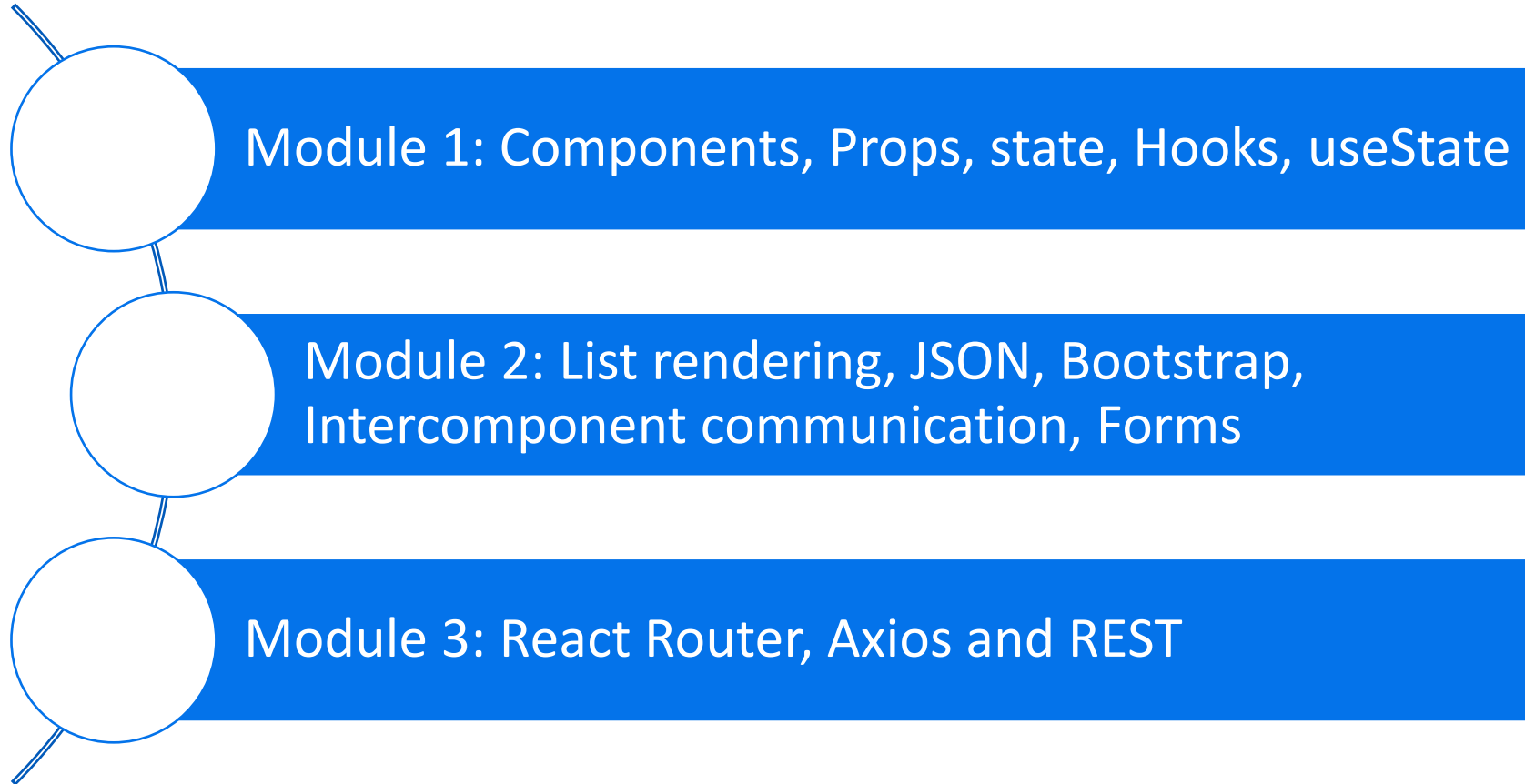


React



Agenda



S/W Requirements

IDE : Visual Studio

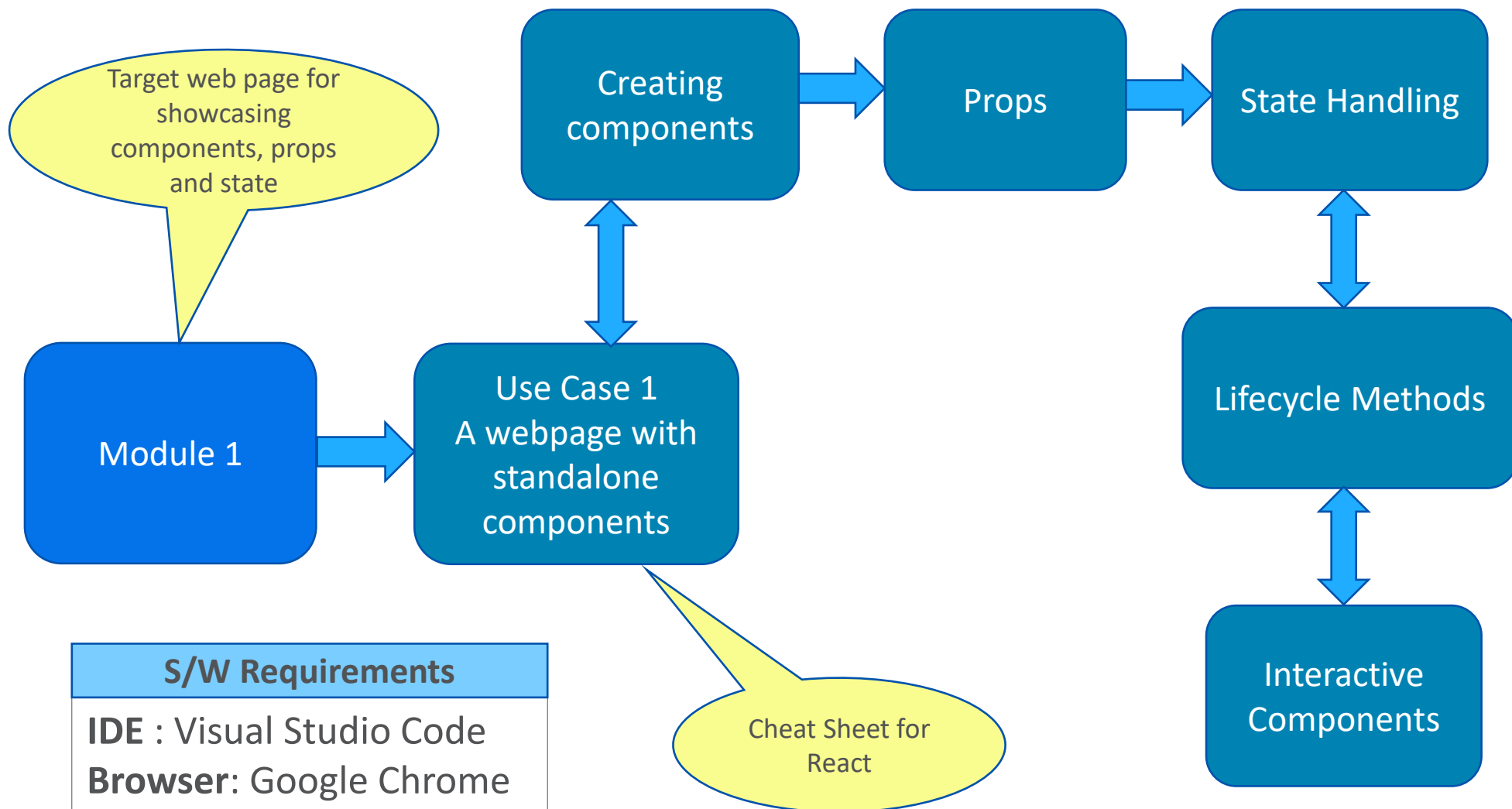
Browser: Google Chrome

Module 1

Components, Props, state, Hooks, useState



Module 1 Design



Module 1: Overview

- As part of this module, we will cover the most important and basic concepts of react. These concepts form the building block of any react application
- The basic concepts that would be covered in this module are
 - Components
 - JSX syntax
 - Props and prop handling
 - Introduction to Hooks and useState Hook
 - State and updating state
 - Basic component hierarchy
 - Best practices for state updates
 - Interactive components (Event Handling)

Introducing Use Case 1



Presenting Use Case 1

UseCase 1 - Components, Props and States

Hello World Component

Single Prop

Message from Euler : Hi, Hello

Multiple Props

Message from Ramanujam : I got this in my dreams

State and Virtual DOM

Time Elapsed : 28

Interactive Component - Event Handling

Click me Please

Components in our use case

UseCase 1 - Components, Props and States

Hello World Component

HelloWorld Component

Single Prop

Message from Euler : Hi, Hello

HelloMessage (prop1)

Multiple Props

Message from Ramanujam : I got this in my dreams

HelloMessage (prop1, prop2)

State and Virtual DOM

Time Elapsed : 28

Counter (State Changes)

Interactive Component - Event Handling

Click me Please

Alert (event handling)

 Sent props

 Prop not sent – default value

Use case 1 - Explanation

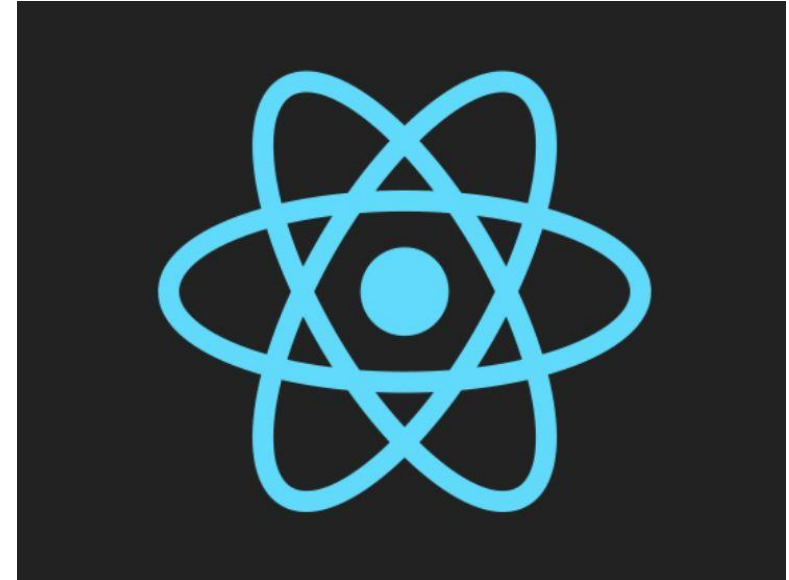
- As part of this use case, Create a simple react app with a single page which contains the below components
- Give this page a title with h3 enabled text (UseCase 1 - Components, Props and States)
- Also ensure that the entire page has a left and a right margin of 100px
- Create all the below mentioned components under **src/components/** subfolder
- Create a component named **HelloWorld** displaying a simple text as “Hello World Component”
- Create a component named **HelloMessage** which can receive 2 properties (name and message). If message prop is missing, use “**Hi, Hello**” as the default message.
- Create a component named **Counter** which displays the time elapsed in seconds. Implement this by using react state concept. Also, use the browser’s event **setInterval** for the implementation.
- Create a component named **Alert** which shows up a button and when the button is clicked, should show an alert dialog with the message “React is a great UI library”
- Display all these above components one by one in the order as shown in the use case 1

What is React?



