IT3501- FULL STACK WEB DEVELOPMENT

UNIT-5: REACT

MERN STACK – Basic React applications – React Components – React State – Express REST APIs - Modularization and Webpack - Routing with React Router – Server-side rendering

Basics

The MERN stack is a technology stack used for building modern web applications. It's an acronym that represents four key technologies:

- MongoDB
- Express
- React
- Node.js

MongoDB:

- MongoDB is a NoSQL database that stores data in a flexible, JSON-like format.
- It is widely used for its scalability and ease of use.
- MongoDB offers powerful query capabilities and supports various indexing options, including geospatial and text-based search, making it versatile for different use cases.

Express:

- Express is a backend framework for Node.js that simplifies building web APIs.
- It is known for its simplicity and robustness.
- Express is known for its performance and is well-suited for building scalable and high-performance web servers and applications.

React:

- React is a JavaScript library for building user interfaces.
- Components are the building blocks in React.
- React uses a virtual DOM to efficiently update the actual DOM, reducing the need for unnecessary re-renders and improving performance.

Node.js:

- Node.js is a runtime environment for running server-side JavaScript.
- It is built on the V8 JavaScript engine.
- Non-blocking I/O for real-time and high-concurrency apps.
- Ideal for building scalable and event-driven server applications.

Why MERN stack?

- MERN uses JavaScript for both frontend and backend, streamlining development and promoting consistency.
- React's component architecture simplifies UI development, enhancing reusability and maintainability.
- MERN components like MongoDB and Node.js support efficient scaling for growing applications.
- Benefit from a thriving community and a rich ecosystem of libraries and resources.
- MERN, particularly React, excels at building responsive single-page applications for a smoother user experience.

Basic React Application

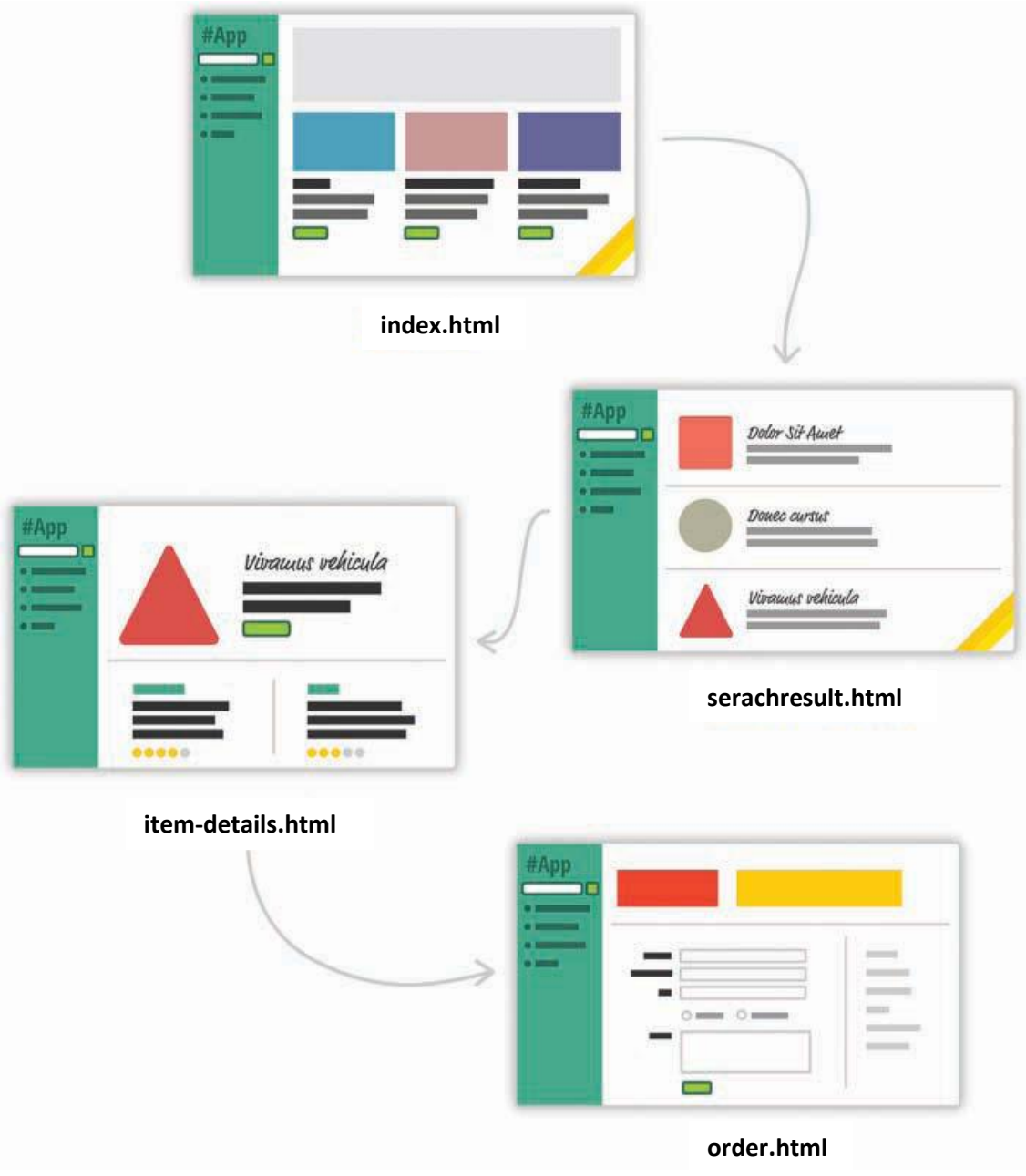
React:

