LAB # 12

Statement Purpose:

To familiarize the students with

- React Hooks (useState, useEffect, useContext), Installation of Node.js, Express.js
- Sign Up Application (React, Node.js, Express and MongoDB)

React Hooks

Hooks allow function components to have access to state and other React features.

useState Example

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

function CountApp() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
export default CountApp;
```

In the above example,

useState is the Hook which needs to call inside a function component to add some local state to it. The **useState** returns a pair where the first element is the current state value/initial value, and the second one is a function which allows us to update it. After that, we will call this function from an event handler or somewhere else.

The **useState** is similar **to this.setState** in class. The equivalent code without Hooks

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

class CountApp extends React.Component {
  constructor(props) {
    super(props);
```

```
    this.state = {
      count: 0
    };
  }
  render() {
    return (
      <div>
        <p><b>You clicked {this.state.count} times</b></p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
export default CountApp;
```

Hooks Effect

The Effect Hook allows us to perform side effects (an action) in the function components. It does not use components lifecycle methods which are available in class components. In other words, Effects Hooks are equivalent to **componentDidMount(), componentDidUpdate(),** and **componentWillUnmount()** lifecycle methods.

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

function CounterExample() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
export default CounterExample;
```

Custom Hooks

- A custom Hook is a JavaScript function. The name of custom Hook starts with "**use**" which can call other Hooks.
- A custom Hook is just like a regular function, and the word "**use**" in the beginning tells that this function follows the rules of Hooks.
- Building custom Hooks allows you to extract component logic into reusable functions.

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

const useDocumentTitle = title => {
  useEffect(() => {
    document.title = title;
  }, [title])
}

function CustomCounter() {
  const [count, setCount] = useState(0);
  const incrementCount = () => setCount(count + 1);
  useDocumentTitle(`You clicked ${count} times`);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={incrementCount}>Click me</button>
    </div>
  )
}
export default CustomCounter;
```

React Context

- React Context is a way to manage state globally.
- It can be used together with the **useState** Hook to share state between deeply nested components more easily than with **useState** alone.

Example: Passing "props" through nested components:

```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function Component1() {
  const [user, setUser] = useState("Web Developer");

  return (
    <>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 user={user} />
    </>
  );
}
```

```
function Component2({ user }) {
  return (
    <>
      <h1>Operating System</h1>
      <Component3 user={user} />
    </>
  );
}

function Component3({ user }) {
  return (
    <>
      <h1>Machine Learning</h1>
      <Component4 user={user} />
    </>
  );
}

function Component4({ user }) {
  return (
    <>
      <h1>Thoery of Automata</h1>
      <Component5 user={user} />
    </>
  );
}

function Component5({ user }) {
  return (
    <>
      <h1>English Comprehension</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Component1 />);

export default Component1;
```

Example: using React Context:

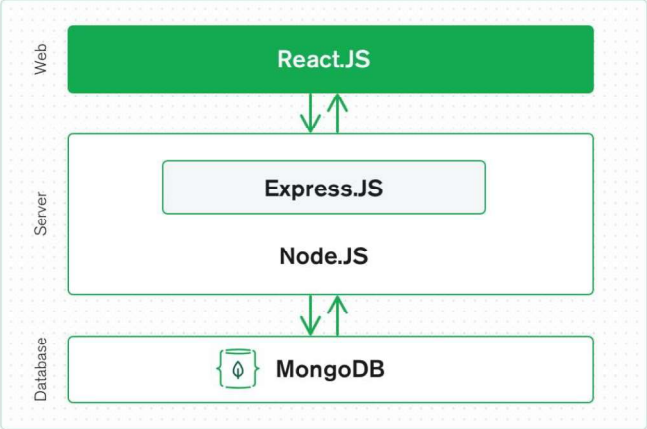
```
import { useState, createContext, useContext } from "react";
import ReactDOM from "react-dom/client";

const UserContext = createContext();
```

```
function Component1() {
  const [user, setUser] = useState("Web Developer");

  return (
    <UserContext.Provider value={user}>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 user={user} />
    </UserContext.Provider>
  );
}

function Component2() {
  return (
    <>
      <h1>Operating System</h1>
      <Component3 />
    </>
  );
}

function Component3() {
  return (
    <>
      <h1>Theory of Automata</h1>
      <Component4 />
    </>
  );
}

function Component4() {
  return (
    <>
      <h1>Wireless communication</h1>
      <Component5 />
    </>
  );
}

function Component5() {
  const user = useContext(UserContext);

  return (
    <>
      <h1>Analysis of Algorithms</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
```

```
root.render(<Component1 />);

export default Component1;
```

How does the MERN stack work?

The MERN architecture allows you to easily construct a 3-tier architecture (frontend, backend, database) entirely using JavaScript and JSON.



Node(.js) and Express(.js)

Express is a framework that provides basic features for developing a web application in Node.js. While Node.js is already capable of listening to requests on a port, Express makes it simpler to set up a web server by eliminating boilerplate and offering a simpler API for creating endpoints.

Instead of writing full web server code by hand on Node.js directly, developers use Express to simplify the task of writing server code. There's no need to repeat the same code over and over, as you would with the Node.js HTTP module.

- Open **node.js** command prompt
- Write commands
  - **node -v** or **process.version** in node.js terminal
- install express (get us to build express apps)
  - npm install -g express-generator
- write command to generate app
  - express fswd
  - cd fswd
  - npm install
  - express fswd --hogan -c less (hogan template and -c is for css)
  - cd fswd && npm ustall
  - DEBUG=fswd:* & npm start
  - http://localhost:3000/

- change port or app name in ./bin/www
    - npm install -g nodemon

The **nodemon** Module is a module that develop node.js based applications by automatically restarting the node application when file changes in the directory are detected. Nodemon does not require any change in the original code and method of development.

index.js file

```
router.get('/', function(req, res)
{
  res.send('ok');
});
```

Index.js

```
<!DOCTYPE html>
<html>
  <head>
    <title>{{ title }}</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1>{{ title }}</h1>
    <p>He is  {{ age }} years old</p>
  </body>
</html>
```

Index.js

```
var express = require('express');
var router = express.Router();

router.get('/', function(req, res)
{
  res.render('index', {
    title: 'My App',
    age: 31
  })
});
module.exports = router;
```