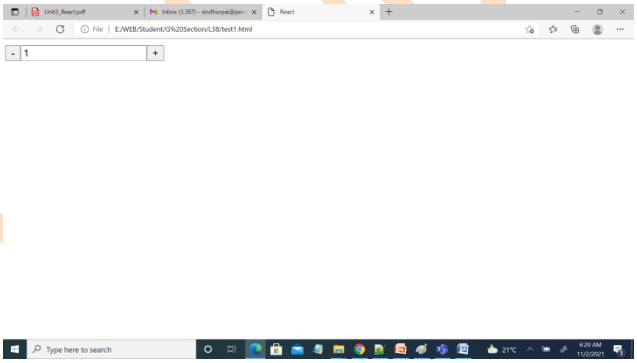
1. When + buton or – button is clicked, the render function is getting called by default as shown in the output below.

2. Unable to edit the input text box

**Case 3: + is clicked, observe the mouse pointer**

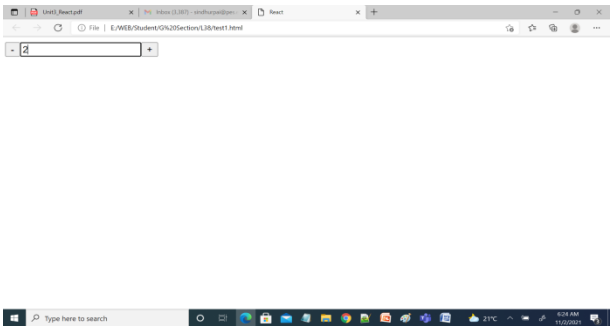



**Case 4: Ok is clicked**



Note: Same with - button

**Case 5:If any key is pressed, input text box no effect**



To avoid above problems, we use refs.

## Creation and Accessing references

Refs are created using **React.createRef().** Can be assigned to React elements via the **ref attribute**. Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.

When a ref is passed to an element in render, **a reference to the node** becomes accessible at the **current attribute of the ref.** The value of the ref differs depending on the type of the node:

When used on a HTML element, the ref created in the constructor with React.createRef() receives the underlying DOM element as its current property.

When used on a custom class component, the ref object receives the mounted instance of the component as its current.

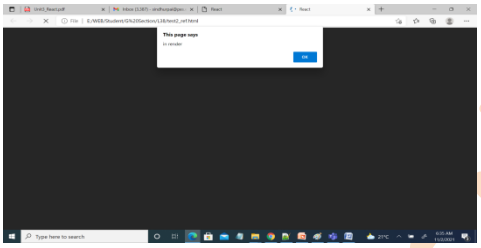
**Coding example 2:**

```
<div id="root"></div>
    <script type = "text/babel">
        class My_component extends React.Component
        {   constructor()
            {   super(); this.myref = React.createRef()          }
            render()
            {   alert("in render")
                return (
                    <input type = "text" ref = {this.myref} />
                    <button onClick = {this.increment}>+</button>
                    </div>
                )   }
            increment=()=>
            {   this.myref.current.value++;                  }
            decrement=()=>
            {   this.myref.current.value--;                  }
        }
```

```
ReactDOM.render(<My_component/>, document.getElementById("root"))
</script>
```

**Outputs:**

**Case 1: When the page is loaded**



**Case 2: When OK is clicked**



**Case 3: + button is clicked, render not called**

**Case 4: Again + is clicked, render not called**

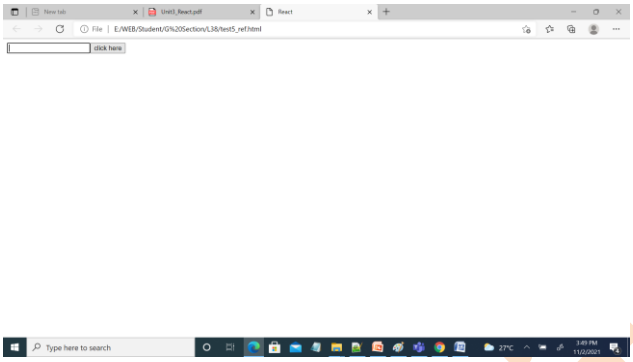
**Coding Example 3: Autofocus the input field by adding ref to a DOM element**

```
<body>
<div id="root"></div>
<script type = "text/babel">
        class CustomInput extends React.Component
        {       constructor()
                {       super();        this.textref = React.createRef()
                }
                focusTextInput=()=>
                {           this.textref.current.focus()        }
                render()
                {       return (<div>
                                <input type = "text" ref = {this.textref} />
                                <input type = "button" value = "submit" onClick =
                                {this.focusTextInput}/>
                                </div> )
                }
        }
        ReactDOM.render(<CustomInput />,document.getElementById("root"))
</script>
</body>
```

**Output:**

**Case 1: When the page is loaded**

## Case 2: After clicking on click here button



**Think about having no button. When the page is loaded, input box must automatically get focused.**

**Coding Example 4:**

```
<body>
<div id="root"></div>
<script type = "text/babel">
        class CustomInput extends React.Component
        {       constructor()
                {       super(); this.textref = React.createRef()       }
                componentDidMount()
                 //Removed focusInoutText function. Added the functionality in one of the life cycle method
                {       this.textref.current.focus()                    }
                render()
                {       return (
                                <div>
                                <input type = "text" ref = {this.textref} />
                                </div> )
                }
        }
        ReactDOM.render(<CustomInput />,document.getElementById("root"))
</script>
</body>
```

**Few points to think:**

- Can we have the ref set for component rather than the DOM Element?
- Can you create more than references for the same element?

## Callback refs

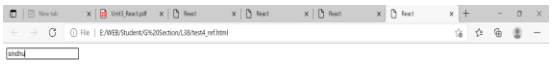
React also supports another way to set refs called "callback refs", which gives more fine-grain control over when refs are set and unset. Instead of passing a ref attribute created by createRef(), you pass a function. **The function receives the React component instance or HTML DOM element as its argument,** which can be stored and accessed elsewhere.

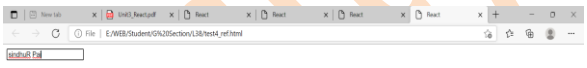
**Coding Example 5: Consider the input box. As and when the user types into it, the content of input box is displayed on the page. If the user presses shift key, content has to be displayed in red color**

```
<body>
    <div id="root"></div>
    <script type = "text/babel">
        var txt;
        class My_component extends React.Component
        {    constructor()
             {  super(); this.myref = (ele) => {this.setref = ele}// callback ref    }
             render()
             { return(<div>
                    <input type = "text" onKeyPress = {this.show} />
                    <h1 ref = {this.myref}></h1>
                    </div>
             ) }
             show=(e)=>
             {   txt = e.key
                 if(e.shiftKey){
                  this.setref.innerHTML += '<span style =
```

```
                    "color:red"}>'+txt+'</span>'   }
                    else {   this.setref.innerHTML += txt   } // current not available
                }
            }
            ReactDOM.render(<My_component/>, document.getElementById("root"))
        </script>
</body>
```

**Output:**




## Introduction to Keys

A key is a unique identifier which helps to identify which items have changed, added, or removed. Useful when we dynamically created components or when users alter the lists. The best way to pick a key is to choose a string that uniquely identifies the items in the list. Keys used within arrays should be unique among their siblings. However, they **don't need to be globally unique.** Also helps in efficiently updating the DOM.

**Coding example 6:** Consider an array containing n elements in it. Display these elements using n bullet items in an unordered list

```
<script type = "text/babel">
        const arr = ["book1","book2","book3", ","book4"]
```

```
function Booklist(props)
        {       return (<ul> <li >{props.books[0]}</li>
                             <li >{props.books[1]}</li>
                             <li >{props.books[2]}</li>
                             <li >{props.books[3]}</li>
                        </ul>)
        }
    ReactDOM.render(<Booklist books = {arr}/>,document.getElementById("root"))
</script>
```

**Observation:** As and when the number of elements changes in the array, the code runs into trouble. Refer to the below code to have this dynamism.

**Coding example 7:**

```
<script type = "text/babel">
    function Booklist(props)
    {
        const book_lists= props.books
        //console.log(book_lists)
        const b = book_lists.map((book,index) => <li key = {index} >{book}</li>)
        return <ul>{b}</ul>
    }
    ReactDOM.render(<Booklist books = {arr}/>,document.getElementById("root"))
</script>
```

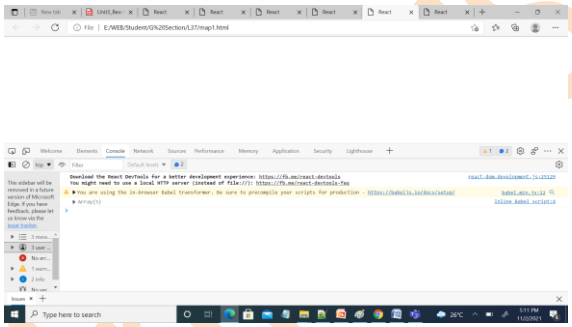
## Usage of map

React really simplifies the rendering of lists inside the JSX by supporting the Javascript .map() method. The .map() method in Javascript **iterates through the parent array and calls a function on every element of that array**. **Then it creates a new array with transformed values. It doesn't change the parent array.**

**Coding example 8:**

```
<body>
        <div id="root"></div>
        <script type = "text/babel">
                const numbers = [1, 2, 3, 4, 5];
                const doubled = numbers.map((number) => number * 2);
                console.log(doubled); //[2, 4, 6, 8, 10]
        </script>
 </body>
```