React anchors the MERN stack. In some sense, this is the defining component of the MERN stack. React is an open-source JavaScript library maintained by Facebook that can be used for creating views rendered in HTML. Unlike AngularJS, React is not a framework. It is a library. Thus, it does not, by itself, dictate a framework pattern such as the MVC pattern. You use React to render a view (the V in MVC), but how to tie the rest of the application together is completely up to you.

Not just Facebook itself, but there are many other companies that use React in production like Airbnb, Atlassian, Bitbucket, Disqus, Walmart, etc. The 120,000 stars on its GitHub repository is an indication of its popularity.

Multipage Design (old-School):

If you had to build this app a few years ago, you might have taken an approach that involved multiple, individual pages.

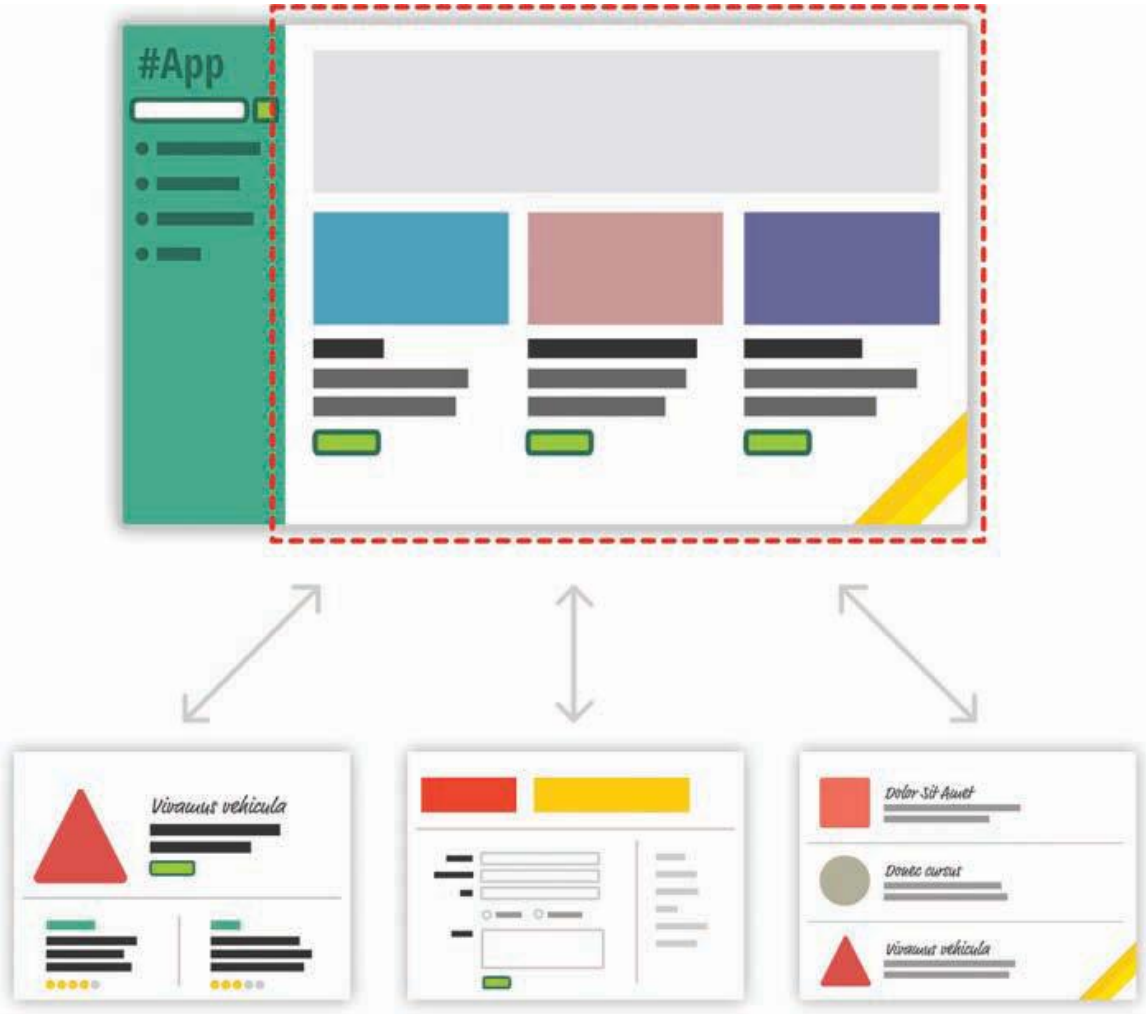


For almost every action that changes what the browser displays, the web app navigates you toa whole different page. This is a big deal, beyond just the less-than-stellar user experience users will see as pages get torn down and redrawn.