What is React

- React is a JavaScript library, created and open sourced by Facebook
- React is a declarative, efficient, and flexible JavaScript library for building user interfaces
- React lets you compose complex UIs from small and isolated pieces of code called “components”
- React can be used to make Rich internet applications, with more complex UI logic and user interactions. E.g. Facebook, Netflix
- React can be used to built all modern applications right from financial applications to consumer apps



Installation and Creating a React app

- Install one of the latest Node.js version (LTS release) v8.0 would be optimal
- Check if node.js is properly installed using the below command
 - ***node -v***
- Refer the below official getting started page
 - <https://reactjs.org/docs/create-a-new-react-app.html>
- Install the create-react-app tool using the below npm command
 - ***npm install -g create-react-app***
- Create a new react app using the below command
 - ***create-react-app app-name***
- Change to the app folder and run the below command to start the react application
 - ***npm start***
- Once the app is run, it should open up the browser and show up the default react application

First app's output

- The first react application which you created using create-react-app should show the default output as shown in the below image. This output ensures that everything went well with your installation and build process in your local development machine.

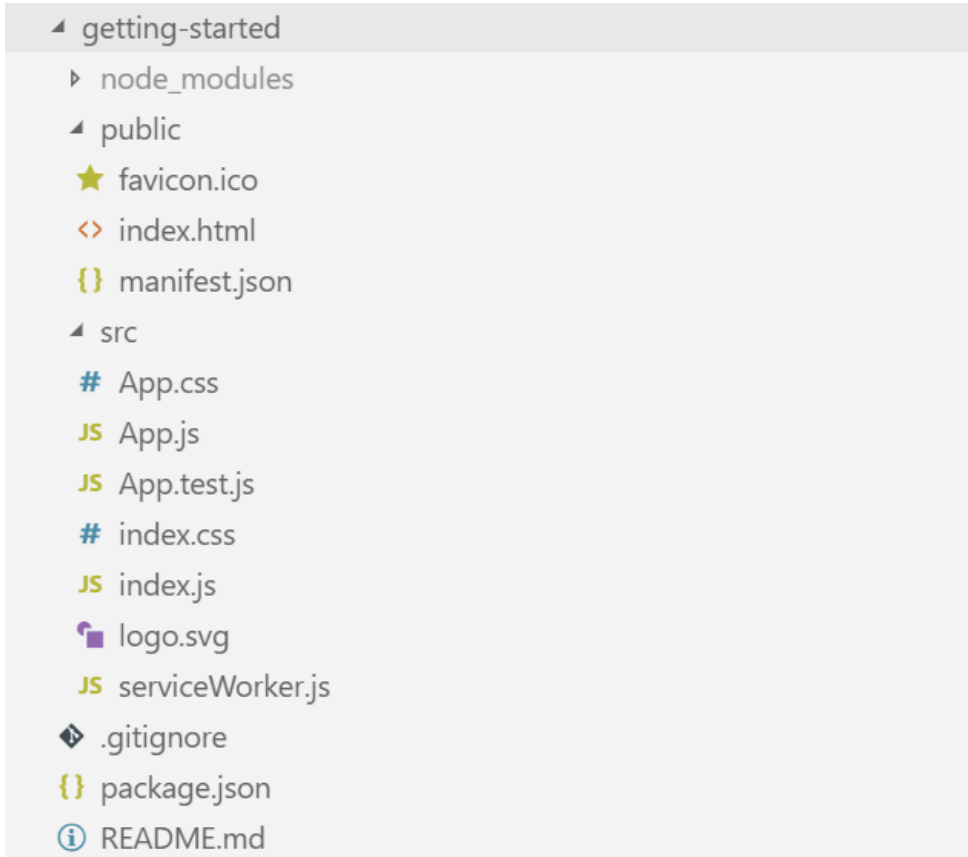


Edit `src/App.js` and save to reload.

[Learn React](#)

React app – Folder Structure

- **Folder Structure** – The default app created by create-react-app contains the below folder structure and the set of files



```
└─ getting-started
   └─ node_modules
   └─ public
      └─ favicon.ico
      └─ index.html
      └─ manifest.json
   └─ src
      └─ App.css
      └─ App.js
      └─ App.test.js
      └─ index.css
      └─ index.js
      └─ logo.svg
      └─ serviceWorker.js
   └─ .gitignore
   └─ package.json
   └─ README.md
```

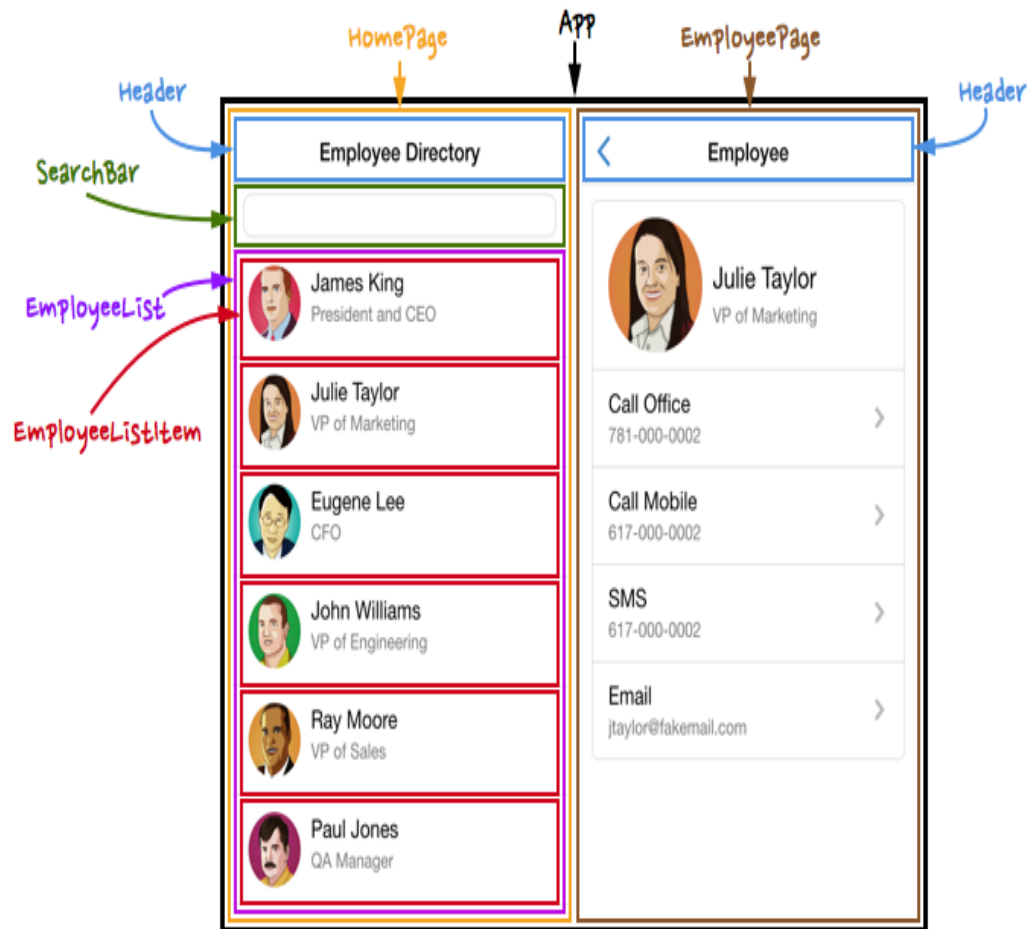
About the app files

- For the project to build, these files must exist with exact filenames
 - ***public/index.html*** - the page template
 - ***src/index.js*** - the JavaScript entry point
- Regarding the other major files/folders
 - ***node_modules/*** - *the folder where all the required packages are installed*
 - ***package.json*** - *the package manager file, where all dependencies are mentioned*
 - ***App.js*** - The root component of our app
 - ***App.css*** - css specific to the app component
- All the new application code that the developer has to add for writing business logic/components are typically put under the **src/** folder
- Only files under ***public*** can be used from ***public/index.html***

Why create-react-app

- **create-react-app** is an officially supported way to create single-page React applications. It offers a modern build setup with no configuration
- React apps are mostly written with ES6. Babel is used for ES6 to ES5 conversion
- **create-react-app** has the following advantages
 - Takes care of babel conversion from ES6 to ES5
 - Scaling to many files and components
 - Using third-party libraries from npm
 - Detecting common mistakes early
 - Live-editing CSS and JS in development
 - Optimizing the output for production

What are Components?



What are Components?

- Components let you split the UI into independent, reusable pieces, and think about each piece in isolation
- Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen
- A component contains both the UI and the data logic associated with the UI
- Components can be created using es6 functions.
- React components are written using the JSX syntax (more specific to React only).

What is JSX and Why?