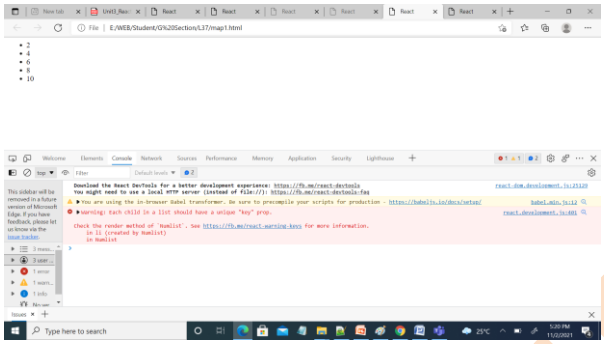
**Output:**



**Coding example 9: Rendering the doubled numbers on the browser**

```
<body>
        <div id="root"></div>
        <script type = "text/babel">
        function Numlist()
        {
                const numbers = [1, 2, 3, 4, 5];
                const doubled = numbers.map((number) => <li>{number * 2}</li>);
                return <ul>{doubled}</ul>
        }
        ReactDOM.render(<Numlist />,document.getElementById("root"))
        </script>
 </body>
```

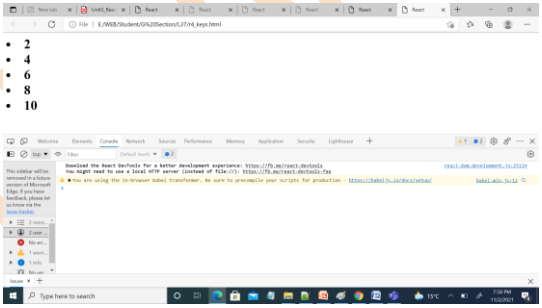
**Output:**



**Note: Please observe the warning. To avoid warning, refer to the below code**

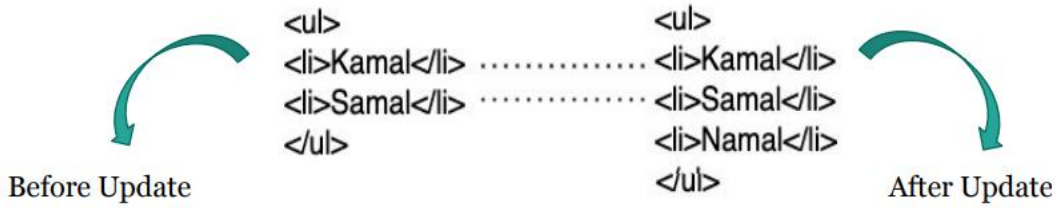
**Coding example 10:**

The below line of statement, **const doubled = numbers.map((number) => <li>{number * 2}</li>);** must be changed to **const doubled = props.numbers.map((number) => <li key = {number.toString()}>{number * 2}</li>);**




## Importance of keys in react

Consider the scenario of updating a list.




In the above case, React will verify that the first and second elements have not been

changed and adds only the last element to the list.

Now, consider a change in the updated list.


In this case, React cannot identify that "Kamal" and "Namal" have not been changed. Therefore, it will update three elements instead of one which will lead to a waste of performance. To solve this, keys are used.


Try writing the code to update the given list using react keys

# Event Handling in React

## Introduction

Events make the **web app interactive and responsive** to the user. Most of the time, you don't have static UIs; you need to build elements that are smart enough to respond to user actions. So far in the examples we've seen, we brought in a certain level of user interaction such as performing some action on a button click. By now, you would have realized that handling events with React elements is very similar to handling events on DOM elements.

## React Event Handling vs DOM Event handling

**1. With JSX in ReactJs, you pass a function as event handler and in DOM element we pass function as string**

```
// event handling in ReactJs element
<input id="inp" name="name" onChange={onChangeName} />
```

```
// event handling in DOM element
<input id="inp" name="name" onchange="onChangeName()"/>
```

**2. In DOM elements the event name is in lowercase while in ReactJs it is in camalCase. List of events are as follows.**

**Mouse**

- onClick
- onContextMenu
- onDoubleClick
- onMouseDown
- onMouseEnter
- onMouseLeave
- onMouseMove
- onMouseOut
- onMouseOver
- onMouseUp

**Image**

- onLoad
- onError

**Selection**

- onSelect

**Form**

- onChange
- onInput
- onSubmit

**Focus**

- onFocus
- onBlur

**Keyboard**

- onKeyDown
- onKeyPress
- onKeyUp

**UI**

- onScroll

**3. Cannot return false to prevent default behavior in React. Must call preventDefault explicitly.**

*HTML*

```
<form onsubmit="console.log('You clicked submit.'); return false">
  <button type="submit">Submit</button>
</form>
```

*React*

```
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('You clicked submit.');
      }
  return (<form onSubmit={handleSubmit}>
   <button type="submit">Submit</button>
   </form>);
        }
```

## Event Registration

It tells the browser that a particular function should be called whenever a definite event occurs i.e whenever an event is triggered, the function which is bound to that event should be called. Essentially, it allows you to add an event handler for a specified event. This can be used to bind to any event, such as keypress , mouseover or mouseout . Since class methods are not bound by default, it's necessary to bind functions to the class instance so that the this keyword would not return "undefined".

## Synthetic Event Objects

React event handling system is known as Synthetic Events. The event object passed to the event handlers are SynthticEvent Objects. It is a wrapper around the DOMEvent object. The event handlers are registered at the time of rendering. Whenever you call an event handler within ReactJS, they are passed an instance of SyntheticEvent. A SyntheticEvent event has all of its usual **properties and methods**. These include its **type, target, mouse coordinates,** and so on. React defines these synthetic events according to the W3C spec to take care of **cross-browser compatibility.**

SyntheticEvent that wraps a **MouseEvent will have access to mouse-specific properties** such as the following:

```
boolean altKey                          boolean metaKey
number button                           number pageX
number buttons                          number pageY
number clientX                          DOMEventTarget relatedTarget
number clientY                          number screenX
boolean ctrlKey                         number screenY
boolean getModifierState(key)           boolean shiftKey
```

A SyntheticEvent that wraps a **KeyboardEvent will have access to keyboard-related properties** such as the following:

```
boolean altKey                          string locale
number charCode                         number location
boolean ctrlKey                         boolean metaKey
boolean getModifierState(key)           boolean repeat
string key                              boolean shiftKey
number keyCode                          number which
```

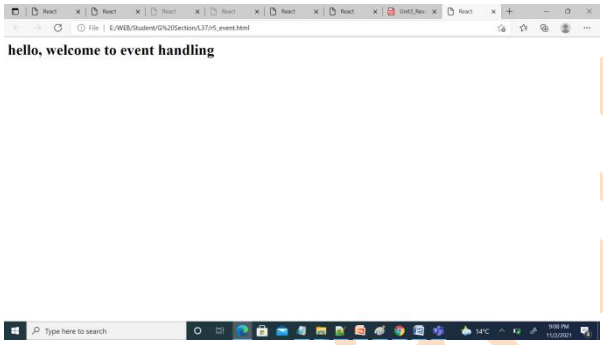
**Coding example 1: Simple code to demo event handling**

```
<body>
        <div id="root"></div>
        <script type = "text/babel">
                class NewOne extends React.Component    {
                        constructor()
                        {       super();
                                this.state = {content:"hello, welcome to event handling"}
                        }
                        render()
                        {return <h1 onClick = {this.fun1}>{this.state.content}</h1>  }
                        fun1=()=>
                        {       this.setState({content:"new text"})             }
```

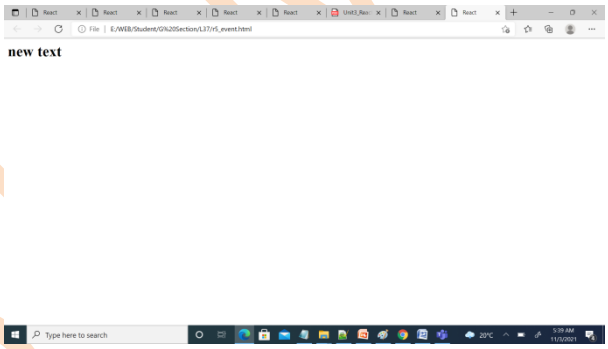
```
        }
        ReactDOM.render(<NewOne/>,document.getElementById("root"))
    </script>
<body>
```

**Output:**

**Case 1: When page is loaded**




**Case 2: When text is clicked**



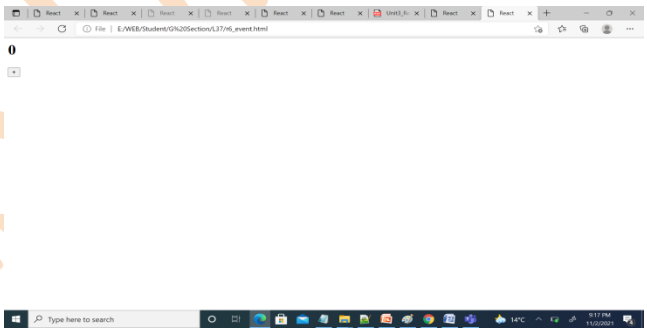

- **Coding example 2: Requirement is to click on the + button, the value of counter must be incremented by one**

```
        <body>
            <div id="root"></div>
            <script type = "text/babel">
                class NewOne extends React.Component    {
                constructor()
                {  super();              this.state = {counter:0}        }
```