Single-Page Apps (New-School):

These days, going with a web app model that requires navigating between individual pages seems dated.

Instead, modern apps tend to adhere to what is known as a single-page app (SPA) model. This model gives you a world in which you never navigate to different pages or ever even reload a page. In this world, the different views of your app are loaded and unloaded into the same page itself.



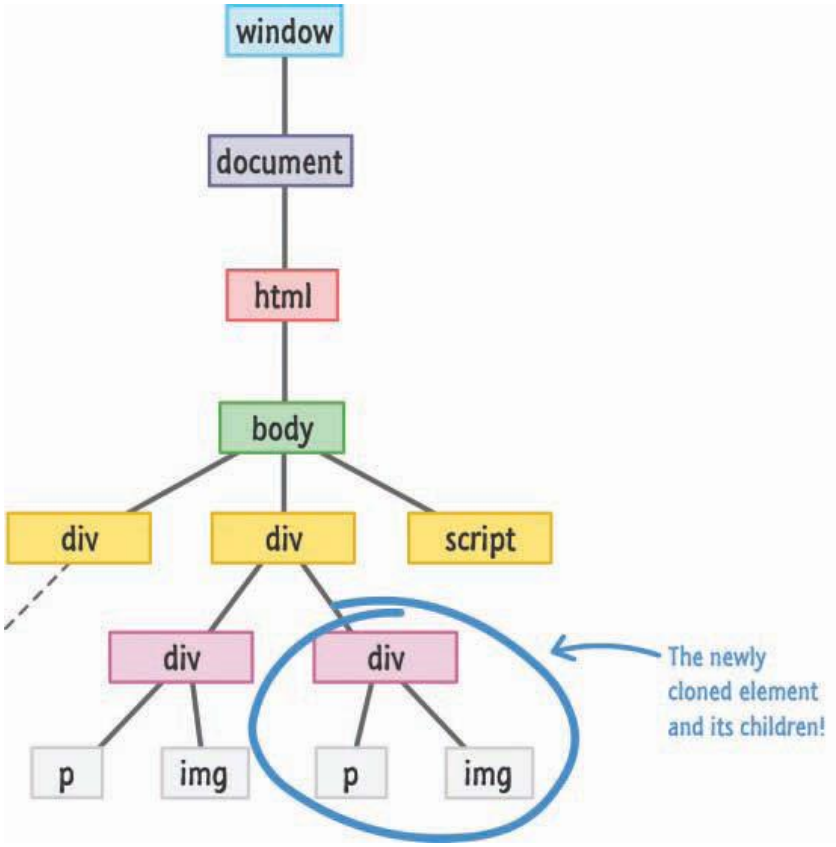
As users interact with our app, we replace the contents of the dotted red region with the data and HTML that matches what the user is trying to do. The end result is a much more fluid experience. You can even use a lot of visual techniques to have your new content transition nicely, just like you might see in cool apps on your mobile device or desktop. This sort of stuff is simply not possible when navigating to different pages.

When building single-page apps, you'll encounter three major issues at some point:

1. In a single-page application, you'll spend the bulk of your time keeping your data in sync with your UI. For example, if a user loads new content, do you explicitly clear out the search field? Do you keep the active tab on a navigation element still visible? Which elements do you keep on the page, and which do you destroy? These are all problems that are unique to single-page apps. When navigating

between pages in the old model, we assumed everything in our UI would be destroyed and just built back up again. This was never a problem.

2. Manipulating the DOM is really, really slow. Manually querying elements, adding children (see Figure 1.5), removing subtrees, and performing other DOM operations is one of the slowest things you can do in your browser. Unfortunately, in a single-page app, you'll be doing a lot of this. Manipulating the DOM is the primary way you are able to react to user actions and display new content.



3. Working with HTML templates can be a pain. Navigation in a single-page app is nothing more than you dealing with fragments of HTML to represent whatever you want to display. These fragments of HTML are often known as templates, and using JavaScript to manipulate them and fill them out with data gets really complicated really quickly.

Features of React:

- **Automatic UI State Management:**

With single-page apps, keeping track of your UI and maintaining state is hard … and also very time consuming.

- **Lightning-Fast DOM Manipulation**

Because DOM modifications are really slow, you never modify the DOM directly using React. Instead, you modify an in-memory **virtual** DOM.

