UNIVERSITY OF BUEA
FACULTY OF ENGINEERING AND TECHNOLOGY
COMPUTER ENGINEERING

CEF 344: CLIENT-SERVER AND WEB APPLICATION DEVELOPMENT

CLIENT PART (React Js)

In this course, we will see what the react js library is and how to use it to create a web application.

Prerequisites: html, css, basics of javascript, version control(Git).

objective:
At the end of this course, the student should be able to develop a responsive web client using the react js library.

During this course, we are going to follow the react roadmap below to meet our objective.

Road Map of the course

1. Core concepts

a. **Setting up react project** - You can set up a react project by executing some commands, on completing this it gives a React app boilerplate, you can then run your react app to see the initial UI of React. By completing this step you should be familiar with the concept of creating a new react project locally using CLI(Command-line Interface).

b. **Execution flow of React** - It is important for a React developer to understand the code flow of a react app. Also understanding the project structure. By completing this step you should be familiar with React app folder structure, Also how react app runs in the browser, what is virtual DOM and how DOM is manipulated in React.

c. **Using JSX** - JSX looks like HTML, It is based on XML, JSX transformed into HTML tags during runtime. By learning the JSX concept you should be having a good understanding of JSX, Similarities between HTML and JSX and Advantages of JSX over HTML element tags.

d. **Component types** - It is important to have knowledge of Components, their types and component life-cycle. By learning the Component concept you should have a good understanding of different component types and which to use when.

e. **Handling states/useState hook** - State is an object within which we can store, read and update data in a component. In the functional component we can use the 'useState' hook to manage state locally. By learning the React state you should have a good understanding of how a state works, using the useState hook, how to show state value in JSX and how to update the state.

f. **Handling functions** - Function in react is same as javascript functions, we can create our own functions to perform specific tasks. By learning about functions you should be able to create functions in the react component and call the function.

g. **Handling JSX events** - JSX events allows us to handle events which are react's element(JSX) specific. By learning about JSX events you should have a good understanding of different JSX events(such as onClick, onChange, etc), and their use.

h. **Modules** - Modules lets you write sharable code so that you can reuse it by importing. By learning about modules you should be able to understand the concept of modules such as importing and exporting modules, public and private properties and function.

i. **Components nesting and reusability** - Components are an independent and reusable block of code which returns JSX and can also perform some specific tasks. By learning component reusability you should be able to understand component nesting, also root, parent and child component, and component tree. you should also be having a good knowledge of creating components in such a way that they can be used in multiple places.

j. **Props** - Using Props we can pass data from one component to another. By learning Props you must be familiar with passing the props from the parent component to the child component and then receiving and using them in the

child component.

k. **Conditional rendering** - Conditional rendering lets you render JSX conditionally just like we use if else condition in javascript. By learning about conditional rendering you should be able to display UI conditionally, and understand the ternary operator used in React JSX.

2. Basics

a. **Debugging and logging** - React dev tools extension helps you monitor react state and components within the browsers window. By learning this step you should be able to debug React app.

b. **Fetching & displaying data from external API** - This lets you perform REST API methods using the built-in method of javascript. By learning this step you should be having a good understanding of javascript's built-in fetch function to fetch the data from an endpoint, storing the data and displaying accordingly on the UI.

c. **Understanding and using Axios package** - Axios is one of the popular library for making HTTP requests to external API endpoints effectively. By learning axios you should be familiar with fetching async data from an endpoint using axios and should also be familiar with other REST API methods using axios.

d. **UseEffect hook** - UseEffect hook lets you decide what to perform after rendering a component, Also affecting the component based on the dependency provided to it. By learning the UseEffect hook you should be familiar with the side effect of the component rendering with dependency.

e. **Context API/useContext hook** - Context API is React's built-in functionality to perform state management without using any external library. By learning about Context API you should be having a good understanding of global state and state management, consumer and provider, useContext and UserReducer hook.

f. **Browser's local storage** - Local storage lets you store the data locally within the browser's storage. By learning this step you should have a good understanding of browser local storage, reading and writing to and from local storage.

g. **React router** - React router DOM is one of the popular external libraries for navigating your react app, It also lets your UI in sync with the URL. By learning about React router DOM library you should be familiar with navigating and routing your react application and switching the UI as per the route.

3. Styling UI

a. **Grid layout** - CSS Grid Layout concept offers a grid-based layout system for designing the UI on the basis of rows and columns. By learning Grid layout you should be having a good understanding of placing UI elements as a cell in the grid layout.

b. **Flexbox** - Flexbox Layout lets you build flexible and responsive UI without using CSS's positioning and float. By learning flexbox you should be familiar with concept of container, and aligning and justifying elements.