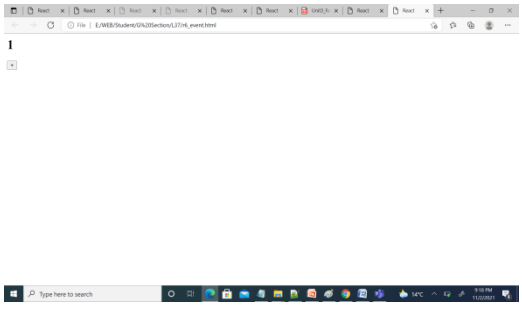
```
            render()        {
                return (<div>
                            <h1 >{this.state.counter}</h1>
                            <button onClick = {this.fun1}>+</button>
                            </div>)

            }
            fun1=()=>
            {       //this.setState(counter:this.state.counter+1})
                    this.setState((prevState) => ({counter:prevState.counter+1}))
            }
        }
        ReactDOM.render(<NewOne/>,document.getElementById("root"))
    </script>
<body>
```

**Output:**

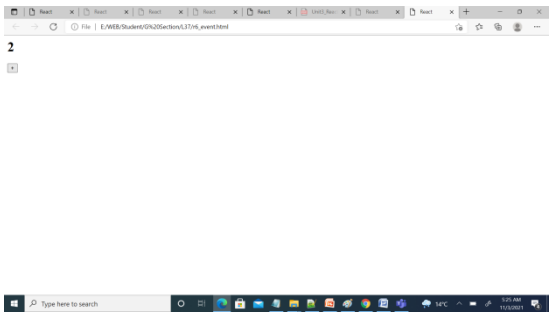
**Case 1: when the page is loaded**




**Case 2: When + button is clicked once**

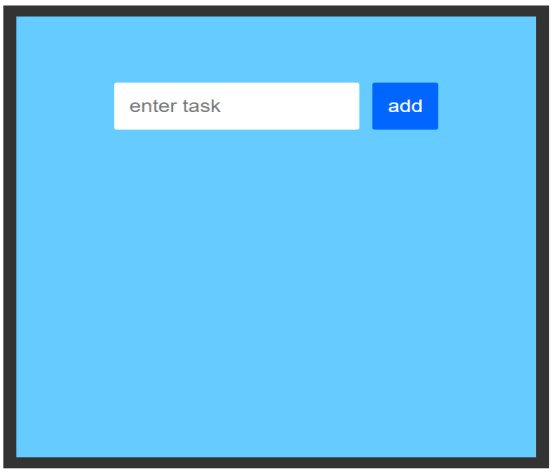


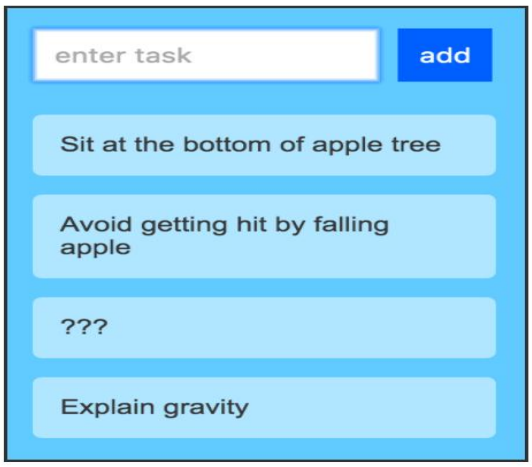
**Case 3: When + button is clicked again**




**Practice Programs**

**1. Build an awesome todo list as shown below. Type the task in the input box provided and click on add button. This must create an element below the input box with a new background color for this. Clicking on the task directly, must delete the element from the list.**




**2. Simulate the below to obtain the current date string on the click of a button.**

**Current Time:**

Sat Oct 16 2021 17:10:00 GMT+0530 (India Standard Time)

Get Current Time!

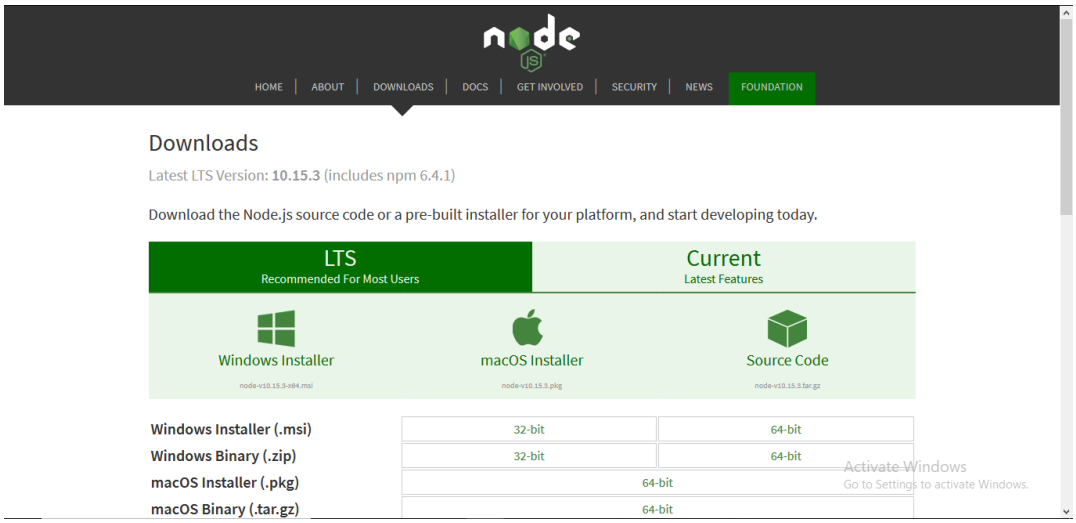
## Installation of Node.js on Windows

### Installing Node on Windows (WINDOWS 10):

You have to follow the following steps to install the Node.js on your Windows:

**Step-1:** Downloading the Node.js '.msi' installer.

The first step to install Node.js on windows is to download the installer. Visit the official Node.js website i.e) https://nodejs.org/en/download/ and download the .msi file according to your system environment (32-bit & 64-bit). An MSI installer will be downloaded on your system.




## NOTE :

A prompt saying – "This step requires administrative privileges" will appear.

Authenticate the prompt as an "Administrator"

### Step-2: Installing Node.js.

Do not close or cancel the installer until the install is complete

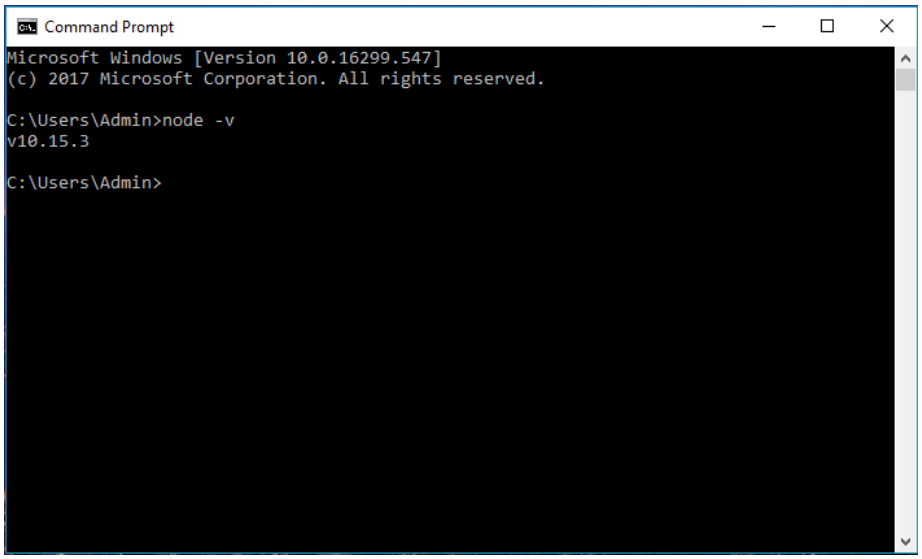
Complete the Node.js Setup Wizard.

Click “Finish”

**Step 3: Verify that Node.js was properly installed or not.**

To check that node.js was completely installed on your system or not, you can run the following command in your command prompt or Windows Powershell and test it:-

**C:\Users\Admin> node -v**




If node.js was completely installed on your system, the command prompt will print the version of the node.js installed.

**Step 4: Updating the Local npm version.**

The final step in node.js installed is the updation of your local npm version(if required) – the package manager that comes bundled with Node.js.

You can run the following command, to quickly update the npm

**npm install npm –global // Updates the ‘CLI’ client**

## Node.js Basics

Node.js is a cross-platform JavaScript runtime environment.

It allows the creation of scalable Web servers without threading and networking tools using JavaScript and a collection of “modules” that handle various core functionalities.

It can make console-based and web-based node.js applications.

**Datatypes:** Node.js contains various types of data types similar to JavaScript.

- Boolean
- Undefined
- Null
- String
- Number

Loose Typing: Node.js supports loose typing, it means you don’t need to specify what type of information will be stored in a variable in advance. We use var keyword in Node.js to declare any type of variable. Examples are given below:

Example:

```
// Variable store number data type

var a = 35;

console.log(typeof a);


// Variable store string data type

a = "PES";

console.log(typeof a);


// Variable store Boolean data type

a = true;

console.log(typeof a);
```

```
// Variable store undefined (no value) data type
a = undefined;
console.log(typeof a);
```

## Objects & Functions

Node.js objects are same as JavaScript objects i.e. the objects are similar to variable and it contains many values which are written as name: value pairs. Name and value are separated by colon and every pair is separated by comma.

Example:

```
var company = {
   Name: "PES University",
   Address: "India",
   Contact: "+919876543210",
   Email: "www.pes.edu"
};

// Display the object information
console.log("Information of variable company:", company);

// Display the type of variable
console.log("Type of variable company:", typeof company);
```

**Functions**:

Node.js functions are defined using function keyword then the name of the function and parameters which are passed in the function.

In Node.js, we don’t have to specify datatypes for the parameters and check the number of arguments received.