Manipulating this virtual DOM is extremely fast, and React takes care of updating the real DOM when the time is right. It does so by comparing the changes between your virtual DOM and the real DOM, figuring out which changes actually matter, and making the fewest number of DOM changes needed to keep everything up-to-date in a process called reconciliation.

- **APIs to Create Truly Composable UIs**

Instead of treating the visual elements in your app as one monolithic chunk, React encourages you to break your visual elements into smaller and smaller components. As with everything else in programming, it's a good idea to make things modular, compact, and self-contained. React extends that well-worn idea to how we think about user interfaces.

- **Visuals Defined Entirely in JavaScript**

Using JSX, you are able to easily define your visuals using a very familiar syntax, while still getting all the power and flexibility that JavaScript provides.

Creating a basic React application:

Creating a basic React application involves setting up a development environment, creating a simple React component, and rendering it to the DOM. Here are the steps to create a basic React application:

Step 1: Set Up Your Development Environment

- Before you start, ensure you have Node.js and npm (Node Package Manager) installed.
- You can check if you have them installed by running these commands in your terminal:

  node -v

  npm -v

- If they are not installed, you can download and install them from the official Node.js website **(https://nodejs.org/).**

Step 2: Create a New React Application

- You can create a new React application using Create React App, which is a tool that sets up a new React project with the necessary configuration. Open your terminal and run the following command:

  npx create-react-app first-react-app

- Replace "my-react-app" with the name of your project. This command will create a new directory with the same name in your current folder and set up a basic React application inside it.

```
D:\Projects>npx create-react-app first-react-app

Creating a new React app in D:\Projects\first-react-app.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...

[                ] / idealTree:mini-css-extract-plugin: timing idea
```

Step 3: Navigate to Your Project Directory

- Once the setup is complete, navigate to your project directory:

  cd first-react-app

Step 4: Start the Development Server

- To start the development server and see your React application in action, run the following command:

  npm start

- This command will start the development server and open your React application in a web browser.

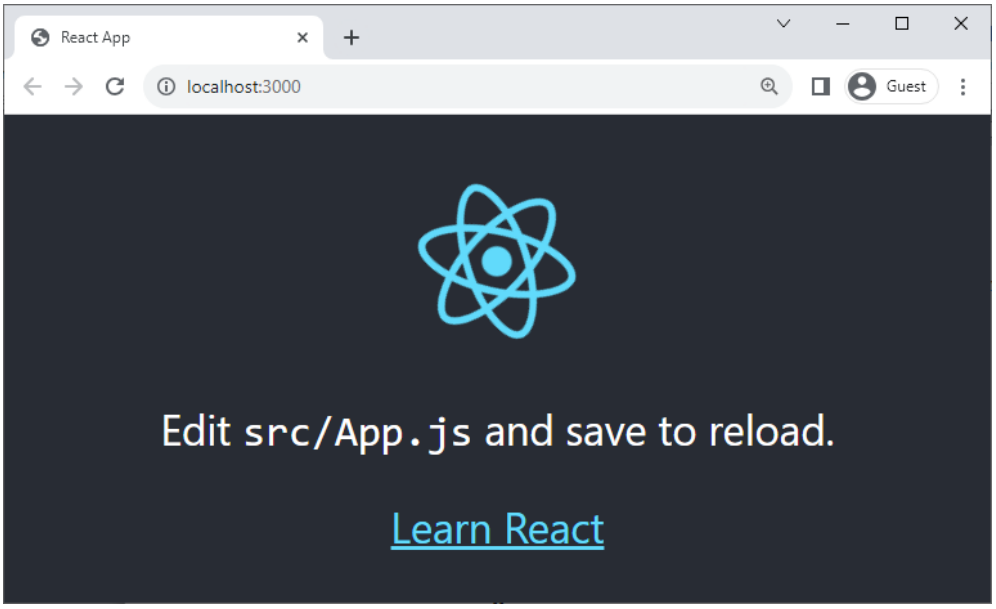
```
Compiled successfully!

You can now view first-react-app in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://172.16.159.4:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
```