- JSX is a syntax extension to JavaScript
- JSX is a template language, but it comes with the full power of JavaScript
- Example of a simple JSX code

```
// JSX syntax (mix of javascript and html)
const element = <h1>Hello, world!</h1>;
```

- **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
- Embedding expressions in JSX

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

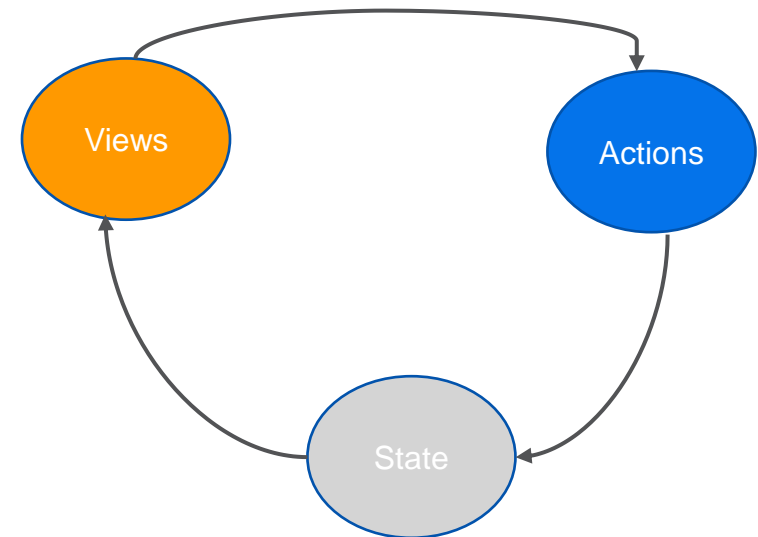
- You can put any valid JavaScript expression inside the curly braces in JSX

One way data binding

- Data binding is a common terminology in UI development, which determines how data gets bound to the UI
- React uses 1-way data binding, where the data flows in one direction (Unidirectional data flow)

- **How 1-way data binding works?**

- Whenever the state changes, the view is updated
- View can trigger actions
- Actions could update the state
- Any state changes will update the view again
- As depicted, the single source of truth is the state



Creating Components

- A component can be created using a simple JavaScript function also

```
function HelloWorld() {  
  return <div>Hello World Component</div>;  
}
```

Component Props

- **What are Props?**

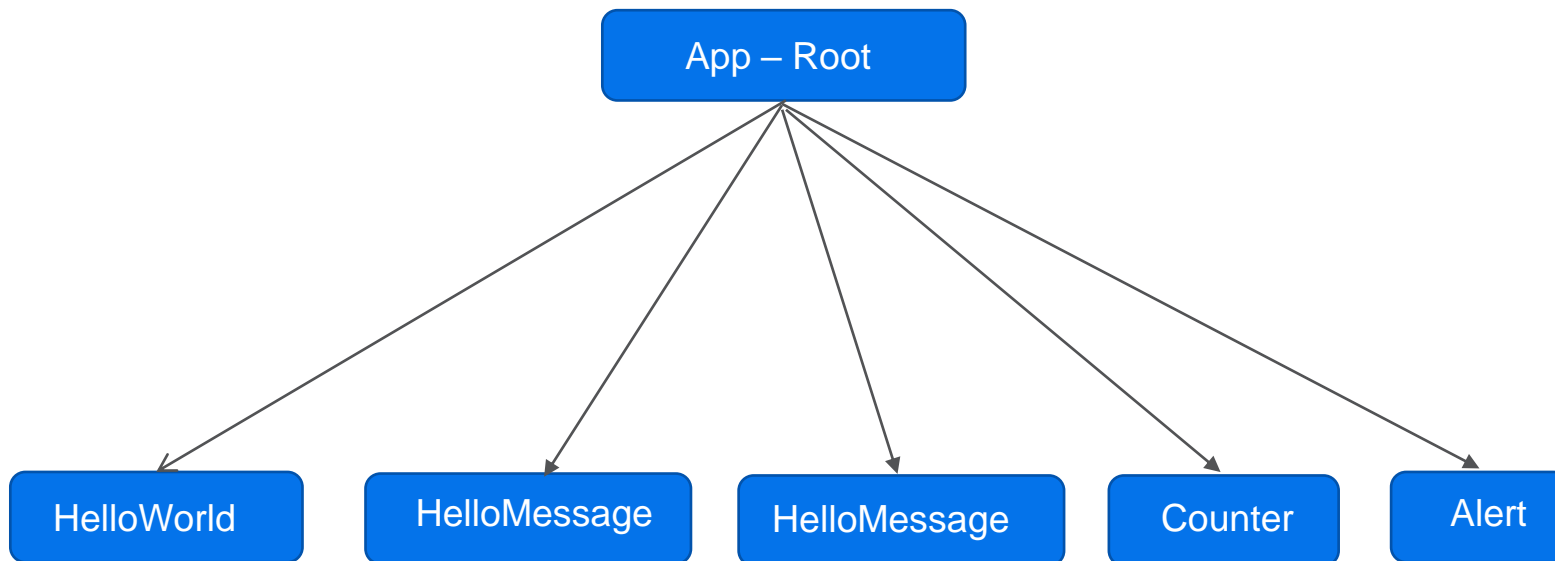
- Props represents a single object containing all the attributes/properties sent to the component
- Props can be compared to something like a collection of all parameters passed to a function

- **Why Props?**

- Props can be used for inter component message passing
- Components can be logically organized into a parent child relationship
- The parent component can pass props to the child component

Component Hierarchy Example

- The below picture shows the simplest component hierarchy that can be a possible in a simple react app
- App component is the root component and it has 5 children components



Using Props in a component

- Props can be passed to a component just like any html attributes
- Props could be strings, numbers, Boolean or objects (e.g employee, product etc)
- A simple component HelloMessage which can receive two props (name and message)

```
export default function HelloMessage(props) {  
  return (  
    <div>  
      message from {props.name} : {props.message}  
    </div>  
  );  
}
```

- The HelloMessage component is receiving two props(name and message)
- If props are missing, no values displayed.

Default Props

- Default values of props can be declared as de-structuring objects as shown in the below code

```
export default function HelloMessage(props){  
  const{  
    name,  
    message='good luck'  
  }=props  
  
  return(  
    <div>  
      message from {name}:{message}  
    </div>  
  )  
}
```

Passing props to a component

- The HelloMessage component can be used and rendered in other components as shown below

```
<h6>Single Props</h6>
<HelloMessage name="jack" />
<br/>
<h6>Multiple Props</h6>
<HelloMessage name='marry' message="I got This in my dreams" />
```

- We have used HelloMessage component in 2 different ways as shown in the above diagram
- In the first usage, we are only sending the name prop. Hence the default message 'good luck' will be used and rendered.

Output

Single Props

message from jack : good luck

- But in the 2nd usage, we are sending both the props, name and message

Output

Multiple Props

message from marry : I got This in my dreams

Component State

What is a component state?

- The heart of every React component is its “state”
- State is an object that determines how that component renders & behaves
- In other words, “state” is what allows you to create components that are dynamic and interactive
- State is a JavaScript object that stores a component’s dynamic data and determines the component’s behavior

Unique features of state

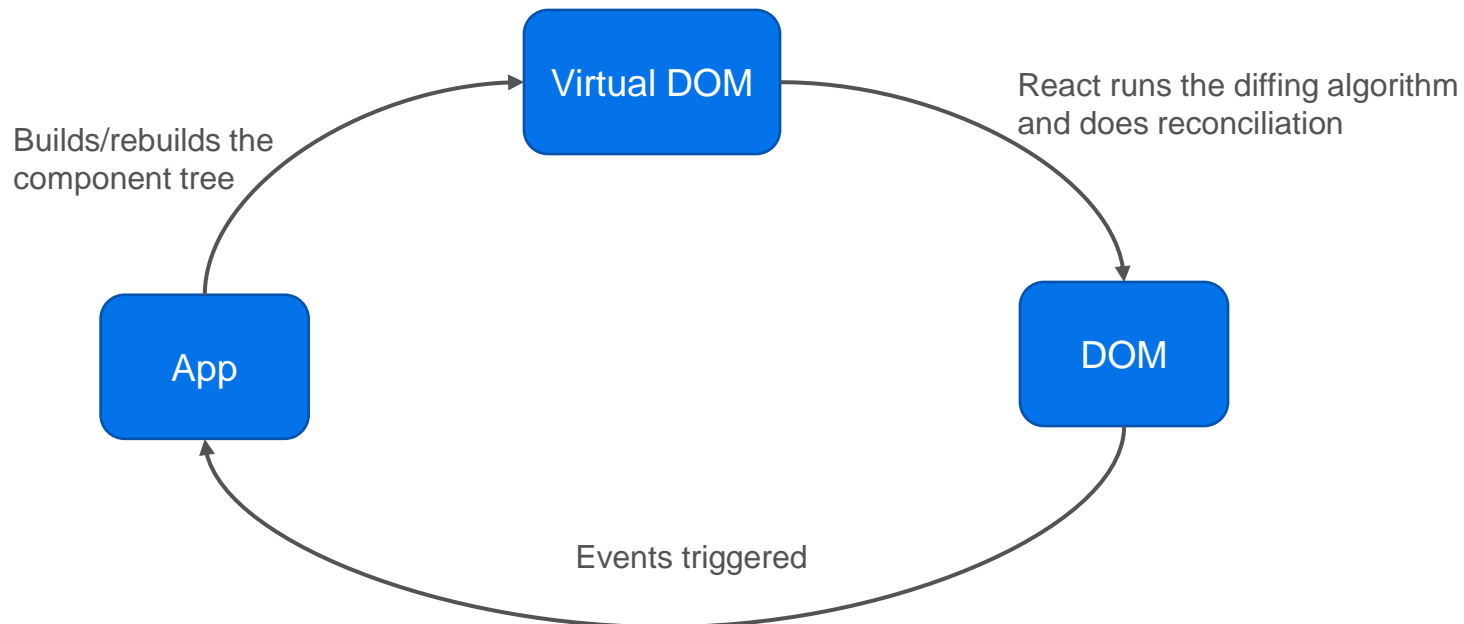
- State can be managed inside functional component
- Unlike props, state is private to a component
- In React, state is immutable
- Immutability dictates that state should never be changed directly but changes should be made to a copy of the current state
- State changes can be through a special react method called **useState()** in functional component.