Node.js functions follow every rule which is there while writing JavaScript functions.

```
function multiply(num1, num2) {

   // It returns the multiplication
   // of num1 and num2
   return num1 * num2;
}

// Declare variable
var x = 2;
var y = 3;

// Display the answer returned by
// multiply function
console.log("Multiplication of", x, "and", y, "is", multiply(x, y));
```

If you observe in the above example, we have created a function called “multiply” with parameters same like JavaScript.

**String and String Functions**: In Node.js we can make a variable as string by assigning a value either by using single (’’) or double (“”) quotes and it contains many functions to manipulate to strings.

Following is the example of defining string variables and functions in node.js.

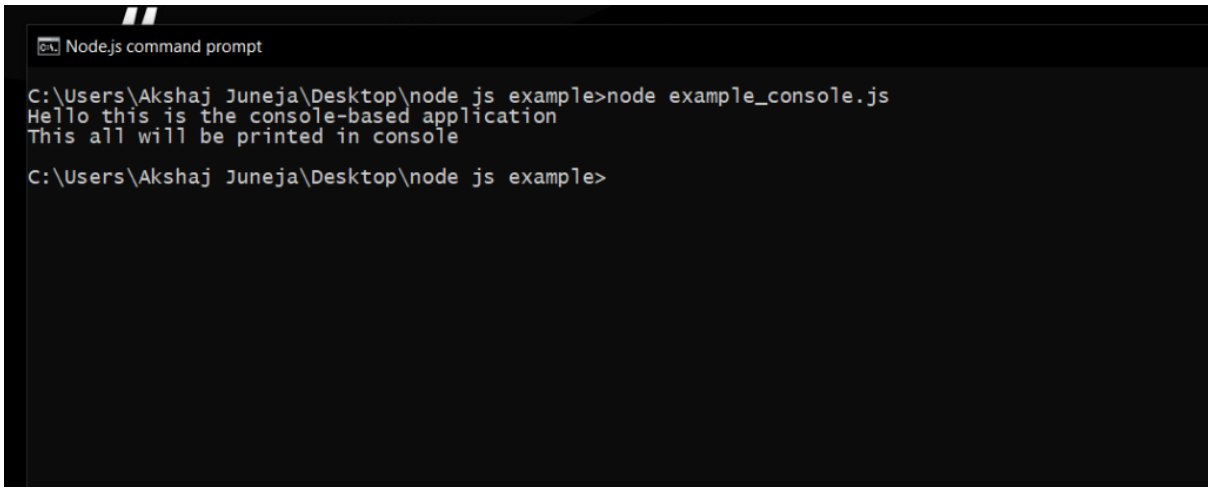
```
var x = "Welcome ";
var y = 'Node.js Tutorials';
var z = ['Node', 'server', 'side'];
console.log(x);
console.log(y);
```

```
console.log("Concat Using (+) :", (x + y));

console.log("Concat Using Function :", (x.concat(y)));

console.log("Split string: ", x.split(' '));

console.log("Join string: ", z.join(', '));

console.log("Char At Index 5: ", x.charAt(5) );
```

**Node.js console-based application:** Make a file called console.js with the following code.

```
console.log('Hello this is the console-based application');

console.log('This all will be printed in console');

// The above two lines will be printed in the console.
```

To run this file, **open node.js command prompt** and go to the folder where console.js file exist and write the following command. It will display content on console.



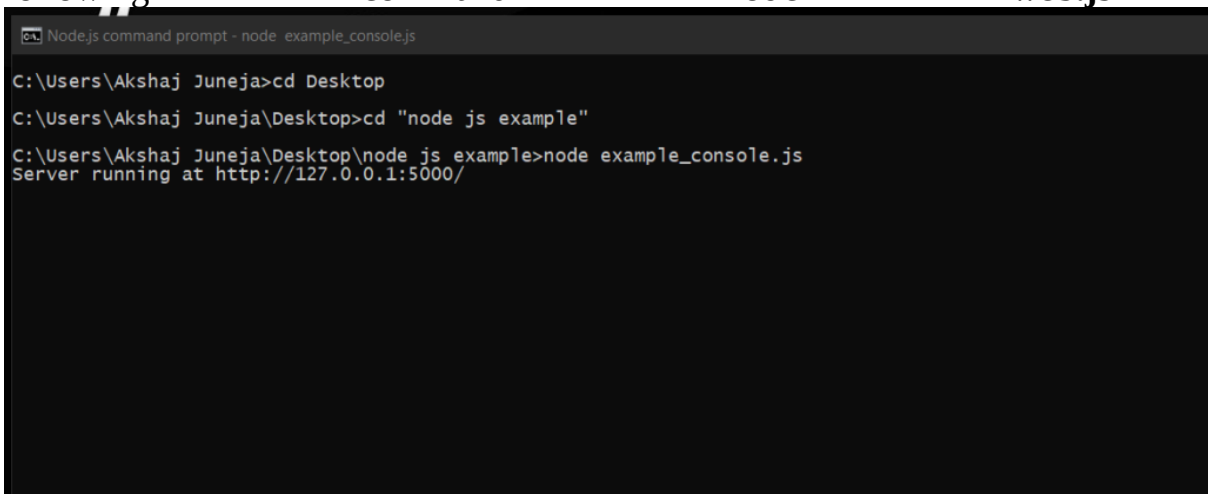

**Node.js web-based application:** Node.js web application contains different types of modules which is imported using **require()** directive and we have to create a server and write code for the read request and return response. Make a file web.js with the following code.

```
// Require http module
var http = require("http");
// Create server
http.createServer(function (req, res) {
   // Send the HTTP header
   // HTTP Status: 200 : OK
   // Content Type: text/plain
   res.writeHead(200, {'Content-Type': 'text/plain'});

   // Send the response body as "This is the example
   // of node.js web based application"
  res.end('This is the example of node.js web-based application \n');

// Console will display the message
}).listen(5000,
   ()=>console.log('Server running at http://127.0.0.1:5000/'));
```

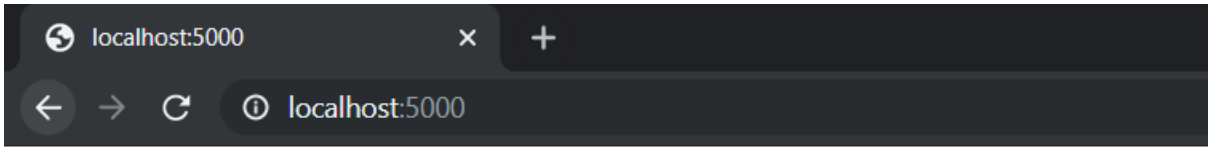
**To run this file follow the steps as given below:**

- Search the node.js command prompt in the search bar and open the node.js command prompt.

- Go to the folder using **cd** command in command prompt and write the following command **node web.js**




Now the server has started and go to the browser and open this url localhost:5000




This is the example of node.js web-based application

# NODE Modules

**NPM Package**

NPM is a package manager for Node.js packages, or modules if you like.

www.npmjs.com hosts thousands of free packages to download and use.

The NPM program is installed on your computer when you install Node.js

**A package in Node.js contains all the files you need for a module.**

**Modules are JavaScript libraries you can include in your project.**

Example:

```
D:\nodejs>npm install validator
```

validator package is downloaded and installed. NPM creates a folder named "node_modules", where the package will be placed.

To include a module, use the require() function with the name of the module


```
D:\nodejs>npm install validator
```


**What is a Module in Node.js?**

- Modules are the blocks of encapsulated code that communicates with an external application on the basis of their related functionality.
- Modules can be a single file or a collection of multiples files/folders.
- The reason programmers are heavily reliant on modules is because of their re-usability as well as the ability to break down a complex piece of code into manageable chunks

**There are 2 types of Modules:**

- **Built in Modules**
- **Local Modules**

To see all the available modules

- https://nodejs.org/dist/latest-v12.x/docs/api/

**Modules can be imported to code using**

**require () function**

- Few modules are inbuilt globally available **no need of require function.**

**Ex: Console module, Timer Module**

- Many modules need to be **explicitly included** in our application

**Ex: File System module**

Such modules need to be required at first in the application

Node.js has many built-in modules that are part of the platform and comes with Node.js installation. **These modules can be loaded into the program by using the require function.**

Syntax:

**var module = require('module_name');**

The require() function will return a JavaScript type depending on what the particular module returns.

**Importing your own modules**

- The module.exports is a special object which is included in every JavaScript file in the Node.js application by default.
- The module is a variable that represents the current module, and exports is an object that will be exposed as a module.
- So, whatever you assign to module. exports will be exposed as a module. It can be
    - Export Literals
    - Export Objects
    - Export Functions
    - Export Function as a class

## Node JS Local-Modules

local modules are created locally in your Node.js application. Let's create a simple calculating module that calculates various operations. Create a calc.js file that has the following code:

Filename: calc.js

```
exports.add = function (x, y) {
   return x + y;
};

exports.sub = function (x, y) {
   return x - y;
};

exports.mult = function (x, y) {
   return x * y;
};
```

Since this file provides attributes to the outer world via exports, another file can use its exported functionality using the require() function.