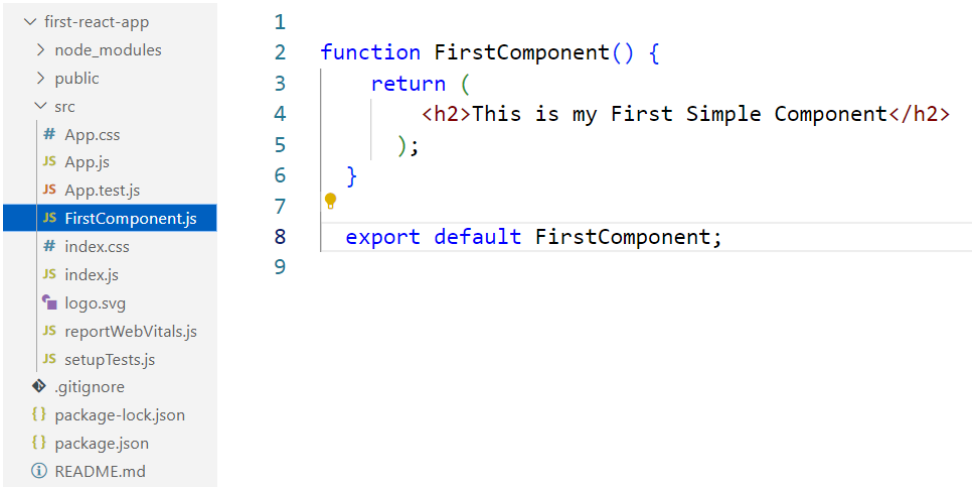
Step 5: Create a Simple React Component

- Delete all the content inside the function App().
- In the "src" folder of your project, You can create a ***javascript*** the file to create a simple React component. Here's an example:
- Add the following Code : File name: **`FirstComponent.js`**

```
function FirstComponent() {
    return (
        <h2>This is my First Simple Component</h2>
      );
  }

export default FirstComponent;
```

VS code Looks like



- Add the following Code in **`App.js`** (import the ***FirstComponent*** inside it)

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

function App() {
  return (
      <FirstComponent></FirstComponent>
    );
}
export default App;
```

Step 6: Render Your Component

- To render your component, open the "src/index.js" file and import the filename.

Here, you import your "App" component and render it inside the "root" element in your HTML.

It automatically created during create-react-app command.

Step 7: View Your React Application

- Save the changes and refresh your browser. You should see your basic React application.

In http://localhost:3000/

React Components

Components are one of the **pieces that make** React, They are one of the **primary** ways we have for defining the visuals and interactions that make up what people see when they use our app.

React components are ***reusable chunks*** of JavaScript that output (via JSX) HTML elements.

In the real world, you will want to do much more than what a simple single-line JSX can do. That is where React components come in. React components can be composed using other components and basic HTML elements; they can respond to user input, change state, interact with other components, and much more.

React Classes

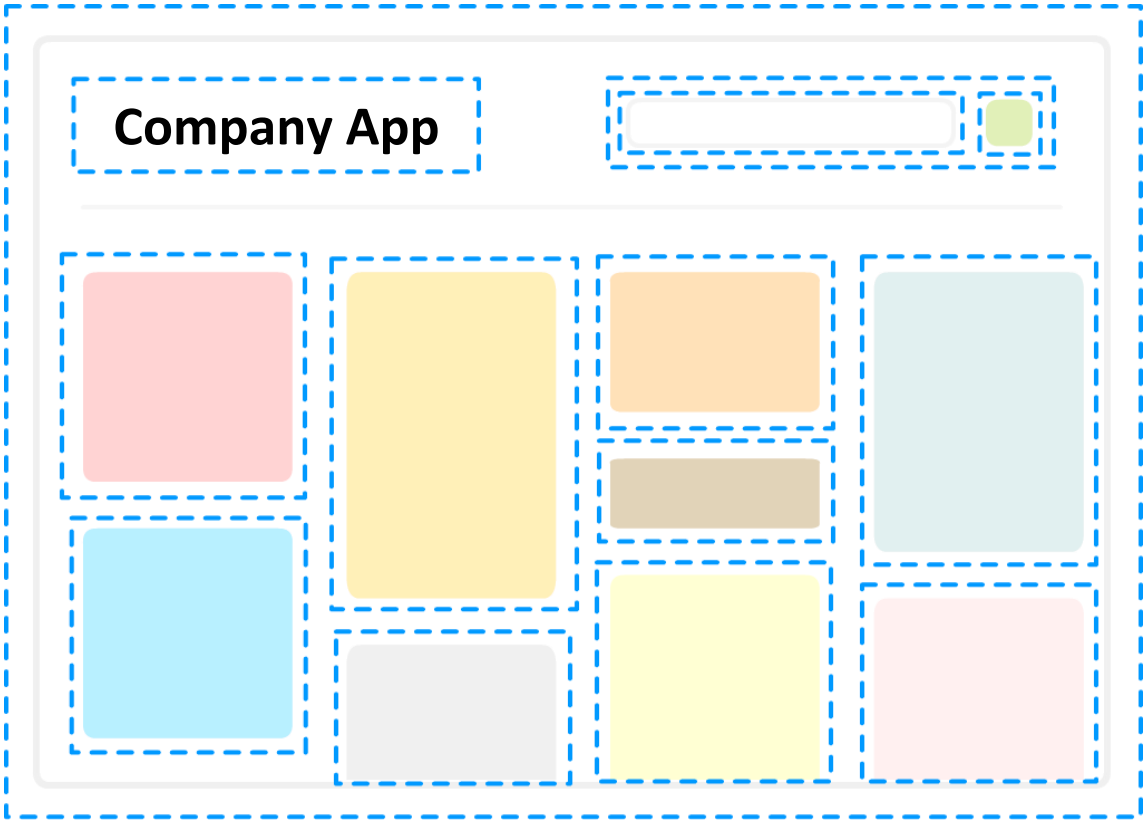
React classes are used to create real components. These classes can then be reused within other components, handle events, and so much more.

React classes are created by extending React.Component, the base class from which all custom classes must be derived. Within the class definition, at the minimum, a render() method is needed. This method is what React calls when it needs to display the component in the UI.

There are other methods with special meaning to React that can be implemented, called the Lifecycle methods. These provide hooks into various stages of the component formation and other events.

A **Component** is one of the core building blocks of React. In other words, we can say that every application we will develop in React will be made up of pieces called components. Components make the task of building UIs much easier.

We can see a UI broken down into multiple individual pieces called components and work on them independently and merge them all in a parent component which will be our final UI.



Each dotted line represents an individual component that is responsible for both what we see and any interactions that it is responsible for. Don’t let this scare we. While this looks really complicated, we will soon see that it will start to make a whole lot of sense once we have had a chance to play with components and some of the awesome things they do or, at least, try really hard to do.

Components in React basically return a piece of JSX code that tells what should be rendered on the screen.

Types of components in ReactJS:

There are two main types of React components:

1. Class components and
2. Functional components

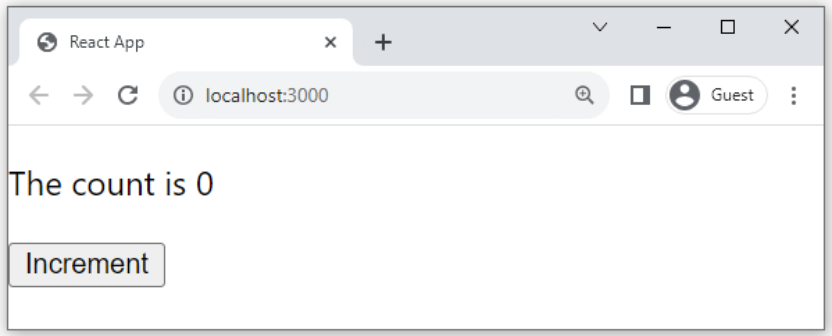
Class components:

Class components are JavaScript classes that extend the React component class. They possess a render method, which returns the HTML to be shown. Class components feature state and lifecycle methods, enabling them to manage their state and react to component changes. Here's how you can create a class component:

```
class ClassComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
  }
  incrementCount() {
    this.setState({
      count: this.state.count + 1,
    });
  }
  render() {
    return (
      <div>
        <p>The count is {this.state.count}</p>
        <button onClick={this.incrementCount}>Increment</button>
      </div>
    );
  }
}
 export default ClassComponent;
```

Add the following code in App.js

```
import ClassComponent from './ClassComponent';
function App() {
  return (  <ClassComponent></ClassComponent>
    );
}
export default App;
```



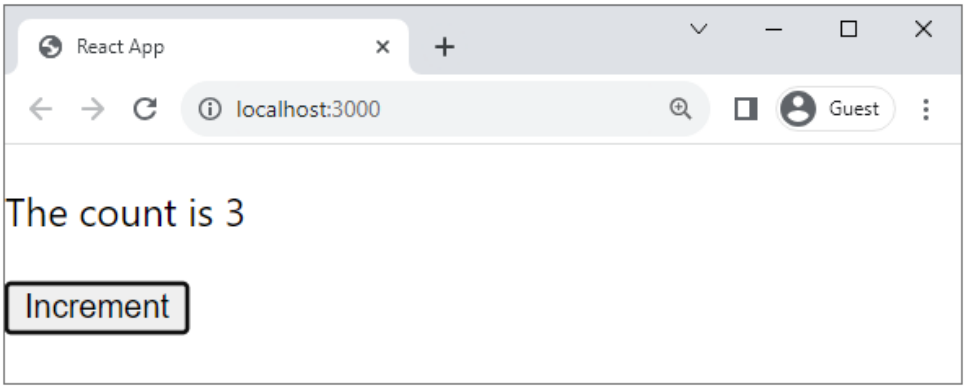
Functional components:

Functional components are simple JavaScript functions. These components accept props as arguments and return the HTML to be displayed. Functional components are less complex than class components and can be easier to reason about. Here's how you can create a functional component:

```
import { useState } from "react";
function FunctionComponent() {
   let [count,setCount] = useState(0)
   return (
     <div>
       <p>The count is {count}</p>
       <button onClick={() => setCount(count + 1)}>Increment</button>
      </div>
    );
  }
export default FunctionComponent;
```

Add the following code in App.js