State Usage and State Changes

- We can refer to state all throughout the component code using **useState()**. We will be focusing on useState hook in our projects.
- To modify the state, we have to use the **useState assigned method in the declaration**.
- Whenever we want to change the UI, we update the state with corresponding data, then react will automatically update the UI based on the state changes
- This process is called re-rendering and react uses the concept of a virtual DOM to make state changes/transition fast and efficient
- Hence the UI changes are pretty responsive and fast compared to the standard DOM manipulation

React virtual DOM

- The **virtual DOM (VDOM)** is a programming concept where an ideal, or “virtual”, representation of a UI is kept in memory and synced with the “real” DOM by a library such as ReactDOM. This process is called **reconciliation**
- DOM manipulation is a very costly operation and React minimizes the DOM manipulation by using the virtual DOM
- Whenever state changes occur, React makes a diff between the actual DOM and the virtual DOM and updates only what is needed. Hence React based UIs are pretty-fast in the benchmarks



React Hooks

- Hooks are a new addition in React 16.8.
- Hooks are functions that help in using state and Lifecycle features without writing class.
- If we want to add state for a functional component, we can use React hooks.
- React provides a few inbuilt hooks like “useState”, “useEffect”, etc .
- Custom hooks can also be created to reuse component logic.

useState Hook

- useState is a special function that declares a state variable.
- It takes the initial state as input and returns a pair of values:
 - the current state
 - a function that updates it

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

export default function Counter() {

  const [counterValue, setValue] = useState(0)
```

- We declare a state variable called **counterValue** and set it to 0.
- React will remember its current value between re-renders and provide the most recent one to our function. If we want to update the current count, we need to use **setValue()** method.

useState() correct usage

- There are a couple of things one should know about **useState()**
- Syntax:

Array with two elements

```
const [counterValue, setValue] = useState(initialValue)
```

value :
React assigns
the *current*
value to the first
element

function :
React assigns the
update function
to the second element

Initial Value :
Pass the initial
value to the
useState Hook. It
can accept *string*,
number, *array*,
object

useState() usage

- State updates may be asynchronous
- Multiple state variables can be assigned inside a functional component

```
const [counterValue, setValue] = useState(0)

const [users, setUsers] = useState([])
```

- useState() only works in functional component

Functional Component using useState() - Counter

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

export default function Counter() {

  const [counterValue, setValue] = useState(0)

  const handleIncrement = () => {
    setValue(value + 1)
  }
  const handleDecrement = () => {
    setValue(value - 1)
  }
  return (
    <div>
      <h3>Counter</h3>

      <div>
        <button onClick={handleIncrement}>+</button> &nbsp;
        <button onClick={handleDecrement}>-</button> &nbsp;
      </div>

      <div>
        <h4>Value : {counterValue}</h4>
      </div>
    </div>
  )
}
```

Setting Initial State

Changing State with state method

Using value with {} syntax

React component lifecycle methods

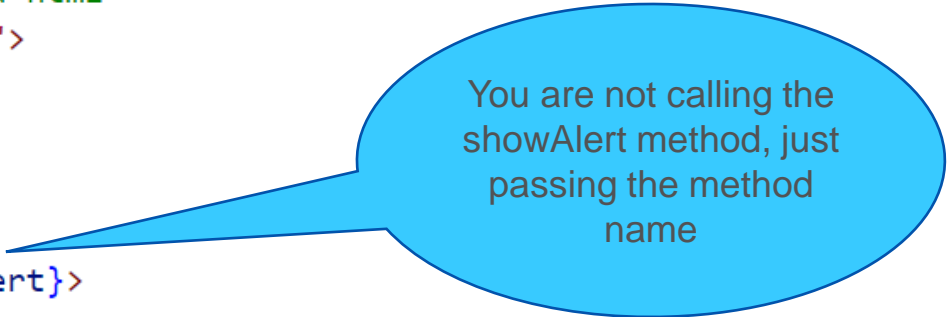
- Every react component has a lifecycle associated with it. The component goes from a creation state to a destruction state and everything in between
- To put it in other words, during the component lifecycle, react components gets mounted to the DOM, then goes through an updating phase, and finally gets unmounted from the DOM
- During this lifecycle, various events could happen and react provides certain methods/hooks to hook into the lifecycle event and add some custom behaviour. These are called **lifecycle methods/hooks**
- After React 16.8 release, Hooks have eliminated lot of boilerplate code which was required in Class Components and made the lifecycle management less complicated for developers
- Hooks are used in functional components

Interactive components – Event handling

- To make components more interactive, with user input, we need to handle events in components
- Handling events with React elements is very similar to handling events on DOM elements. But there are some syntactic differences
- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string. In functional component, **this** keyword is not required.

```
// event handling in standard html
<button onclick="showAlert()">
  Click me please
</button>
```

```
// event handling in react
<button onClick={this.showAlert}>
  Click me please
</button>
```



You are not calling the showAlert method, just passing the method name

An interactive React component - Alert

```
export default function Alert () {  
  
  const showAlert = () => {  
    alert("React is a great UI library")  
  }  
  
  return (  
    <div>  
      <button onClick={showAlert}>  
        Show Alert  
      </button>  
    </div>  
  )  
}
```

Event handler, as a class function

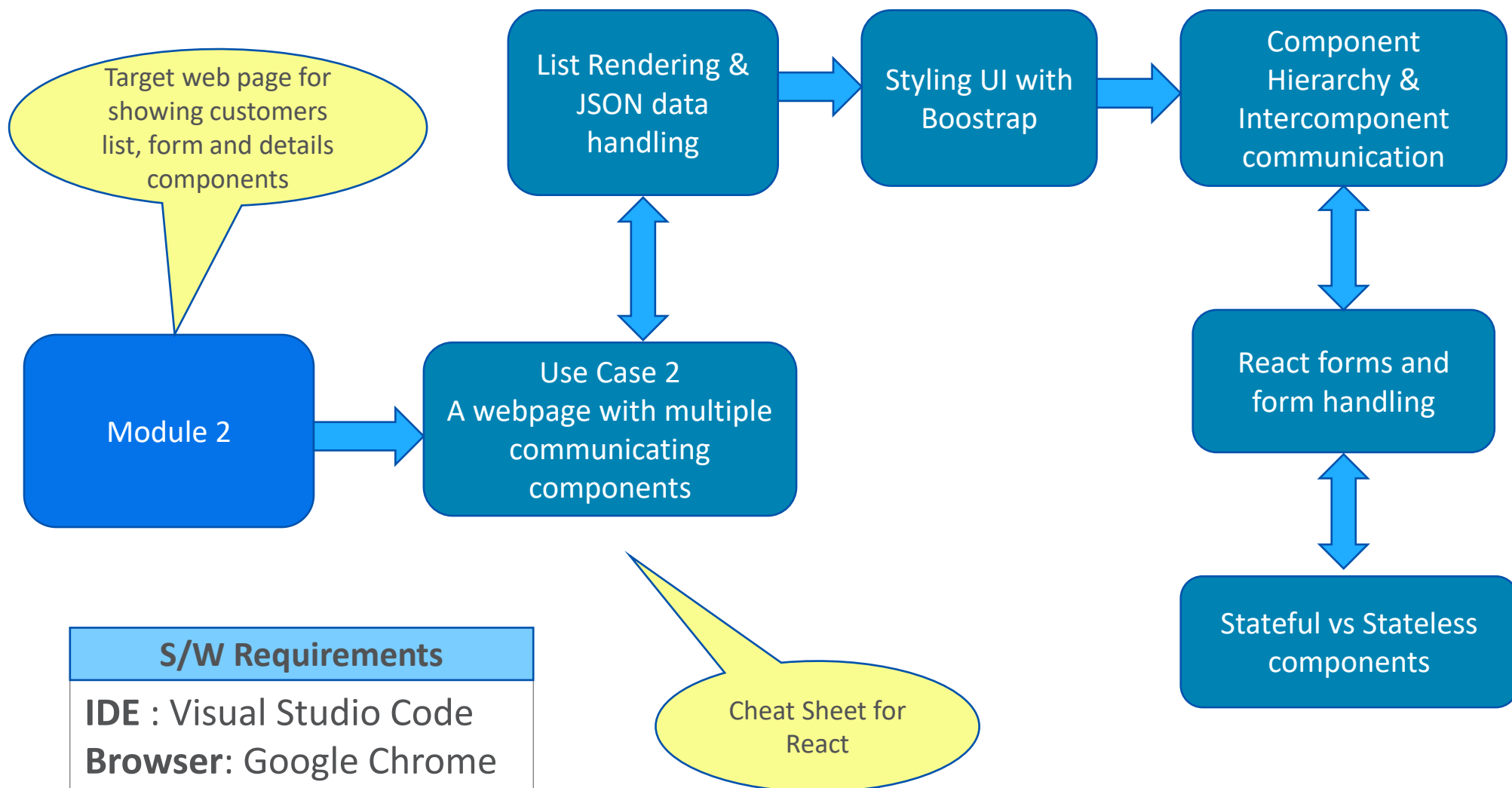
React way of assigning event listeners,
camelCase event name and pass the
event handler function name

Module 2

List rendering, JSON, Bootstrap,
Intercomponent communication, Forms



Module 2 – Design



Module 2 - Overview

- As part of this module, we will cover some advanced concepts of react that will help in building real world components and applications
- The concepts that we are covering in this module are
 - List rendering
 - Using JSON data
 - Installing and adding 3rd party libraries
 - Using bootstrap UI library in our react apps
 - Handlings Forms
- Advanced component hierarchy
- React devtools
- Forms and form handling in React
- Parent to child communication through props
- Child to parent communication through events
- Conditional rendering of components
- Forms Validation

Introducing Use Case – 2



Presenting Use Case 2



Customers List

Id	First Name	Last Name	Email
1	Sundar	Pichai	sundar.pichai@google.com
2	Satya	Nadella	satya.nadella@microsoft.com
3	Jeff	Bezos	jeff.bezos@amazon.com
4	Sergey	Brin	sergey.brin@google.com
5	Larry	Page	larry.page@google.com