Filename: index.js

```
var calculator = require('./calc');

var x = 50, y = 20;

console.log("Addition of 50 and 10 is "
            + calculator.add(x, y));

console.log("Subtraction of 50 and 10 is "
            + calculator.sub(x, y));
```

**console.log("Multiplication of 50 and 10 is "**

**+ calculator.mult(x, y));**

**Step to run this program: Run index.js file using the following command:**

**node index.js**

**Create Modules in Node.js**

- To create a module in Node.js, use **exports** keyword tells Node.js that the function can be used outside the module.
- **Create a file that you want to export**
-

JS calc.js > ...

```
1   exports.add = function (a, b) {
2       return a + b;
3   };
4
5   exports.sub = function (a, b) {
6       return a - b;
7   };
8
9   exports.mult = function (a, b) {
10      return a * b;
11  };
12    exports.div = function (a, b) {
13      return a / b;
14  };
```

JS U4L2.js > ...

```
14
15    var a = 50, b = 20;
16
17    console.log("Addition of 50 and 20 is "
18                + calculator.add(a, b));
19
20    console.log("Subtraction of 50 and 20 is "
21                + calculator.sub(a, b));
22
23    console.log("Multiplication of 50 and 20 is "
24                + calculator.mult(a, b));
25
26    console.log("Division of 50 and 20 is "
27                + calculator.div(a, b));
```

## Node JS Built-in Modules

- Node.js has many built-in modules that are part of the platform and comes with Node.js installation.
- These modules can be loaded into the program by using the require function.
- Syntax:
- **var module = require('module_name');**
- The require() function will return a JavaScript type depending on what the particular module returns.

| Core Modules | Description |
| --- | --- |
| **http** | **creates an HTTP server in Node.js.** |
| **assert** | **set of assertion functions useful for testing.** |
| **fs** | **used to handle file system.** |
| **path** | **includes methods to deal with file paths.** |
| **process** | **provides information and control about the current Node.js process.** |
| **os** | **provides information about the operating system.** |
| **querystring** | **utility used for parsing and formatting URL query strings.** |
| **url** | **module provides utilities for URL resolution and parsing.** |

Some Modules which are Globally Available where there is no need of Require () function

Ex: Console

Timer Module

**Node JS Timer Module**

- This module provides a way for functions to be called later at a given time.
- The Timer object is a global object in Node.js, and it is not necessary to import it

| **Method** | **Description** |
|---|---|
| clearImmediate() | Cancels an Immediate object |
| clearInterval() | Cancels an Interval object |
| clearTimeout() | Cancels a Timeout object |
| ref() | Makes the Timeout object active. Will only have an effect if the Timeout.unref() method has been called to make the Timeout object inactive. |
| setImmediate() | Executes a given function immediately. |
| setInterval() | Executes a given function at every given milliseconds |
| setTimeout() | Executes a given function after a given time (in milliseconds) |
| unref() | Stops the Timeout object from remaining active |

Ex:

```
function printHello() {
   console.log( "Hello, World!");
}
// Now call above function after 2 seconds
var timeoutObj = setTimeout(printHello, 2000);
```

Some More examples of Local Modules:

**1.Exporting Date Module**

Date.js

```
exports.myDateTime=function(){
return Date()
};
```

Main Program where we are importing Date

```
var D=require('./Date.js')
console.log(D.myDateTime());
```

2.

**LOG.JS**

```
var log = {
        info: function (info) {
           console.log('Info: ' + info);
        },
        warning:function (warning) {
           console.log('Warning: ' + warning);
        },
        error:function (error) {
           console.log('Error: ' + error);
        }
   };

module.exports = log
```

In the above example of logging module, we have created an object with three functions - info(), warning() and error(). At the end, we have assigned this object to module.exports. The module.exports in the above example exposes a log object as a module.

The module.exports is a special object which is included in every JS file in the Node.js application by default. Use module.exports or exports to expose a function, object or variable as a module in Node.js.

## Loading Local Module

To use local modules in your application, you need to load it using require() function in the same way as core module. However, you need to specify the path of JavaScript file of the module.

The following example demonstrates how to use the above logging module contained in Log.js.

```
var myLogModule = require('./Log.js');

myLogModule.info('Node.js started');
```

In the above example, app.js is using log module. First, it loads the logging module using require() function and specified path where logging module is stored. Logging module is contained in Log.js file in the root folder. So, we have specified the path './Log.js' in the require() function. The '.' denotes a root folder.

**The require() function** returns a log object because logging module exposes an object in Log.js using **module.exports**. So now you can use logging module as an object and call any of its function using dot notation e.g myLogModule.info() or myLogModule.warning() or myLogModule.error()

3.

## Local.js

```
const welcome = {
   sayHello : function() {
       console.log("Hello  user");
  },
  currTime : new Date().toLocaleDateString(),
  companyName : "PESU"
}
```

## Main.js

```
var local = require("./Welcome.js");
local.sayHello();
```

```
console.log(local.currTime);

console.log(local.companyName);

module.exports = welcome
```

**Export Module in Node.js**

The module.exports is a special object which is included in every JavaScript file in the Node.js application by default. The module is a variable that represents the current module, and exports is an object that will be exposed as a module. So, whatever you assign to module.exports will be exposed as a module.

**FILE Module**

**- https://nodejs.org/api/fs.html, https://nodejs.org/api/querystring.html**

The fs module provides a lot of very useful functionality to access and interact with the file system.

There is no need to install it. Being part of the Node.js core, it can be used by simply requiring it:

const fs = require('fs')

Common use for the File System module:

Read files

Create files

Update files

Delete files

Rename files

**Synchronous vs Asynchronous**

- Every method in the fs module has synchronous as well as asynchronous forms.
- Asynchronous methods take the last parameter as the completion function callback and the first parameter of the callback function as error.
- It is better to use an asynchronous method instead of a synchronous method, as the former never blocks a program during its execution, whereas the second one does.

What is Synchronous and Asynchronous approach?

Synchronous approach: They are called blocking functions as it waits for each operation to complete, only after that, it executes the next operation, hence blocking the next command from execution i.e. a command will not be executed until & unless the query has finished executing to get all the result from previous commands.

Asynchronous approach:

They are called non-blocking functions as it never waits for each operation to complete, rather it executes all operations in the first go itself.

The result of each operation will be handled once the result is available i.e. each command will be executed soon after the execution of the previous command.

While the previous command runs in the background and loads the result once it is finished processing the data.

Example:

Synchronous:

```
var fs = require("fs");

// Synchronous read
var data = fs.readFileSync('input.txt');
console.log("Synchronous read: " + data.toString());
```

Asynchronous:

```
var fs = require("fs");

// Asynchronous read
fs.readFile('input.txt', function (err, data) {
   if (err) {
      return console.error(err);
   }
   console.log("Asynchronous read: " + data.toString());
});
```

***Once you do so, you have access to all its methods, which include:***

- fs.access(): check if the file exists and Node.js can access it with its permissions
- fs.appendFile(): append data to a file. If the file does not exist, it's created
- fs.chmod(): change the permissions of a file specified by the filename passed. Related: fs.lchmod(), fs.fchmod()

- fs.chown(): change the owner and group of a file specified by the filename passed. Related: fs.fchown(), fs.lchown()
- fs.close(): close a file descriptor
- fs.copyFile(): copies a file
- fs.createReadStream(): create a readable file stream
- fs.createWriteStream(): create a writable file stream
- fs.link(): create a new hard link to a file
- fs.mkdir(): create a new folder
- fs.mkdtemp(): create a temporary directory
- fs.open(): set the file mode
- fs.readdir(): read the contents of a directory
- fs.readFile(): read the content of a file. Related: fs.read()
- fs.readlink(): read the value of a symbolic link
- fs.realpath(): resolve relative file path pointers (., ..) to the full path
- fs.rename(): rename a file or folder
- fs.rmdir(): remove a folder
- fs.stat(): returns the status of the file identified by the filename passed. Related: fs.fstat(), fs.lstat()
- fs.symlink(): create a new symbolic link to a file
- fs.truncate(): truncate to the specified length the file identified by the filename passed. Related: fs.ftruncate()
- fs.unlink(): remove a file or a symbolic link
- fs.unwatchFile(): stop watching for changes on a file
- fs.utimes(): change the timestamp of the file identified by the filename passed. Related: fs.futimes()
- fs.watchFile(): start watching for changes on a file. Related: fs.watch()
- fs.writeFile(): write data to a file. Related: fs.write()

One peculiar thing about the fs module is that all the methods are asynchronous by default, but they can also work synchronously by appending Sync.

For example:

- fs.rename()
- fs.renameSync()
- fs.write()
- fs.writeSync()

This makes a huge difference in your application flow.

## Open a File

### Syntax

Following is the syntax of the method to open a file in asynchronous mode −

```
fs.open(path, flags[, mode], callback)
```

### Parameters

Here is the description of the parameters used −

- **path** − This is the string having file name including path.
- **flags** − Flags indicate the behavior of the file to be opened. All possible values have been mentioned below.
- **mode** − It sets the file mode (permission and sticky bits), but only if the file was created. It defaults to 0666, readable and writeable.
- **callback** − This is the callback function which gets two arguments (err, fd).

### Flags

Flags for read/write operations are −

| Sr.No. | Flag & Description |
|---|---|
| 1 | **r**<br>Open file for reading. An exception occurs if the file does not exist. |
| 2 | **r+**<br>Open file for reading and writing. An exception occurs if the file does not exist. |
| 3 | **rs**<br>Open file for reading in synchronous mode. |
| 4 | **rs+**<br>Open file for reading and writing, asking the OS to open it synchronously. See notes for 'rs' about using this with caution. |
| 5 | **w**<br>Open file for writing. The file is created (if it does not exist) or truncated |

| | |
|---|---|
| | (if it exists). |
| 6 | **wx**<br>Like 'w' but fails if the path exists. |
| 7 | **w+**<br>Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists). |
| 8 | **wx+**<br>Like 'w+' but fails if path exists. |
| 9 | **a**<br>Open file for appending. The file is created if it does not exist. |
| 10 | **ax**<br>Like 'a' but fails if the path exists. |
| 11 | **a+**<br>Open file for reading and appending. The file is created if it does not exist. |
| 12 | **ax+**<br>Like 'a+' but fails if the the path exists. |

## Writing a File

### Syntax

Following is the syntax of one of the methods to write into a file −

```
fs.writeFile(filename, data[, options], callback)
```

This method will over-write the file if the file already exists. If you want to write into an existing file then you should use another method available.