1. Core concepts

a. Setting up React

React is a popular JavaScript framework for creating front-end applications. Originally created by Facebook, it has gained popularity by allowing developers to create fast applications using an intuitive programming paradigm that ties JavaScript with an HTML-like syntax known as JSX.

Starting a new React project used to be a complicated multi-step process that involved setting up a build system, a code transpiler to convert modern syntax to code that is readable by all browsers, and a base directory structure. But now, Create React App includes all the JavaScript packages you need to run a React project, including code transpiling, basic linting, testing, and build systems. It also includes a server with *hot reloading* that will refresh your page as you make code changes. Finally, it will create a structure for your directories and components so you can jump in and start coding in just a few minutes.

Prerequisites

First of all, you are going to need NPM (or Yarn, alternatively). Let's use NPM for this example.

If you don't have it installed on your system, then you need to head to the official Node.js website to download and install Node, which also includes NPM (Node Package Manager).

What is create-react-app?

Since it is complicated and takes a lot of time, we don't want to configure React manually. create-react-app is a much easier way which does all the configuration and necessary package installations for us automatically and starts a new React app locally, ready for development.
Another advantage of using create-react-app is that you don't have to deal with Babel or Webpack configurations. All of the necessary configurations will be made by create-react-app for you.

How to Install Create-React-App

In order to install your app, first go to your workspace (desktop or a folder) and run the following command:

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

Note: If you're on Mac and receiving permission errors, don't forget to be a super user first with the sudo command.

How to Run the App You Created with Create-React-App

After the installation is completed, change to the directory where your app was installed:

```
cd my-app
```

and finally run npm start to see your app live on localhost:

```
npm start
```

If you see something like this in your browser, you are ready to work with React. Congratulations! :)