Add Customer

First Name

Last Name

Email

Customer Details

ID : 2

First Name : Satya

Last Name: Nadella

Email: satya.nadella@microsoft.com

Use case 2 - Explanation

- As part of this use case, Create a react app with a single page which contains the below components and interactions between them
- Ensure that the entire page has a left and a right margin of equal width
- Create all the below mentioned components under **src/components/customers** subfolder
- Create a **CustomerList** component, which can show a list of customers. For this use case, pick up the customers from a json file named **customers.json**
- The customer data should have 4 fields (id, firstName, lastName, email)
- Ensure that the **CustomerList** component is displayed in the top of the page, with the title “**Customer List**” just above the table
- Create a **CustomerForm** component which contains a form to enter the new customer details. The customer form should have 3 fields i.e First Name, Last Name, Email
- Once the user enters the customer details and hit the submit button, a new customer has to be added to the list in the **CustomersList** component
- Create a **CustomerDetails** component, side-by-side to the **CustomerForm** component.
- When the user clicks/selects a row in the **CustomersList** component, then the **CustomerDetails** component should show up or update itself with the corresponding customer details

Rendering Lists

- You can build collections of elements and include them in JSX using curly braces {}
- Lets assume we want to display a list of items in an unordered list or want to display a list of items in a table, we need to render multiple elements inside a component
- So, we loop through the items array using the JavaScript map() function and return a or <tr> element for each item. Finally we use/assign the array of react elements inside a component

```
// creating the set of list elements
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
|   <li>{number}</li>
);

// rendering the list items
<ul>{listItems}</ul>
```

JSON sample data

- In the previous slide, we handled an array of numbers which is hardcoded in the code
- In many scenarios, we might have to use a JSON file
- E.g. **customers.json** or **listOfCustomers.json**
- A sample JSON file, **customers.json** is shown below

```
[
  {
    "id": 1,
    "firstName": "Sundar",
    "lastName": "Pichai",
    "email": "sundar.pichai@google.com"
  },
  {
    "id": 2,
    "firstName": "Satya",
    "lastName": "Nadella",
    "email": "satya.nadella@microsoft.com"
  }
]
```

Loading JSON data in Components

- To load json data, use the below syntax

```
// importing json file  
import customers from "./customers.json";
```

- JSON data is used commonly during development phase for storing mock data
- JSON can be used for metadata, such as user settings or theme settings etc

Rendering a set of row elements

- Displaying an array of customers in a table row format could be achieved as

```
<tbody>  
  {tabRows}  
</tbody>
```

- `tabRows()` should return the entire set of react elements needed to show the array of customers
- `tabRows()` should return an array of `<tr><tr/>` elements, where in each `<tr>` element should represent one customer data
- The implementation of the `tabRows()` method (presented in the next slide) should render output as shown below

1	Sundar	Pichai	sundar.pichai@google.com
2	Satya	Nadella	satya.nadella@microsoft.com
3	Jeff	Bezos	jeff.bezos@amazon.com
4	Sergey	Brin	sergey.brin@google.com
5	Larry	Page	larry.page@google.com

Rendering set of rows through map

- `tabRows()` - loops through the `customers` array, then creates an array of react elements i.e an array of `<tr>` elements with customer data

```
const tabRows = customers.map((customer, i) => {  
  return (  
    <tr onClick={(e) => onCustomerSelect(e, customer)} key={i}>  
      <td>{customer.id}</td>  
      <td>{customer.firstName}</td>  
      <td>{customer.lastName}</td>  
      <td>{customer.email}</td>  
    </tr>  
  );  
});
```

* *The above component is functional component example.*

- A “**key**” is a special string attribute you need to include when creating lists of elements. This is internally used by react for optimizing how it handles the rendering logic and updates

Bootstrap



Adding Bootstrap to react

- Lets add bootstrap to our react app to make our react components more beautiful, responsive and compatible with most popular browsers,
- To add bootstrap to a react application, use the below command
 - ***npm install bootstrap***
- Enable the styling by adding css import in our index.js or app.js file
 - **import 'bootstrap/dist/css/bootstrap.min.css';**

A table with bootstrap styles

- Use the below bootstrap styles to make a bordered table and less padding between the columns

```
<table className="table table-hover table-bordered table-sm">
  <thead className="thead-light">
    <tr>
      <th>Id</th>
      <th>First Name</th>
      <th>Last Name</th>
      <th>Email</th>
    </tr>
  </thead>

  <tbody>
    {tabRows}
  </tbody>
</table>
```

- React uses the property name **className** to assign css classes, as compared to class keyword in standard html

Table styles – Bootstrap enabled React tables

- We added the most common table styles such as **table**, **table-hover**, **table-bordered**, **table-sm**
 - **table** – will add rows and column lines to make it look like a proper table
 - **table-hover** – when the user hovers the mouse over the table, the row would be selected
 - **table-bordered** – makes the outer borders for the table
 - **table-sm** – controls the spacing between columns (sm – small spacing)
- We also added **thead-light** to the table header, which adds light color filling to the table header
- After applying bootstrap classes, the customer list component should look below

Id	First Name	Last Name	Email
1	Sundar	Pichai	sundar.pichai@google.com
2	Satya	Nadella	satya.nadella@microsoft.com
3	Jeff	Bezos	jeff.bezos@amazon.com
4	Sergey	Brin	sergey.brin@google.com
5	Larry	Page	larry.page@google.com

Code Snippet

```
export default function Customers() {
  const onCustomerSelect = (e, customer) => console.log(customer)
  const tabRows = customers.map((customer, i) => {
    return (
      <tr onClick={(e) => onCustomerSelect(e, customer)} key={i}>
        <td>{customer.id}</td>
        <td>{customer.firstName}</td>
        <td>{customer.lastName}</td>
        <td>{customer.email}</td>
      </tr>
    );
  });
  return (
    <table className="table table-hover table-bordered table-sm">
      <thead className="thead-light">
        <tr>
          <th>Id</th>
          <th>First Name</th>
          <th>Last Name</th>
          <th>Email</th>
        </tr>
      </thead>
      <tbody>
        {tabRows}
      </tbody>
    </table>
  );
}
```

Intercomponent communication

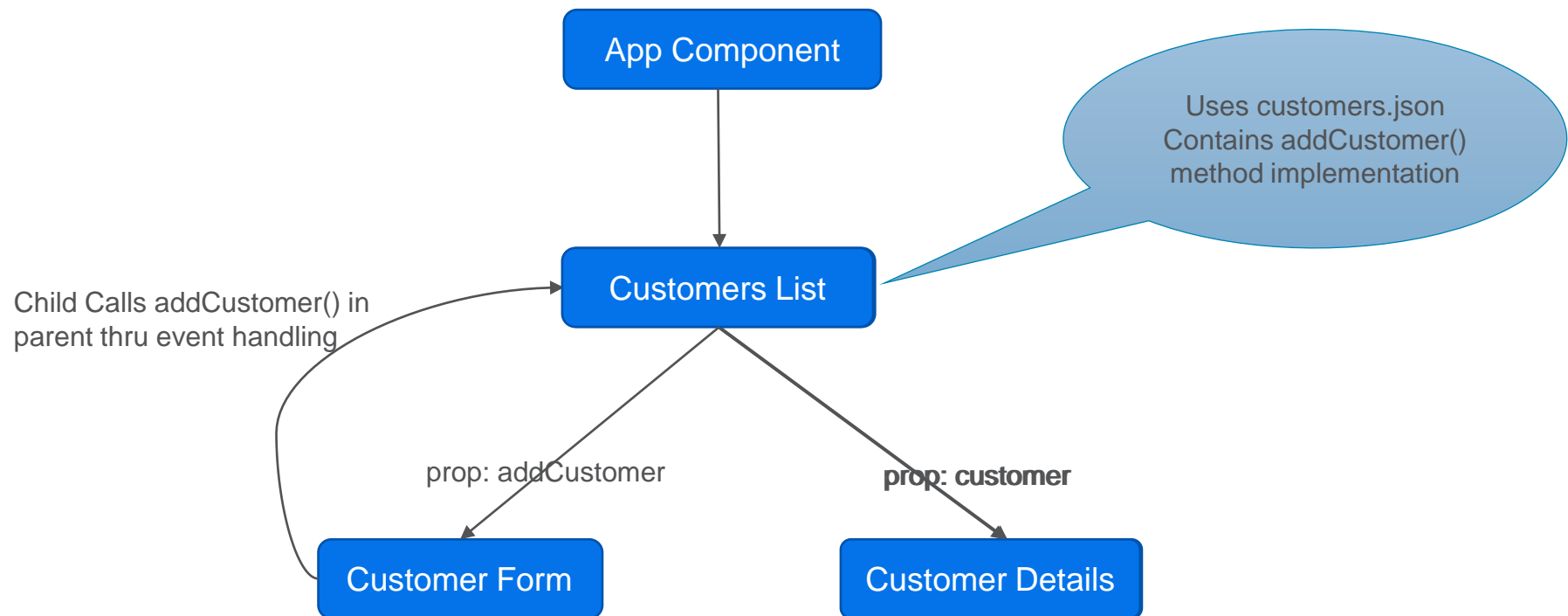


Intercomponent communication

- In the first module, we saw how to develop standalone components. We looked at props, state and event handling
- But in real world applications, we will have more complex component hierarchy, where in components have to communicate to each other to exhibit their behavior
- The ultimate goal of component oriented ui development approach is to define exactly what the component is internally made up of and what the component expects from the outside world

Two level component hierarchy

- The below picture shows the 2 level component hierarchy
- App is the root component, that contains the child component named **CustomerList**. The **CustomerList** component in turn contains 2 children. i.e **CustomerForm** and **CustomerDetails** component