### Parameters

Here is the description of the parameters used −

- **path** − This is the string having the file name including path.

- **data** − This is the String or Buffer to be written into the file.
- **options** − The third parameter is an object which will hold {encoding, mode, flag}. By default. encoding is utf8, mode is octal value 0666. and flag is 'w'
- **callback** − This is the callback function which gets a single parameter err that returns an error in case of any writing error.

## Reading a File

- Node implements File I/O using simple wrappers around standard POSIX functions.
- The Node File System (fs) module can be imported using the following syntax −

## Synchronous vs Asynchronous

- Every method in the fs module has synchronous as well as asynchronous forms.
- Asynchronous methods take the last parameter as the completion function callback and the first parameter of the callback function as error.
- It is better to use an asynchronous method instead of a synchronous method, as the former never blocks a program during its execution, whereas the second one does.

## Syntax

Following is the syntax of one of the methods to read from a file −

```
fs.read(fd, buffer, offset, length, position, callback)
```

This method will use file descriptor to read the file. If you want to read the file directly using the file name, then you should use another method available.

## Parameters

Here is the description of the parameters used −

- **fd** − This is the file descriptor returned by fs.open().
- **buffer** − This is the buffer that the data will be written to.
- **offset** − This is the offset in the buffer to start writing at.
- **length** − This is an integer specifying the number of bytes to read.

- **position** − This is an integer specifying where to begin reading from in the file. If position is null, data will be read from the current file position.
- **callback** − This is the callback function which gets the three arguments, (err, bytesRead, buffer).

## Closing a File

### Syntax

Following is the syntax to close an opened file −

```
fs.close(fd, callback)
```

### Parameters

Here is the description of the parameters used −

- **fd** − This is the file descriptor returned by file fs.open() method.
- **callback** − This is the callback function No arguments other than a possible exception are given to the completion callback.

## Truncate a File

### Syntax

Following is the syntax of the method to truncate an opened file −

```
fs.ftruncate(fd, len, callback)
```

### Parameters

Here is the description of the parameters used −

- **fd** − This is the file descriptor returned by fs.open().
- **len** − This is the length of the file after which the file will be truncated.
- **callback** − This is the callback function No arguments other than a possible exception are given to the completion callback.

## Delete a File

### Syntax

Following is the syntax of the method to delete a file −

```
fs.unlink(path, callback)
```

### Parameters

Here is the description of the parameters used −

- **path** − This is the file name including path.
- **callback** − This is the callback function No arguments other than a possible exception are given to the completion callback.

## URL Module:

As nodejs.org suggests:

The URL module provides utilities for URL resolution and parsing. It can be accessed using:

```
1. var url = require('url');
```

Url module is one of the core modules that comes with node.js, which is used to parse the URL and its other properties.

By using URL module, it provides us with so many properties to work with.

These all are listed below:

| Property | Description |
|---|---|
| .href | Provides us the complete url string |
| .host | Gives us host name and port number |
| .hostname | Hostname in the url |
| .path | Gives us path name of the url |
| .pathname | Provides host name , port and pathname |
| .port | Gives us port number specified in url |
| .auth | Authorization part of url |
| .protocol | Protocol used for the request |
| .search | Returns query string attached with url |

As you can see in the above screen, there are various properties used for URL module.

Code Demonstrated in Class:

```
//////////////////////////////////////////////////////////////////
Opening a file:
//////////////////////////////////////////////////////////////////
var fs = require("fs");
```

```
// Asynchronous - Opening File
console.log("Going to open file!");
fs.open('input.txt', 'r+', function(err, data) {
   if (err) {
      return console.error(err);
   }
   console.log(data);
   console.log("File opened successfully!");
});


////////////////////////////////////////////////////////////////////////
Reading from file:
////////////////////////////////////////////////////////////////////////
var fs = require('fs');

fs.readFile('input.txt', function (err, data) {
                  if (err) throw err;

    console.log(data.toString());
});

var fs = require('fs');

var data = fs.readFileSync('data.txt', 'utf8');
console.log(data);

////////////////////////////////////////////////////////////////////////////////
Writing a file:
////////////////////////////////////////////////////////////////////////
var fs = require("fs");

console.log("Going to write into existing file");
fs.writeFile('input.txt', 'PES University!', function(err) {
   if (err) {
      return console.error(err);
   }

   console.log("Data written successfully!");
   console.log("Let's read newly written data");

   fs.readFile('data.txt', function (err, data) {
      if (err) {
```

```
        return console.error(err);
      }
      console.log("Asynchronous read: " + data.toString());
   });
});

////////////////////////////////////////////////////////////////////
Reading from a file
////////////////////////////////////////////////////////////////////

var fs = require('fs');

fs.open('data.txt', 'r', function (err, fd) {

                        if (err) {
                        return console.error(err);
    }

                        var buffr = new Buffer(1024);
   console.log("File opened successfully!");
  console.log("Going to truncate the file after 3 bytes");

  // Truncate the opened file.
  fs.ftruncate(fd, 3, function(err) {
     if (err) {
        console.log(err);
     }
     console.log("File truncated successfully.");
     console.log("Going to read the same file");

    fs.read(fd, buffr, 0, buffr.length, 0, function (err, bytes) {

                        if (err) throw err;

                        // Print only read bytes to avoid junk.
                        if (bytes > 0) {
           console.log(buffr.slice(0, bytes).toString());
       }

                        // Close the opened file.
       fs.close(fd, function (err) {
                        if (err) throw err;
       });
```

```
    });
});

////////////////////////////////////////////////////////////////////////////
Delete a file
////////////////////////////////////////////////////////////////////////////
var fs = require('fs');

fs.unlink('data.txt', function () {

    console.log('write operation complete.');

});
```

## HTTP Module

Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).

To include the HTTP module, use the require() method:

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

To process HTTP requests in JavaScript and Node.js, we can use the built-in http module. This core module is key in leveraging Node.js networking and is extremely useful in creating HTTP servers and processing HTTP requests.

The http module comes with various methods that are useful when engaging with HTTP network requests. One of the most commonly used methods within the http module is the .createServer() method. This method is responsible for doing exactly what its namesake implies; it creates an HTTP server. To implement this method to create a server, the following code can be used:

```
const server = http.createServer((req, res) => {
  res.end('Server is running!');
});

server.listen(8080, () => {
  const { address, port } = server.address();
  console.log(`Server is listening on: http://${address}:${port}`);
})
```

The .createServer() method takes a single argument in the form of a callback function. This callback function has two primary arguments; the request (commonly written as req) and the response (commonly written as res).

The req object contains all of the information about an HTTP request ingested by the server. It exposes information such as the HTTP method (GET,POST, etc.), the pathname, headers, body, and so on. The res object contains methods and properties pertaining to the generation of a response by the HTTP server. This object contains methods such as .setHeader (sets HTTP headers on the response), .statusCode (set the status code of the response), and .end() (dispatches the response to the client who made the request). In the example above, we use the .end() method to send the string 'Server is Running!' to the client, which will display on the web page.

Once the .createServer() method has instantiated the server, it must begin listening for connections. This final step is accomplished by the .listen() method on the server instance. This method takes a port number as the first argument, which tells the server

to listen for connections at the given port number. In our example above, the server has been set to listen on port 8080. Additionally, the .listen() method takes an optional callback function as a second argument, allowing it to carry out a task after the server has successfully started.

## HTTP Module: Some Important Methods

| | |
|---|---|
| http.STATUS_CODES; | A collection of all the standard HTTP response status codes, and the short description of each. |
| http.request(options, [callback]); | This function allows one to transparently issue requests. |
| http.get(options, [callback]); | Set the method to GET and calls req.end() automatically. |
| server = http.createServer([requestListener]); | Returns a new web server object. The requestListener is a function which is automatically added to the 'request' event. |
| server.listen(port, [hostname], [backlog], [callback]); | Begin accepting connections on the specified port and hostname. |
| server.close([callback]); | Stops the server from accepting new connections. |
| request.write(chunk, [encoding]); | Sends a chunk of the body. |
| request.end([data], [encoding]); | Finishes sending the request. If any parts of the body are unsent, it will flush them to the stream. |
| request.abort(); | Aborts a request. |
| response.write(chunk, [encoding]); | This sends a chunk of the response body. If this merthod is called and response.writeHead() has |

| | |
|---|---|
| | not been called, it will switch to implicit header mode and flush the implicit headers. |
| response.writeHead(statusCode, [reasonPhrase], [headers]); | Sends a response header to the request. |
| response.getHeader(name); | Reads out a header that's already been queued but not sent to the client. Note that the name is case insensitive. |
| response.removeHeader(name); | Removes a header that's queued for implicit sending. |
| response.end([data], [encoding]); | This method signals to the server that all of the response headers and body have been sent; that server should consider this message complete. The method, response.end(), MUST be called on each response. |

## Handling HTTP Requests

In the following section we will learn to handle HTTP request when we open http://localhost:9000 url.

The callback function that we saw above when we created the server has two parameters namely, request and response.

The request object has url property and it holds the requested url. So, we will use it.

The response object has writeHead method which helps in setting the header. Then we have the write method that helps to write the response body. And there is end method which helps in sending the response.

Lets say, we want to print out the message Hello World! when we visit the home page at http://localhost:9000. As, it is the home page so we have to check for /.

```
const http = require('http');
const PORT = 9000;
```

```
const server = http.createServer(function(request, response) {
    // we are setting the header
    const header = {
        'Content-Type': 'text/html'
    };
    // checking for home page url /
    if (request.url === '/') {
        response.writeHead(200, header);
        response.write('Hello World!');
    }
    // if requested url is not known then prompt error response
    else {
        response.writeHead(400, header);
        response.write('Bad request!');
    }
    response.end();
});
server.listen(PORT);
console.log(`Server started and listening at port ${PORT}...`);
```

Now, stop the running server in the terminal and re-run it. Now, visit the home page http://localhost:9000 url. This time we will see the message Hello World!