```
import FunctionComponent from './FunctionComponent';
function App() {
  return (
      //<FirstComponent></FirstComponent>
      <FunctionComponent ></FunctionComponent>
    );
}
export default App;
```



React State

To make components that respond to user input and other events, React uses a data structure called state in the component.

The **state** essentially holds the data, something that can change, as opposed to the immutable properties in the form of props that you saw earlier. This state needs to be used in the render() method to build the view. It is only the change of state that can change the view. When data or the state changes, React automatically rer-enders the view to show the new changed data.

State represents the value of dynamic properties of a React component at a given instance. React provides a dynamic data store for each component. The internal data represents the state of a React component and can be accessed using **this.state** member variable of the component. Whenever the state of the component is changed, the component will re-render itself by calling the render() method along with the new state.

Defining a State

State in React can be used with functional and class components. This initial state of a component must be defined in the constructor of the component's class. Following is the syntax to define a state of any Class,

```
state = {attribute: "value"};
```

```
Class SampleClass extends React.Component
{
   constructor(props)
   {
      super(props);
      this.state = { name : "John Doe" };
   }
}
```

React state management is a process for managing the data that React components need in order to render themselves. This data is typically stored in the component's state object. When the state object changes, the component will re-render itself.

useState:

useState is a basic React hook, which allows a function component to maintain its own state and re-render itself based on the state changes. The signature of the useState is as follows,

```
const [ <state>, <setState> ] = useState( <initialValue> )
```

where,

initialValue − Initial value of the state. state can be specified in any type (number, string, array and object).

state − Variable to represent the value of the state.

setState − Function variable to represent the function to update the state returned by the useState.

The signature of the setState function is as follows −

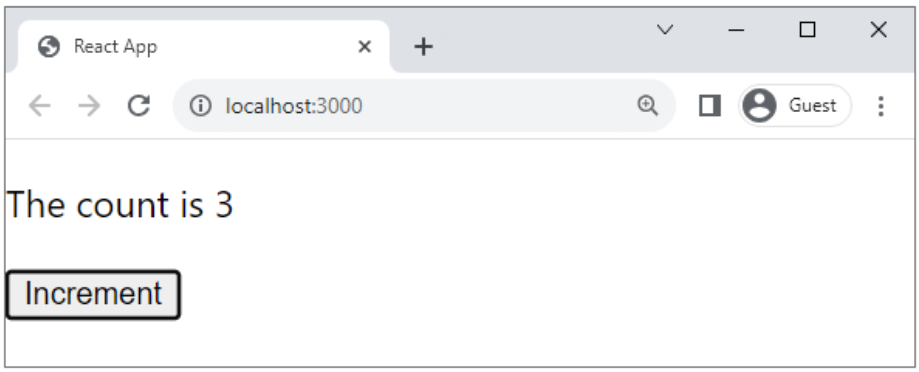
```
setState( <valueToBeUpdated> )
```

Example:

```
import { useState } from "react";
function FunctionComponent() {
   let [count,setCount] = useState(0)
   return (
     <div>
       <p>The count is {count}</p>
       <button onClick={() => setCount(count + 1)}>Increment</button>
      </div>
    );
  }
export default FunctionComponent;
```

Add the following code in App.js

```
import FunctionComponent from './FunctionComponent';
function App() {
  return (
      //<FirstComponent></FirstComponent>
      <FunctionComponent ></FunctionComponent>
    );
}
export default App;
```



Express REST APIs

An API is always needed to create mobile applications, single page applications, use AJAX calls and provide data to clients. An popular architectural style of how to structure and name these APIs and the endpoints is called **REST**(**RE**presentational **S**tate **T**ransfer). HTTP 1.1 was designed keeping REST principles in mind. REST was introduced by Roy Fielding in 2000 in his Paper Fielding Dissertations.

RESTful URIs and methods provide us with almost all information we need to process a request. The table given below summarizes how the various verbs should be used and how URIs should be named. We will be creating a movies API towards the end; let us now discuss how it will be structured.

Method	URI	Details	Function
GET	/students	Safe, cachable	Gets the list of all movies and their details
GET	/students/1234	Safe, cachable	Gets the details of Movie id 1234
POST	/students	N/A	Creates a new movie with the details provided. Response contains the URI for this newly created resource.
PUT	/students/1234	Idempotent	Modifies movie id 1234(creates one if it doesn't already exist). Response contains the URI for this newly created resource.
DELETE	students/1234	Idempotent	Movie id 1234 should be deleted, if it exists. Response should contain the status of the request.
DELETE or PUT	/students	Invalid	Should be invalid. **DELETE** and **PUT** should specify which resource they are working on.

Example: (Student App) => index.js

```
const express = require("express");
const DBConnect = require("./DBConnect")
const cors = require('cors')
const app=express();
const rout=require("./routing")
app.use(express.json())
app.use(express.static("./images"))
app.use(cors())
DBConnect();
app.get("/",rout.getDemo)
app.get("/students",rout.getstudents)
app.get("/login",rout.getlogin)
app.get("/students/:id",rout.getstudent)
app.post("/students",rout.createstudent)
app.put("/students/:id",rout.updatestudent)
app.delete("/students/:id",rout.deletestudent)
app.listen(4000,()=>{
    console.log("Server Started....");
})
```

Routing.js