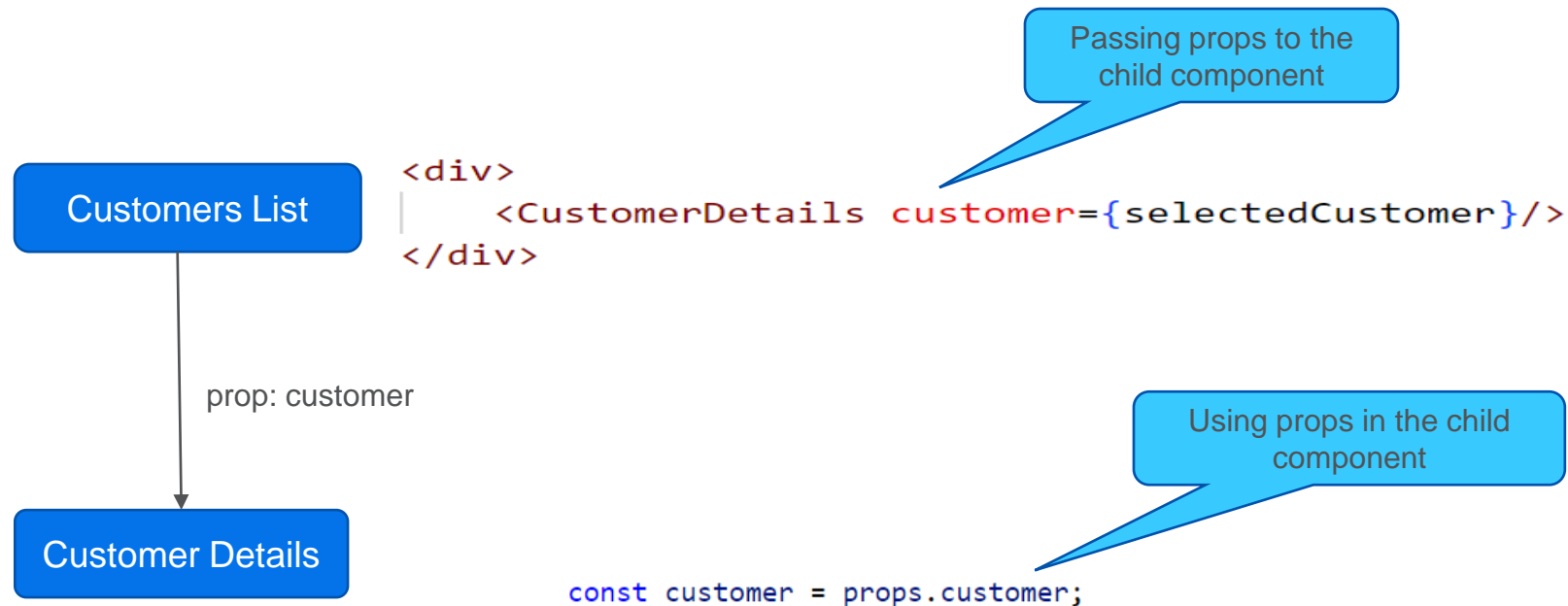


More about component hierarchy

- Real world application comprises of many components which interact with each other. When a component's complexity increases, it is better to split it up into multiple components and have communication between them
- Parent to child communication can happen by sending data through props object, in our example, the **CustomerList** component sends **customer** as the prop to the **CustomerDetails** component
- Child to parent communication can happen by calling a method in the parent. Child can know which method to call based on what the parent sends in the props object. E.g. **addCustomer** is the method name which is sent in the props, which in turn would be called by the child
- The actual implementation of **addCustomer** method is present in the parent component(in our case, it is **CustomerList** component)

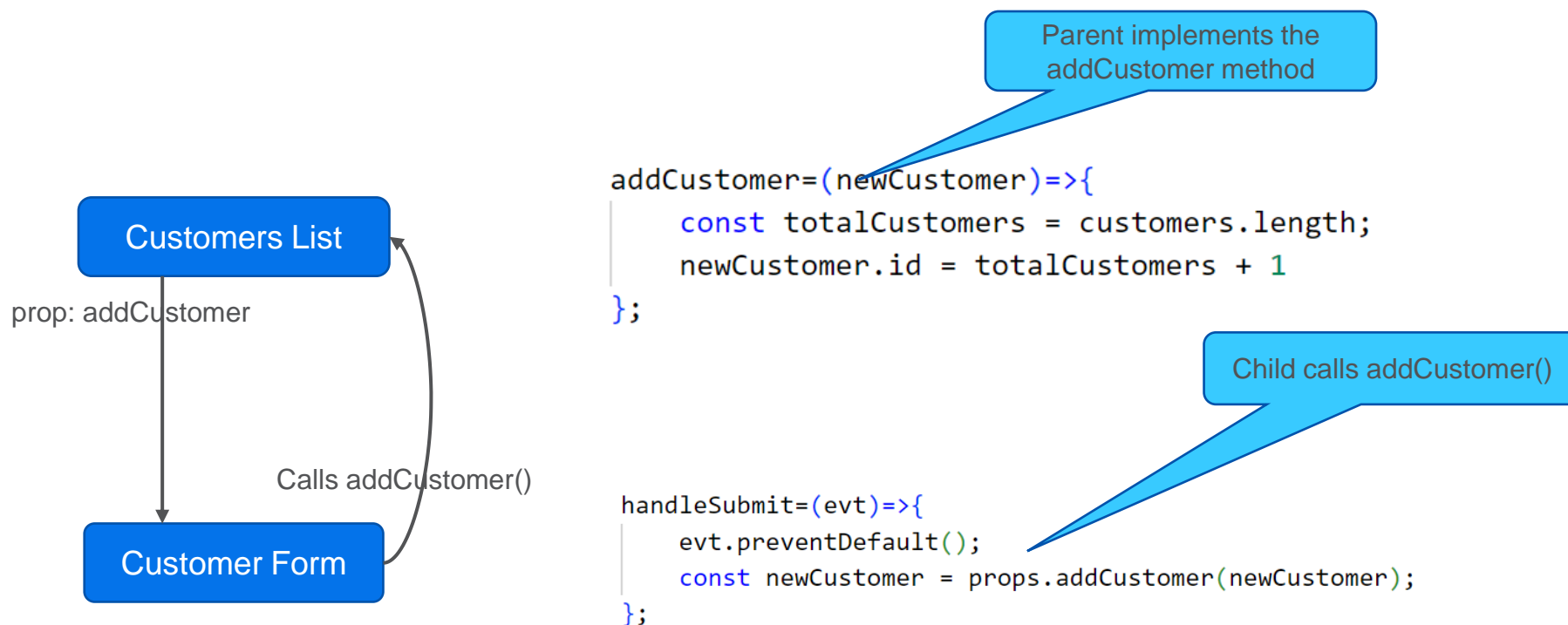
Parent to child communication

- React apps are made up of multi-level component hierarchies (a tree of components)
- In a tree of components, there is a need to communicate data from the parent to the child or vice-versa
- In react, any parent component can send data to the child component through the props object



Child to parent communication

- Child components can also communicate back to the parent
- Child cannot pass data to the parent through props. Rather child component communicates with the parent by calling a method in the parent component
- It can also pass data to the parent through the method arguments



Forms in React



Forms in React

- The standard way to do forms in React is to use what are called as “**controlled components**”
- In html, form elements such as `<input>`, `<textarea>` and `<select>` etc maintain their own state and update it based on user input
- In React, any mutable state is kept in the state property of functional components and only updated with ***useState()*** hook.
- In the standard approach of dealing with forms, we can combine the above two by making the React state be the “single source of truth”

Form handling – listening to changes

- Since the value attribute is set on our form element, the displayed value will always be **firstName** which is assigned to **useState**, making the React state the source of truth

```
<form onSubmit={handleSubmit} className="form">
  <div className="form-group">
    <label>First Name</label>
    <input
      type="text"
      name="firstName"
      className="form-control"
      value={firstName}
      onChange={handleFirstNameChange}
    />
  </div>
</form>
```

- Since **handleFirstNameChange** runs on every keystroke to update the React state, the displayed value will update as the user types

```
const handleFirstNameChange = (e) => {
  setFirstName(e.target.value)
}
```

Forms Validation



Form validations

- In React forms, validations could be done using regular JavaScript techniques or code
- In real world apps, 3rd party validation libraries could also be used, that would simplify the validation logic
- Validation could happen during various user interactions
 - When the user submits the form
 - Or when the user is entering inputs in the form fields
- So, forms could be made as interactive as possible using clientside validations
- Just before the form is getting submitted, we can apply validation logic as shown below

```
handleSubmit=(evt)=>{  
  evt.preventDefault(); //prevents the default form submission behaviour  
  if (validate()){  
    //Form is valid, proceed with form submission  
    console.log('....')  
  }  
};
```

- If valid, the data is sent to the server, else errors should be shown in the form or webpage

Form validations

- A simple form validation method just before submitting customer form is shown below

```
const validateCustomer = () => {  
  let validationErrors = {};  
  if (!customer.firstName)  
    validationErrors.firstName = "FirstName can not be empty";  
  if (!customer.lastName)  
    validationErrors.lastName = "LastName can not be empty";  
  if (!customer.email) validationErrors.email = "Email can not be empty";  
  
  return validationErrors;  
};
```

Displaying Form Errors

- We have to apply dynamic styles to our forms so that on form submission, any error should be indicated as shown below

Add Customer

First Name

First Name cant be empty

Last Name

Last Name cant be empty

Email

Email cant be empty

Phone

Submit

Displaying and styling error messages

- While doing form validations, its important to differentiate valid and invalid fields
- Also, error messages specific to individual columns could be displayed
- There are some styles available in bootstrap library for handling success and error scenarios
 - has-error
 - has-success
- We have to apply the error classes only if any errors are there. So, we have to apply bootstrap styles dynamically
- There is a very useful 3rd party library named **classnames**, which can be used to apply css styles dynamically
- Install the 3rd party library using the below command
 - ***npm install classnames***

Handling events – e.preventDefault()

- In a standard browser, whenever a user submits the form, the page refreshes after making a post request. Also, when the user clicks on a href link, the browser page gets redirects or refreshes
- But when we develop single page apps with React, we don't want the browser page to get refreshed. We want a seamless behaviour without reloading the page
- To achieve this, whenever form submission happens or the link is clicked, we have to capture the event and call **evt.preventDefault()** to suppress the browser's default behaviour

```
handleSubmit=(evt)=>{  
  evt.preventDefault();  
  if (validate()){  
    console.log('....')  
  }  
};
```

```
<form onSubmit = {handleSubmit}>  
  <div>  
    <label>First Name</label>  
  </div>  
</form>
```