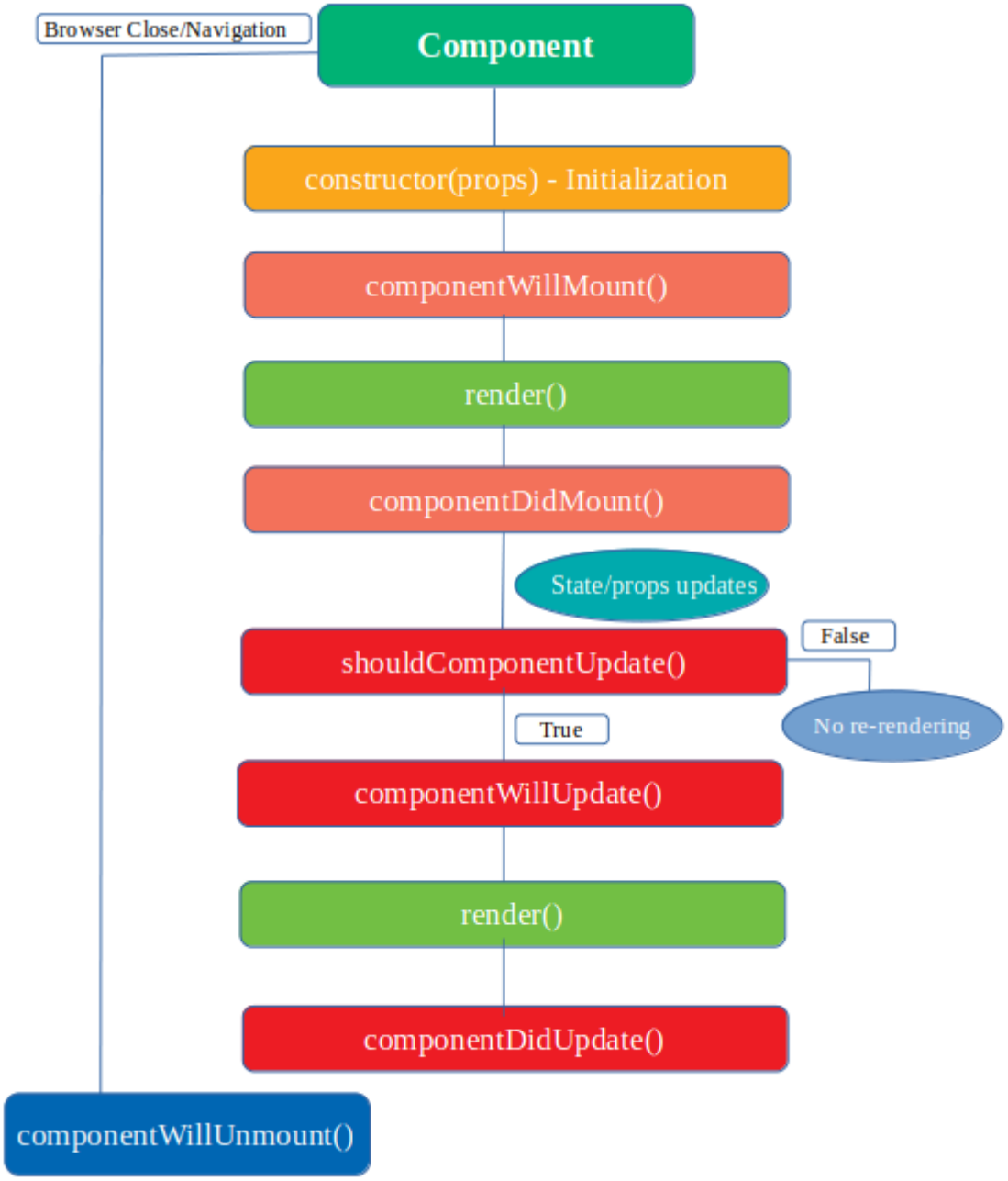
b. Execution flow of react

React web apps are actually a collection of independent components that run according to the interactions made with them. Every React Component has a lifecycle of its own, lifecycle of a component can be defined as the series of methods that are invoked in different stages of the component's existence. The definition is pretty straightforward but what do we mean by different stages? A React Component can go through four stages of its life as follows.

- Initialization: This is the stage where the component is constructed with the given Props and default state. This is done in the constructor of a Component Class.
- Mounting: Mounting is the stage of rendering the JSX returned by the render method itself.
- Updating: Updating is the stage when the state of a component is updated and the application is repainted.

- Unmounting: As the name suggests Unmounting is the final step of the component lifecycle where the component is removed from the page.

React provides the developers a set of predefined functions that if present is invoked around specific events in the lifetime of the component. Developers are supposed to override the functions with desired logic to execute accordingly. We have illustrated the gist in the following diagram.



source : freecodecamp

1. **Initialization**: In this phase, the developer has to define the props and initial

state of the component this is generally done in the constructor of the component. The following code snippet describes the initialization process.

```
class Calendar extends React.Component {
        constructor(props)
        {
                // Calling the constructor of
                // Parent Class React.Component
                super(props);

                // Setting the initial state
                this.state = { date : new Date() };
        }
}
```

2. **Mounting**: Mounting is the phase of the component lifecycle when the initialization of the component is completed and the component is mounted on the DOM and rendered for the first time on the webpage. Now React follows a default procedure in the Naming Conventions of these predefined functions where the functions containing "Will" represents before some specific phase and "Did" represents after the completion of that phase. The mounting phase consists of two such predefined functions as described below.

- **componentWillMount() Function**: As the name clearly suggests, this function is invoked right before the component is mounted on the DOM i.e. this function gets invoked once before the render() function is executed for the first time.
- **componentDidMount() Function**: Similarly as the previous one this function is invoked right after the component is mounted on the DOM i.e. this function gets invoked once after the render() function is executed for the first time

3. **Updation**: Updation is the phase where the states and props of a component are updated followed by some user events such as clicking(here

image the even where user clicks on the **add to card** button), pressing a key on the keyboard(filling info in a form filed), etc. The following are the descriptions of functions that are invoked at different points of Updation phase.

- **componentWillReceiveProps() Function**: This is a Props exclusive Function and is independent of States. This function is invoked before a mounted component gets its props reassigned. The function is passed the new set of Props which may or may not be identical to the original Props. Thus checking is a mandatory step in this regard. The following code snippet shows a sample use-case.

```
componentWillReceiveProps(newProps)
{
        if (this.props !== newProps) {
                console.log(" New Props have been assigned ");
                // Use this.setState() to rerender the page.
        }
}
```

- **setState() Function**: This is not particularly a Lifecycle function and can be invoked explicitly at any instant. This function is used to update the state of a component. You may refer to this article for detailed information.
- **shouldComponentUpdate() Function**: By default, every state or props update re-render the page but this may not always be the desired outcome, sometimes it is desired that updating the page will not be repainted. The shouldComponentUpdate() Function fulfills the requirement by letting React know whether the component's output will be affected by the update or not. shouldComponentUpdate() is invoked before rendering an already mounted

component when new props or state are being received. If returned false then the subsequent steps of rendering will not be carried out. This function can't be used in the case of forceUpdate(). The Function takes the new Props and new State as the arguments and returns whether to re-render or not.

- **componentWillUpdate() Function**: As the name clearly suggests, this function is invoked before the component is rerendered i.e. this function gets invoked once before the render() function is executed after the updation of State or Props.
- **componentDidUpdate() Function**: Similarly this function is invoked after the component is rerendered i.e. this function gets invoked once after the render() function is executed after the updation of State or Props.