This happens because the if condition is satisfied and if-block is executed.

Now, if we visit something like http://localhost:9000/some-unknown-page then we will get Bad Request message.

This happens because we are not handling the /some-unknown-page and hence the else-block is executed which gives the Bad request! message as response.

## Making HTTP Requests

The request() method supports multiple function signatures. You'll use this one for the subsequent example:

```
http.request(Options_Object, Callback_Function)
```

The first argument is a string with the API endpoint. The second argument is a JavaScript object containing all the options for the request. The last argument is a callback function to handle the response.

## Making HTTP Requests: Options

| | |
|---|---|
| host | A domain name or IP address of the server to issue the request to. Defaults to 'localhost'. |
| hostname | To support url.parse() hostname is preferred over host |
| port | Port of the remote server. Defaults to 80. |
| method | A string specifying the HTTP request method. Defaults to 'GET'. |
| path | Request path. Defaults to '/'. Should include query string if any. E.G. '/index.html?page=12' |
| headers | An object containing request headers. |
| auth | Basic authentication i.e. 'user:password' to compute an Authorization header. |

```
const https = require('https')
const options = {
  hostname: 'example.com',
  port: 443,
  path: '/todos',
  method: 'GET'
}

const req = https.request(options, res => {
  console.log(`statusCode: ${res.statusCode}`)

 res.on('data', d => {
    process.stdout.write(d)
  })
})

req.on('error', error => {
  console.error(error)
})

req.end()
```

## MongoDB and Its Features

**MongoDB** is a document-oriented NoSQL database used for high volume data storage. Instead of using tables and rows as in the traditional relational databases, MongoDB makes use of collections and documents. Documents consist of key-value pairs which are the basic unit of data in MongoDB. Collections contain sets of documents and function which is the equivalent of relational database tables. MongoDB is a database which came into light around the mid-2000s

1. Each database contains collections which in turn contains documents. Each document can be different with a varying number of fields. The size and content of each document can be different from each other.
2. The document structure is more in line with how developers construct their classes and objects in their respective programming languages. Developers will often say that their classes are not rows and columns but have a clear structure with key-value pairs.
3. The rows (or documents as called in MongoDB) doesn't need to have a schema defined beforehand. Instead, the fields can be created on the fly.
4. The data model available within MongoDB allows you to represent hierarchical relationships, to store arrays, and other more complex structures more easily.
5. Scalability – The MongoDB environments are very scalable. Companies across the world have defined clusters with some of them running 100+ nodes with around millions of documents within the database

## Working with a Collection

You can perform operations using the `db` variable. Every collection has a property on the `db` variable - e.g. the `apples` collection is `db.apples`.

| JavaScript Database Operations | Description |
|---|---|
| `db.apples.find()` | Finds all documents in a collection named "apples". |
| `db.apples.find({type : "granny smith"})` | Finds all documents in a collection called "apples" with type matching "granny smith". |

| | |
|---|---|
| `db.apples.remove({ bad : true})` | Removes all bad apples! Finds all apples with a property of "bad" set to true, and removes them. |
| `db.apples.update({ type : "granny smith"}, {$set : { price : 2.99 }})`<br><br>`#don't do this!`<br>`db.apples.update({ type : "granny smith"}, { price : 2.99 })` | Updates the price of all granny smith apples.<br><br>**Important!** Note the $set syntax - this doesn't work as many would expect. Without putting the updated fields inside a $set clause, we replace the entire document. |
| `db.apples.insert({ type : "granny smith", "price" : 5.99 })` | Inserts a new document into the apples collection.<br>If your apples collection doesn't exist, it will get created. |
| `db.apples.findOne({ _id : ObjectId ("54324a5925859afb491a0000") })` | Looking up a document by ID is special. We can't just specify the ObjectID as a string - we need to cast it to an ObjectId. |

## MongoDB Connectivity: Create a Database

To create a database in MongoDB, start by creating a MongoClient object, then specify a connection URL with the correct ip address and the name of the database you want to create.

MongoDB will create the database if it does not exist, and make a connection to it.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/mydb";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  console.log("Database created!");
  db.close();
});
```

## MongoDB Connectivity: Create a Collection

To create a collection in MongoDB, use the createCollection() method:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
```

```
MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.createCollection("customers", function(err, res) {
    if (err) throw err;
    console.log("Collection created!");
    db.close();
  });
});
```

## MongoDB Connectivity: Insert into Collection

To insert a record, or *document* as it is called in MongoDB, into a collection, we use the insertOne() method.

A **document** in MongoDB is the same as a **record** in MySQL

The first parameter of the insertOne() method is an object containing the name(s) and value(s) of each field in the document you want to insert.

It also takes a callback function where you can work with any errors, or the result of the insertion:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myobj = { name: "Company Inc", address: "Highway 37" };
  dbo.collection("customers").insertOne(myobj, function(err, res) {
    if (err) throw err;
    console.log("1 document inserted");
    db.close();
  });
});
```

## MongoDB Connectivity: Read

To select data from a table in MongoDB, we can also use the find() method. The find() method returns all occurrences in the selection. The first parameter of the find() method is a query object. In this example we use an empty query object, which selects all documents in the collection.

No parameters in the find() method gives you the same result as **SELECT *** in MySQL.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.collection("customers").find({}).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

## MongoDB Connectivity: Filter results

When finding documents in a collection, you can filter the result by using a query object. The first argument of the find() method is a query object, and is used to limit the search.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var query = { address: "Park Lane 38" };
  dbo.collection("customers").find(query).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

# Lesson 5: HTTP web module

**HTTP Module: - https://nodejs.org/api/http.html**

A Web Server is a software application which handles HTTP requests sent by the HTTP client, like web browsers, and returns web pages in response to the clients.

Web servers usually deliver html documents along with images, style sheets, and scripts.

Most of the web servers support server-side scripts, using scripting languages or redirecting the task to an application server which retrieves data from a database and performs complex logic and then sends a result to the HTTP client through the Web server.

**Apache web server is one of the most commonly used web servers. It is an open source project.**

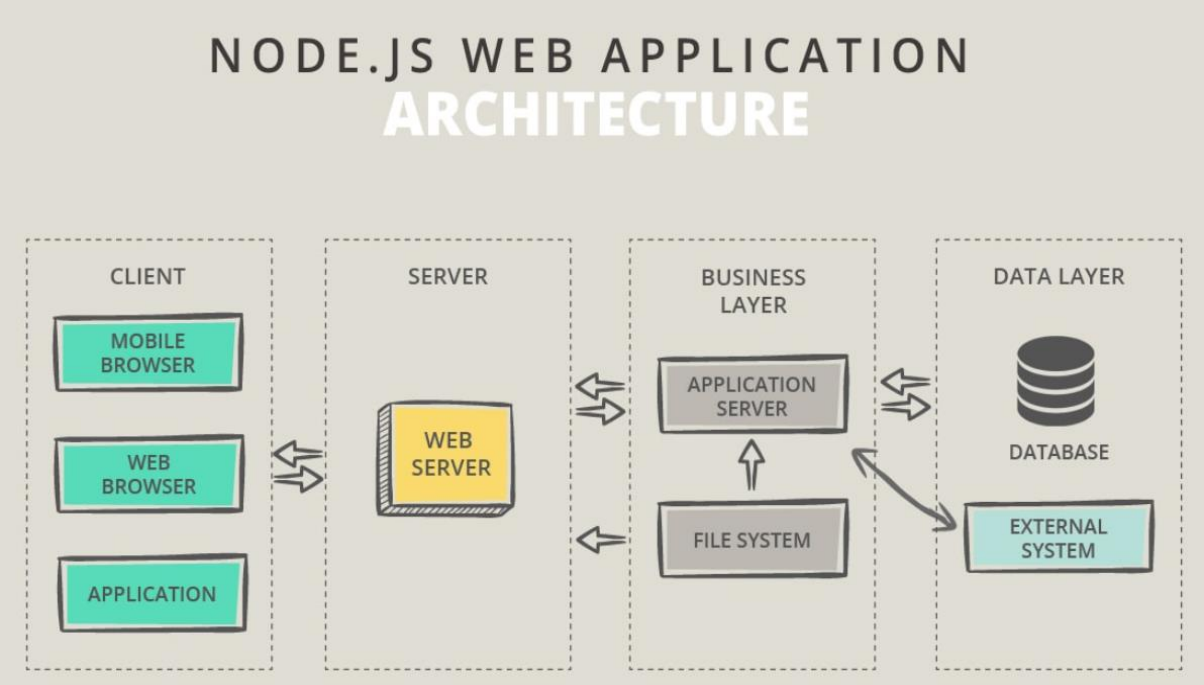
A Web application is usually divided into four layers −

**Client** − This layer consists of web browsers, mobile browsers or applications which can make HTTP requests to the web server.

**Server** − This layer has the Web server which can intercept the requests made by the clients and pass them the response.

**Business** − This layer contains the application server which is utilized by the web server to do the required processing. This layer interacts with the data layer via the database or some external programs.

**Data** − This layer contains the databases or any other source of data.




## Node.js Web Server

- To access web pages of any web application, you need a web server.
- The web server will handle all the http requests for the web application
- e.g IIS is a web server for ASP.NET web applications and Apache is a web server for PHP or Java web applications.
- Node.js provides capabilities to create your own web server which will handle HTTP requests asynchronously.
- You can use IIS or Apache to run Node.js web application but it is recommended to use Node.js web server.

The components of a Node.js application.

**Import required modules** − We use the require directive to load Node.js modules.

**Create server** − A server which will listen to client's requests similar to Apache HTTP Server. Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).

Read request and return response − The server created in an earlier step will read the HTTP request made by the client which can be a browser or a console and return the response.

Creating Node.js Application

**Step 1 - Import Required Module**

We use the require directive to load the http module and store the returned HTTP instance into an http variable as follows

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

**Step 2 - Create Server**

- We use the created http instance and call **http.createServer()** method to create a server instance.
- Then we bind it at port 8088 using the **listen** method associated with the server instance.