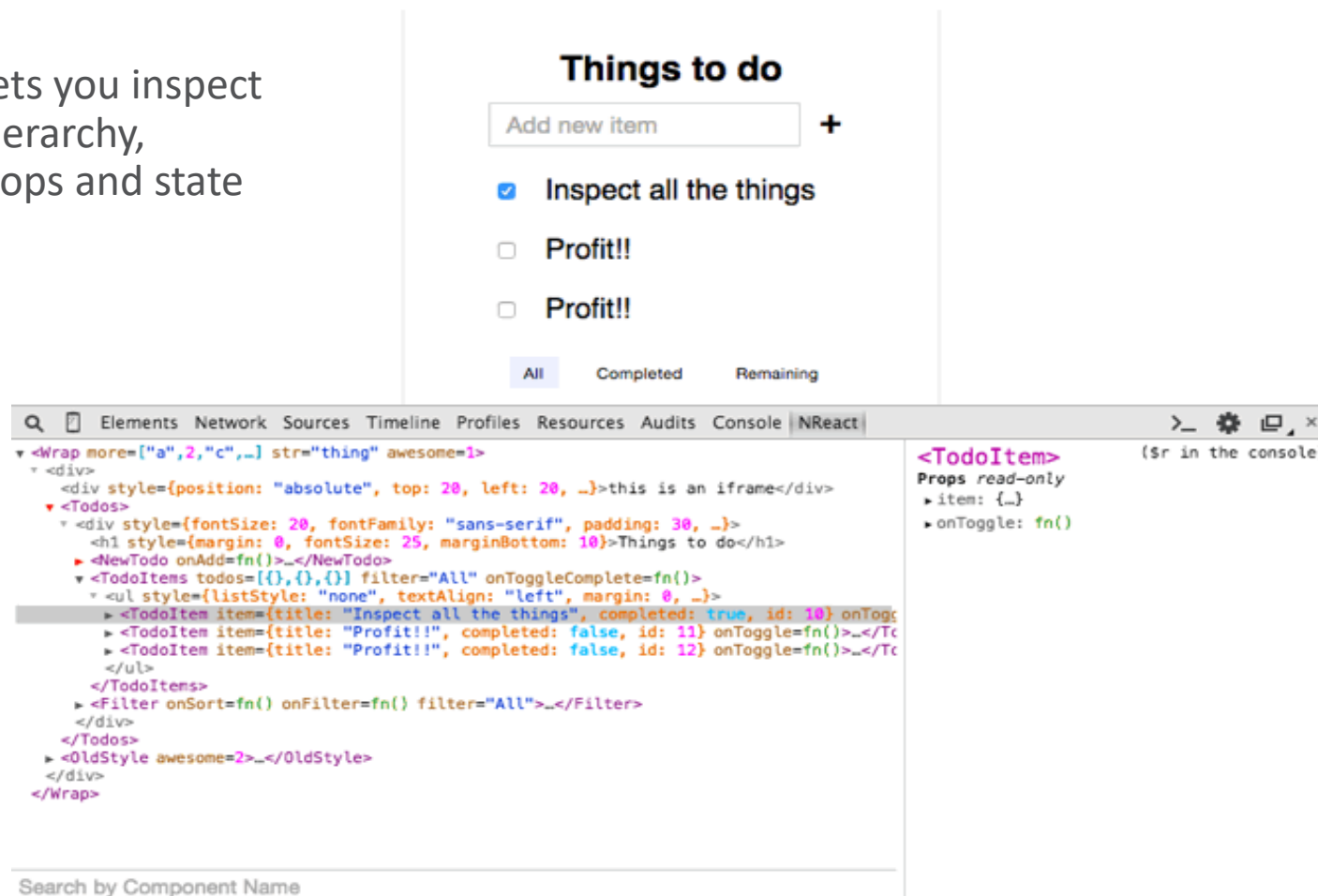
- Here, event is a synthetic event. React defines these synthetic events according to the W3C spec, so you don't need to worry about cross-browser compatibility

React Developer Tools



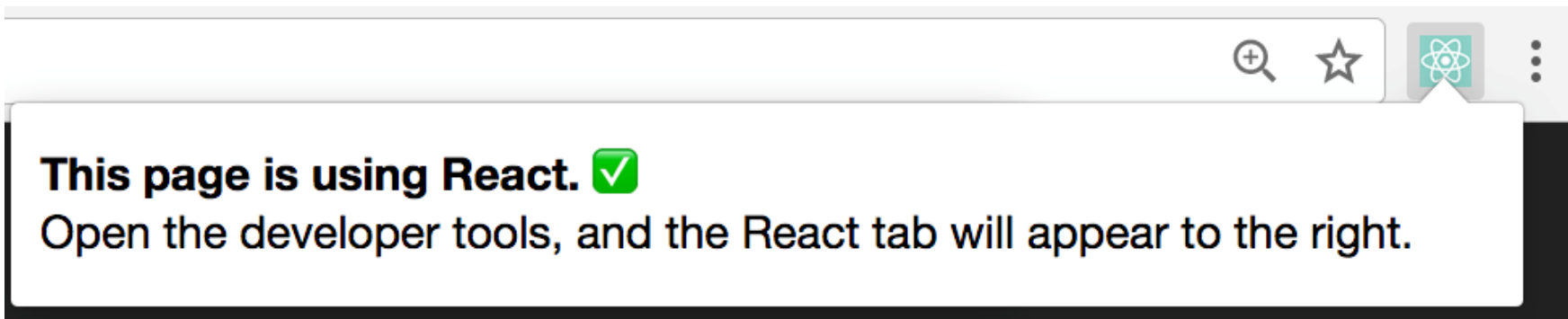
React Developer Tools

- React Developer Tools lets you inspect the React component hierarchy, including component props and state



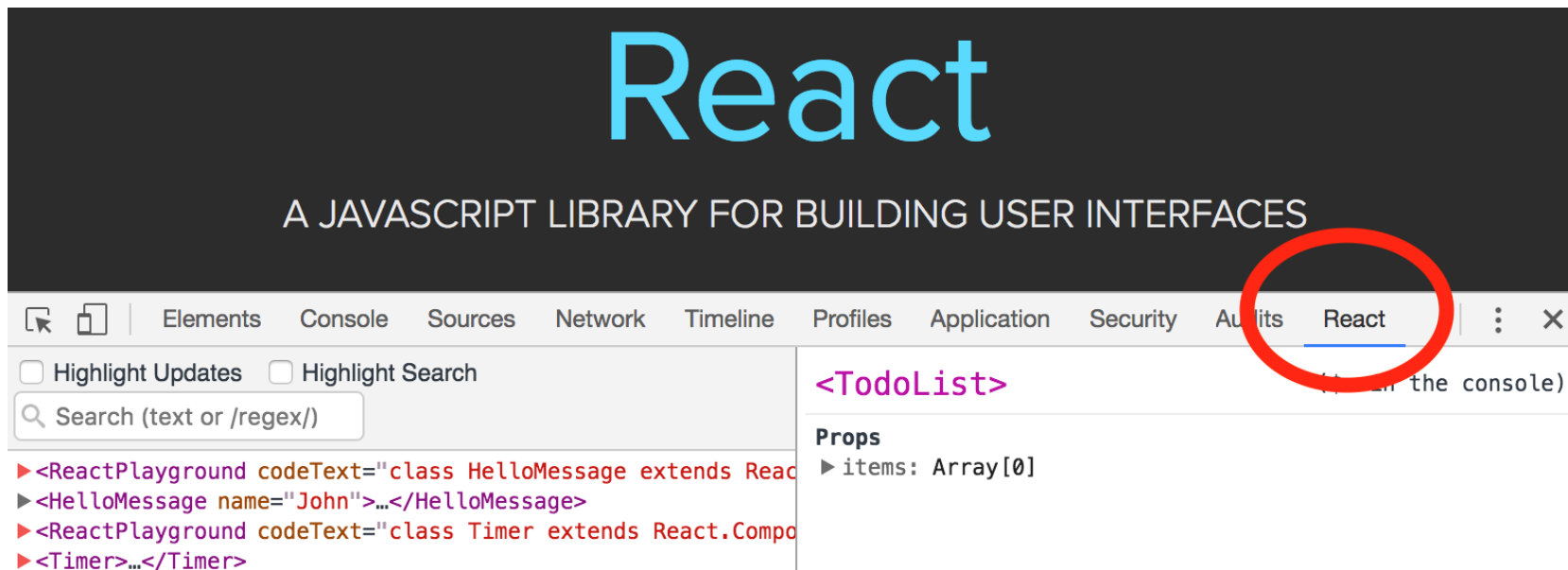
React Developer Tools - Usage

- Its popularly know as React Devtools
- It exists both as a browser extension (for Chrome and Firefox), and as a standalone app
- The extension will light up on the websites using React



React Devtools tab

- On the websites that use React, you will see a tab called React in Chrome Developer Tools
- A quick way to bring up the DevTools is to right-click on the page and press Inspect



Inspecting use case 2 components with Devtools

- Devtools showing the components hierarchy for use case 2 (also showing the current state/props of CustomerForm)

3	Jeff	Bezos	jeff.bezos@amazon.com
4	Sergey	Brin	sergey.brin@google.com
5	Larry	Page	larry.page@google.com

Add Customer

First Name

Last Name

Email

The screenshot shows the Chrome DevTools interface with the React component inspector open. The component hierarchy is displayed on the left, showing the following structure:

```
<App>
  <div className="App">
    <CustomerList>
      <div>
        <h3>Customers List</h3>
        <table className="table table-hover table-bordered table-sm">...</table>
        <br>
        <br>
        <div className="row">
          <div className="col-md-6">
            <CustomerForm>...</CustomerForm>
          </div>
          <div className="col-md-2">
            <div className="col-md-4">
              <CustomerDetails>...</CustomerDetails>
            </div>
          </div>
        </div>
      </div>
    </CustomerList>
  </div>
</App>
```

The right panel shows the props and state for the selected **CustomerForm** component:

Props

- `addCustomer`: bound `addCustomer()`

State

- `email`: "rock.dwayne@gmail.com"
- `firstName`: "Rock Dwayne"
- `lastName`: "Johnson"