```
async function getstudents(req,res){
    try {
        const students=await Student.find()
        res.status(201).json({
            students
```

```
        })
    }
    catch (error) {
        res.send("Some Error Occurred")
    }
}

exports.getlogin=(req,res)=>{
    res.send("This is Login page")
}

exports.getstudent= async (req,res)=>
{
    try {
        const student = await Student.findById(req.params.id);
        res.status(201).json({
            student
        })

    } catch (error) {
         res.send(error);
    }
}
exports.createstudent=async (req,res)=>{
    try {
        const student = await Student.create(req.body)
        console.log(fname);
        res.status(201).json({
            msg:"Created",
            data:student
        })
    }
    catch (error) {
        res.send(error)
    }
}
exports.updatestudent=async (req,res)=>{
    try {
```

```
        const updatedstudent = await
Student.findByIdAndUpdate(req.params.id,req.body,{new:true,runValidators:tr
ue})
        res.status(201).json({
            updatedstudent,
            msg:"Updated"
        })
    } catch (error) {
        res.status(400).json({
            status:"Fail",
            msg:error
        })
    }
}

exports.deletestudent = async (req,res)=>{
    try {
        await Student.findByIdAndDelete(req.params.id);
        res.status(201).json({
            status:"Success",
            message:"Deleted"
        })
    } catch (error) {
        res.status(400).json({
            status:"Fail",
            msg:error
        })
    }

}
exports.getDemo = getDemo;
exports.getstudents = getstudents;
```

Output: in postman


```
{
    "students": [
        {
            "_id": "6507cdd24c31d379c2f680af",
            "stud_name": "salini",
            "stud_dept": "ECE",
            "emailid": "salini2828@gmail.com",
            "marks": 90,
```

```
            "__v": 0,
            "simage": "default.png"
        },
        {
            "_id": "65360884f21789ea36080e6b",
            "stud_name": "kavin",
            "stud_dept": "EEE",
            "emailid": "mail2kavin@gmail.com",
            "marks": 88,
            "simage": "kavinstudent.png",
            "__v": 0
        },
        {
            "_id": "65360bf60b9a09d676b858d3",
            "stud_name": "Arul",
            "stud_dept": "ece",
            "emailid": "arul@gmail.com ",
            "marks": 91,
            "simage": "Arulimage.png",
            "__v": 0
        }
    ]
}
```


Routing with React Router

React Router provides routing capabilities to single-page apps built in React. What makes it nice is that it extends what you already know about React in familiar ways to give you all of this routing awesomeness.

Even in a single-page application (SPA), there are in fact multiple logical pages (or views) within the application. It's just that the page load happens only the first time from the server. After that, each of the other views is displayed by manipulating or changing the DOM rather than fetching the entire page from the server.

To navigate between different views of the application, routing is needed. Routing links the state of the page to the URL in the browser. It's not only an easy way to reason about what is displayed in the page based on the URL, it has the following very useful properties:

- The user can use the forward/back buttons of the browser to navigate between visited pages (actually, views) of the application.
- Individual pages can be bookmarked and visited later.
- Links to views can be shared with others. Say you want to ask someone to help you with an issue, and you want to send them the link that displayed the issue. Emailing them the link is far easier and more convenient for the recipient than asking them to navigate through the user interface.

To start, let's install the package that will help us do all this: React Router.

```
$ npm install react-router-dom
```

Example:

Header.js:

```
import { BrowserRouter,Routes, Route, Link } from 'react-router-dom';
import Home from './Home';
import Info from './Info';
import EntryForm from './EntryForm';
function Header(props){
    function insertStudent(students,file){
       props.insertToDB(students,file)
    }
    function updateStudent(student){
        props.updateStudent(student)
    }
    function getStudID(id)
    {
        //console.log(id);
        props.getStudID(id)
    }
    return(
      <BrowserRouter>
      <div>
        <ul>
           <li><Link to="/">Home</Link></li>
           <li><Link to="/students">Student Details</Link></li>
           <li><Link to="/studentsForm">Entry Form</Link></li>
        </ul>
         <Routes>
          <Route exact path='/' element={< Home />}></Route>
          <Route exact path='/students' element={< Info/>}></Route>
          <Route exact path='/studentsForm' element={< EntryForm/>}>
                                                          </Route>
           </Routes>
        </div>
        </BrowserRouter>
    )
}

export default Header;
```

Home.js:

```
function Home(){
```

```
        return( <h2>Welcome!!! User</h2>  )
    }
    export default Home;
```

Info.js:

```
    function Info(){
        return( <h2>Studnet Info Page</h2> )
    }
    export default Info;
```

EntryForm.js:

```
    function EntryForm(){
        return( <h2>This is Entry Form for Studnet</h2> )
    }
    export default EntryForm;
```

Output