```
http.createServer(function (request, response) {
    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.end('Hello PES University\n');
 }).listen(8088);
  console.log('Server running at http://127.0.0.1:8088/')
```

**Step 3 - Testing Request & Response**

$node app.js

Verify the Output. Server has started.

**HTTP Module – Web Server using Node**

**Server running at http://127.0.0.1:8088/**

**var http = require('http'); // 1 - Import Node.js core module**

**var server = http.createServer(function (req, res) {  // 2 - creating server**

**  //handle incomming requests here..**

**});**

**server.listen(5000); //3 - listen for any incoming requests**

**console.log('Node.js web server at port 5000 is running..')**

In the above example, we import the http module using require() function.

The http module is a core module of Node.js, so no need to install it using NPM.

The next step is to call createServer() method of http and specify callback function with request and response parameter.

Finally, call listen() method of server object which was returned from createServer() method with port number, to start listening to incoming requests on port 5000.

You can specify any unused port here.

## Handle HTTP Request

The http.createServer() method includes request and response parameters which is supplied by Node.js.

**The request object can be used to get information about the current HTTP request e.g., url, request header, and data.**

**The response object can be used to send a response for a current HTTP request.**

**Ex:**

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

//create a server object:

http.createServer(function (req, res) {

  res.write('Hello World!'); //write a response to the client

  res.end(); //end the response

}).listen(8080); //the server object listens on port 8080
```

The module provides some properties and methods, and some classes.

## Properties

### http.METHODS

This property lists all the HTTP methods supported:

```
> require('http').METHODS
```

### http.STATUS_CODES

This property lists all the HTTP status codes and their description:

```
> require('http').STATUS_CODES
```

### http.globalAgent

It points to the global instance of the Agent object, which is an instance of the http.Agent class.

It is used to manage connections persistence and reuse for HTTP clients, and it's a key component of Node.js HTTP networking.

## Methods

### http.createServer()

Return a new instance of the http.Server class.

**Usage:**

```
const server = http.createServer((req, res) => {
  //handle every single request with this callback
})
```

### http.request()

Makes an HTTP request to a server, creating an instance of the http.ClientRequest class.

### http.get()

Similar to http.request(), but automatically sets the HTTP method to GET, and calls req.end() automatically.

## Classes

The HTTP module provides 5 classes:

- http.Agent
- http.ClientRequest
- http.Server
- http.ServerResponse
- http.IncomingMessage

### http.Agent

Node.js creates a global instance of the http.Agent class to manage connections persistence and reuse for HTTP clients, a key component of Node.js HTTP networking.

This object makes sure that every request made to a server is queued and a single socket is reused.

It also maintains a pool of sockets. This is key for performance reasons.

### http.ClientRequest

An http.ClientRequest object is created when http.request() or http.get() is called.

When a response is received, the response event is called with the response, with an http.IncomingMessage instance as argument.

The returned data of a response can be read in 2 ways:

- you can call the response.read() method
- in the response event handler you can setup an event listener for the data event, so you can listen for the data streamed into.

### http.Server

This class is commonly instantiated and returned when creating a new server using http.createServer().

Once you have a server object, you have access to its methods:

- listen() starts the HTTP server and listens for connections
- close() stops the server from accepting new connections

**http.ServerResponse**

Created by an http.Server and passed as the second parameter to the request event it fires.

Commonly known and used in code as res:

```
const server = http.createServer((req, res) => {
  //res is an http.ServerResponse object
})
```

The method you'll always call in the handler is end(), which closes the response, the message is complete and the server can send it to the client. It must be called on each response.

These methods are used to interact with HTTP headers:

- getHeaderNames() get the list of the names of the HTTP headers already set
- getHeaders() get a copy of the HTTP headers already set
- setHeader('headername', value) sets an HTTP header value
- getHeader('headername') gets an HTTP header already set
- removeHeader('headername') removes an HTTP header already set
- hasHeader('headername') return true if the response has that header set
- headersSent() return true if the headers have already been sent to the client

After processing the headers you can send them to the client by calling response.writeHead(), which accepts the statusCode as the first parameter, the optional status message, and the headers object.

To send data to the client in the response body, you use write(). It will send buffered data to the HTTP response stream.

**http.IncomingMessage**

An http.IncomingMessage object is created by:

- http.Server when listening to the request event
- http.ClientRequest when listening to the response event

It can be used to access the response:

- status using its statusCode and statusMessage methods

- headers using its headers method or rawHeaders
- HTTP method using its method method
- HTTP version using the httpVersion method
- URL using the url method
- underlying socket using the socket method

The data is accessed using streams, since http.IncomingMessage implements the Readable Stream interface.

**Class code Demonstration:**

**Client.js**

```
var http = require('http');
var fetch = require('node-fetch');

//options to be used by request
/*
var options =
{
host: 'localhost',
port: '8081',
path : '/pes.html',
};

//callback function is used to deal with the response

var callback = function(response)
{
   var body = '';
   response.on('data',function(data)
   {

      body+= data;
   });
}
//make a request to the server
var req = http.request(options,callback);
req.end();  */


/*fetch('http://localhost:8081/sample.json', {
```

```
        method: 'GET',
        headers: { 'Content-Type': 'application/json' },
    })
    .then(res => res.json())
    .then(res => console.log(res));
*/
    fetch('http://localhost:8081/sample.json', {
        method: 'POST',
        body:    JSON.stringify({"name":"Aruna modified","col":"MIT"}),
        headers: { 'Content-Type': 'application/json' },
    })
     .then(res => console.log(res));
```

**Server.js**

```
// http module ccreating a web server
var url = require('url');
var http = require('http');
var fs = require('fs');
var MongoClient = require('mongodb').MongoClient;

//create a server
http.createServer(function (request, response) {

    //parse the request containing the filename
    var pathname = url.parse(request.url).pathname;


    //print the name of the file for which request is made
    console.log("Request for" + pathname + "Received.");

    if (request.method == "GET") {
        //Read the requested file content from filesystem
        // fs.readFile(pathname.substr(1), function (err, data) {
        //     if (err) {
        //         console.log(err);
        //         //HTTP Status 404 not found
        //         response.writeHead(404, { 'Content-Type': 'text/html' });
        //     }
        //     else {
        //         //page found and status of 200 has to be returned
```

```
    //         response.writeHead(200, { 'Content-Type': 'text/html' });

    //         //write the content of the file to response body
    //         response.write(data.toString());
    //     }
    //     //send the responseBody
    //     response.end();

    // });
    //Read from the mongodb
    console.log('executing mongo');
    MongoClient.connect("mongodb://localhost:27017", {
        useUnifiedTopology:
            true
    }, function (err, client) {
        console.log("Connected successfully to server");
        const db = client.db("pes");
        db.collection("student").find({}).toArray(function (err, docs) {
            response.writeHead(200, { 'Content-Type': 'application/json' });
            //write the content of the file to response body
            response.write(JSON.stringify(docs));
            client.close();
            response.end();
        });
    });

}
else {
    let body = [];
    request.on('data', (chunk) => {
        body.push(chunk);
    }).on('end', () => {
        body = Buffer.concat(body).toString();
        // at this point, `body` has the entire request body stored in it as a string
    });

    // write to file
    // fs.writeFile(pathname.substr(1), body, (err, res) => {
    //     response.writeHead(200, { 'Content-Type': 'application/json' });
    //     response.end();
    // });
```

```
        //write to mongodb
        console.log('executing mongo');
        MongoClient.connect("mongodb://localhost:27017", {
            useUnifiedTopology:
                true
        }, function (err, client) {
            console.log("Connected successfully to server");
            const db = client.db("pes");
            db.collection("student").insertOne(JSON.parse(body)).then(r => {
                response.writeHead(200, { 'Content-Type': 'application/json' });
                //write the content of the file to response body
                client.close();
                response.end();
            })
        });
    }
}).listen(8081);
console.log('server running at the link http://localhost/8081');
```

**Read the Query String**

The function passed into the http.createServer() has a req argument that represents the request from the client, as an object (http.IncomingMessage object).

This object has a property called "url" which holds the part of the url that comes after the domain name:

demo_http_url.js

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

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write(req.url);   //requesting URL
  res.end();
}).listen(8080);
```

## URL Module:

The URL module splits up a web address into readable parts.

To include the URL module, use the require() method:

Parse an address with the url.parse() method, and it will return a URL object with each part of the address as properties:

```
var url = require('url');
```

Parse an address with the url.parse() method, and it will return a URL object with each part of the address as properties:

Node Examples > JS app.js > ...

```
1   var url = require('url');
2   var adr = 'http://localhost:8080/pesu.htm?year=2020&month=September';
3   var q = url.parse(adr, true);
4
5   console.log(q.host); //returns 'localhost:8080'
6   console.log(q.pathname); //returns '/pesu.htm'
7   console.log(q.search); //returns '?year=2020&month=September'
8
9   var qdata = q.query; //returns an object: { year: 2020, month: 'september' }
10  console.log(qdata.month); //returns 'september'
```