4**. Unmounting**: This is the final phase of the lifecycle of the component that is the phase of unmounting the component from the DOM. The following function is the sole member of this phase.

- **componentWillUnmount() Function**: This function is invoked before the component is finally unmounted from the DOM i.e. this function gets invoked once before the component is removed from the page and this denotes the end of the lifecycle.

c. Introduction to JSX

Consider this variable declaration:

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

This funny tag syntax is neither a string nor HTML.

It is called JSX, and it is a syntax extension to JavaScript. We recommend using it with React to describe what the UI should look like. JSX may remind you of a template language, but it comes with the full power of JavaScript.

Why JSX?

React embraces the fact that rendering logic is inherently coupled with other UI logic: how events are handled, how the state changes over time, and how the data is prepared for display.

Instead of artificially separating *technologies* by putting markup and logic in separate files, React separates *concerns* with loosely coupled units called "components" that contain both. We will come back to components in a further section, but if you're not yet comfortable putting markup in JS, this talk might convince you otherwise.

React doesn't require using JSX, but most people find it helpful as a visual aid when working with UI inside the JavaScript code. It also allows React to show more useful error and warning messages.

With that out of the way, let's get started!

Embedding Expressions in JSX

In the example below, we declare a variable called name and then use it inside JSX by wrapping it in curly braces:

```
const name = 'Josh Perez';

const element = <h1>Hello, {name}</h1>;
```

You can put any valid JavaScript expression inside the curly braces in JSX. For example, 2 + 2, user.firstName, or formatName(user) are all valid JavaScript expressions.

In the example below, we embed the result of calling a JavaScript function, formatName(user), into an <h1> element.

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};

const element = (
  <h1>

    Hello, {formatName(user)}!
  </h1>
);
```

JSX is an Expression Too

After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects.

This means that you can use JSX inside of if statements and for loops, assign it to variables, accept it as arguments, and return it from functions:

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

Specifying Attributes with JSX

You may use quotes to specify string literals as attributes:

```
const element = <a href="https://www.reactjs.org"> link </a>;
```

You may also use curly braces to embed a JavaScript expression in an attribute:

```
const element = <img src={user.avatarUrl}></img>;
```

Don't put quotes around curly braces when embedding a JavaScript expression in an attribute. You should either use quotes (for string values) or curly braces (for expressions), but not both in the same attribute.

Specifying Children with JSX

If a tag is empty, you may close it immediately with />, like XML:

```
const element = <img src={user.avatarUrl} />;
```

JSX tags may contain children:

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

JSX Represents Objects

Babel compiles JSX down to React.createElement() calls.
These two examples are identical:

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

React.createElement() performs a few checks to help you write bug-free code but essentially it creates an object like this:

```
// Note: this structure is simplified
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world!'
  }
};
```

d. Component types

What are React Components?

React components are independent and reusable code. They are the building blocks of any React application. Components serve the same purpose as JavaScript functions, but work individually to return JSX code as elements for our UI. Components usually come in two types, functional components and

class components, but today we will also be talking about pure components and higher-order components.

Functional Components

Functional components are functions that takes in props and return JSX. They do not natively have state or lifecycle methods, but this functionality can be added by implementing React Hooks. Functional components are usually used to display information. They are easy to read, debug, and test.

```
// Functional Component Example
import React from 'react';
const HelloWorld = () => {
  return (
    <div>
      <p>Hello World!</p>
    </div>
  )
}
export default HelloWorld;
```

In the code above, it is a very simple component that consists of a constant variable `HelloWorld` that is assigned to an arrow function which returns JSX. Functional components do not need to be arrow functions. They can be declared with regular JavaScript functions. You can also pass in `props` to the function and use them to render data in the JSX code.

Class Components

Class components have previously been the most commonly used among the four component types. This is because class components are able to do everything a functional component do but more. It can utilize the main functions of React, ***state***, ***props***, and ***lifecycle methods***. Unlike functional components, class components are consist of … well, a class.

```
// Class Component Example
import React from 'react';
class HelloWorld extends React.Component {
  render() {
     return (
        <div>
          <p>Hello World!</p>
        </div>
     )
  }
}
export default HelloWorld;
```

Class component syntax differs from functional component syntax. Class components extend `React.Component` after declaring the class `HelloWorld` and require a `render()` method to return JSX code. In this class component, you can declare a `state`, set it to a JavaScript object, and use `props` to be the initial state or change the state in ***lifecycle methods***. Some lifecycle methods are `componentDidMount()`, `componentDidUpdate()`, and `componentWillUnmount()`. These are actions that a functional component cannot perform unless they use React Hooks.

e. Handling state/useState hook