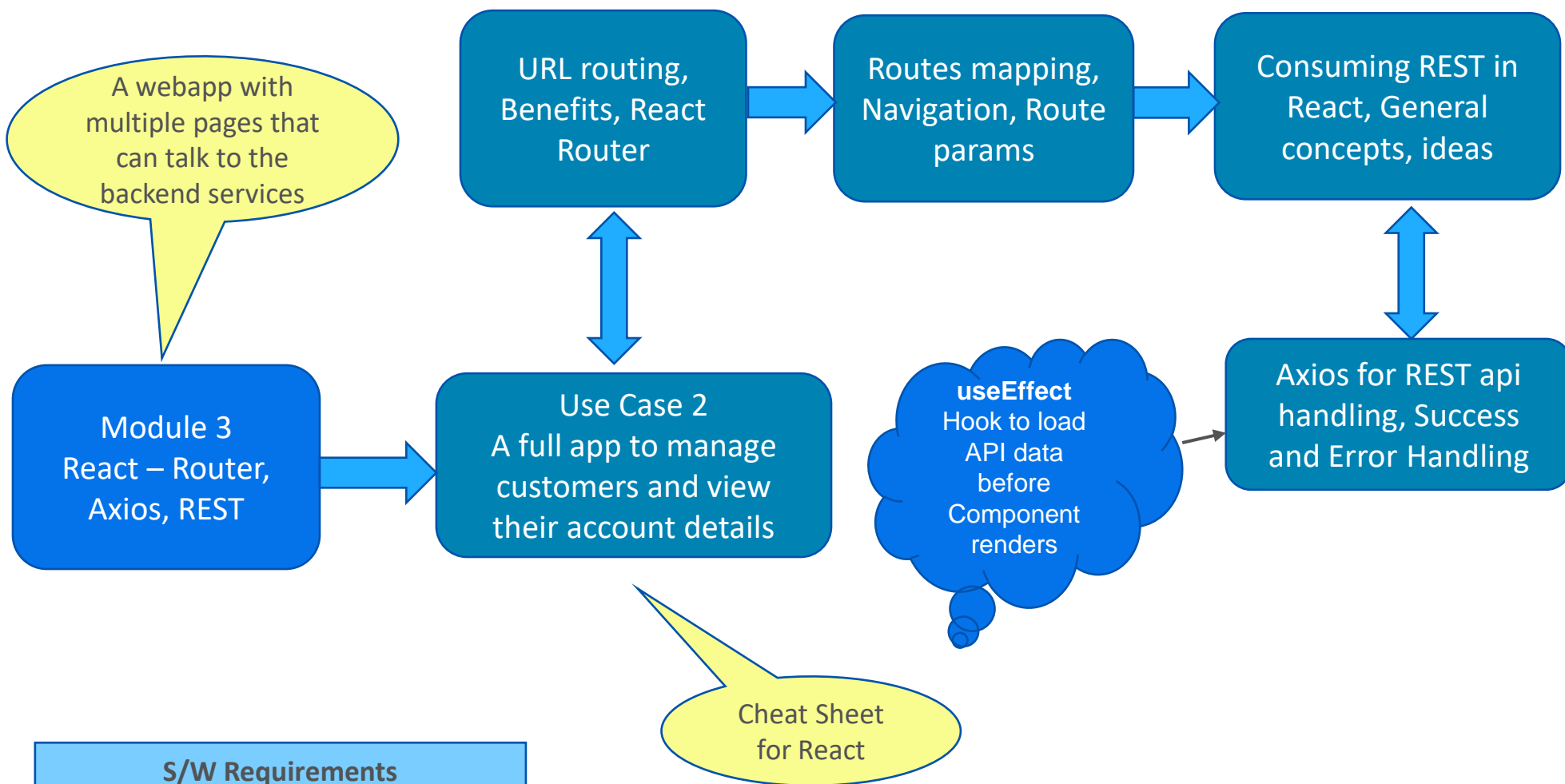
The breadcrumb at the bottom of the component inspector shows the path: **App** > **div** > **CustomerList** > **div** > **div** > **div** > **CustomerForm**.

Module 3

React Router, Axios and REST



Module 3 – React advanced concepts



S/W Requirements

IDE : Visual Studio Code
Browser: Google Chrome

Module 3 - Overview

- As part of this module, we will cover topics such as Routing, Http services and Form validation
- The concepts that we are covering in this module are
 - URL based routing
 - Installing and configuring React router
 - Using route parameters
 - URL Links, navigation and redirection
 - A full set of CRUD pages using react-router
- Basics of consuming REST services in React
- Installing and using axios
- Full CRUD operations using axios
- useEffect Hook and its usages
- Handling success and Error scenarios
- React build and deployment techniques

Use case 3 - Explanation



- Create a react app to manage customers data and view their account details
- The user/admin of this app should be able to manage the CRUD operations of customers
- The admin should be able to view the account details of the selected customer

TopGuns Bank

Home Customers About SCB

[Create new customer](#)

Customers List

Id	First Name	Last Name	Email	Actions
1	Sundar	Pichai	sundar.pichai@google.com	Show Edit Delete
2	Jeff	Bezos	jeff.bezos@amazon.com	Show Edit Delete
3	Satya	Nadella	satya.nadella@microsoft.com	Show Edit Delete
4	Sergey	Brin	sergey.brin@google.com	Show Edit Delete
5	Larry	Page	larry.page@google.com	Show Edit Delete

Use case 3 - Explanation

- As shown in the previous slide, the app should have a navigation bar in the top, using which we navigate to the home page, about page, customers page etc
- The customers page should display all the customer details in a table format, with options/links to do all CRUD operations on the customer data
- When the user clicks on “**Show**” link, the app should be redirected to the customer details page
- Create/Edit links should take the user to the customer form page, where in the create/edit action could be done
- Delete link, when clicked, should ask for confirmation and delete the customer entry from the list
- The home page and about page should have proper information about the bank, and the role of this application in the bank respectively

Use case 3 – Customer Details Page



- The details page should show customer’s details, his/her accounts holding information. It should also have a navigation link back to the customer’s list page

TopGuns Bank

HomeCustomersAboutSCB

[< Back to Customers List](#)

Customer Details

ID : 1

First Name : Sundar

Last Name: Pichai

Email: sundar.pichai@google.com

Phone:

List of Accounts

Account No	Type	Branch	Balance
1001999	SAVINGS_ACCOUNT	Bellandur	1000
1001888	SAVINGS_ACCOUNT	Indira Nagar	2000

Use case 3 – Customer Form Page

- The create/edit links or action should redirect the app to the customer form as shown
- The form should validate if the fields are not empty and are in right format
- Once the form is submitted, the control should get redirected to the list page, if customer creation/update was successful
- If failure, the form should display error message and wait for the user action
- The form should also have a navigation button back to the customers list page

[< Back to Customers List](#)

Add Customer

First Name

Last Name

Email

Phone

Create Customer

Routing in React



Routing in React

- Traditionally, Routing was used in the server side for mapping URLs to html pages
- But in SPA frameworks such as React, the entire control is in client side
- When the user visits a URL, different components could be rendered based on the URL mapping
- This method of routing is also called a component routing
- The following table shows a sample routing configuration

Route URL	Component
‘/about’	About
‘/home’	Home
‘/customers’	CustomersList
‘/customers/new’	CustomerForm
‘/customers/145’	CustomerDetails

What is use of React Routing?

- Single page apps became mainstream to enable desktop like user experience inside the browser
- Then why is URL routing so important in single page apps that are created by React
- The main uses of URL routing are
 - URLs can be bookmarked in browsers for reference later
 - URLs can be shared with others
 - Search engine friendliness
- Routing is not mandatory in react apps, but for the above reasons it could be enabled by using 3rd party libraries

How to enable routing?

- There are many libraries to enable routing in react apps
- The most popular one is the **React router**
- React router provides a declarative way to define routes
- To install react router, use the below command
 - ***npm install react-router-dom***
- The library is called ***react-router-dom*** because *it is targeted for web routing as compared to routing in react-native mobile apps*

React router

- React router provides three types of components for working with routing
 - Router components
 - Route matching components
 - Navigation components
- React router provides a declarative way to do routing, hence react router components also look like typical react components

```
<Route path='/customers' component={CustomersPage}/>
```

Router components

- At the core of every react router application, there should be a router component
- For web projects, *react-router-dom* provides **`<BrowserRouter />`**
- **`<BrowserRouter />`** can be used when we have a server that responds to requests
- In a react router application, the entire app/root component should be encompassed inside a ***BrowserRouter*** component

```
import { BrowserRouter } from "react-router-dom";
```

```
<BrowserRouter>  
|   <App />  
</BrowserRouter>
```

Route matching components

- The route matching components can be used to control what components need to be rendered based on location of the web page
- There are 2 types of route matching components
 - **`<Route />`**
 - **`<Link />`**
- You can include a **`<Route>`** anywhere that you want to render content based on the location
- Route matching is done by comparing a `<Route>`'s path prop to the current location's pathname

```
<Routes>
  <Route path="/" element={<Home />}></Route>
  <Route path="/customer" element={<Customer />}></Route>
</Routes>
```

- In the above example, we have seen only static strings in the path
- If we have match dynamic URLs such as **`'/customers/1'`**, **`'/customers/2'`** etc. we have to use regex pattern matching as shown below

```
<Route path="/customers/:id" component={CustomerDetails} />
```

Route matching - Link

- The **<Link>** component can be used to group <Route>s together

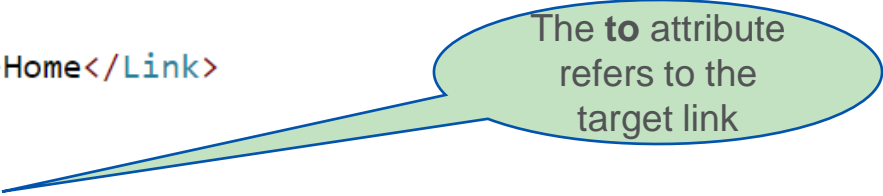
```
<div>  
  <Link to="/">Home</Link>|&nbsp;nbsp;  
  <Link to="/customer">Customer</Link>  
</div>
```

- The **exact** attribute is used to indicate that the route path should be exactly matched and not just prefix matching
- For e.g. the "/" path might match every possible component that starts with "/", but if we use exact, it will not match any other path matching the prefix

Navigation components

- When we want to create links in our application, React Router provides a **<Link>** component
- Whenever we render a **<Link>**, an anchor **<a>** will be rendered in your application's html

```
<li class="nav-item">
  <Link className="nav-link" to="/">Home</Link>
</li>
<li class="nav-item">
  <Link className="nav-link" to="/customers">
    Customers
  </Link>
</li>
```



The **to** attribute
refers to the
target link

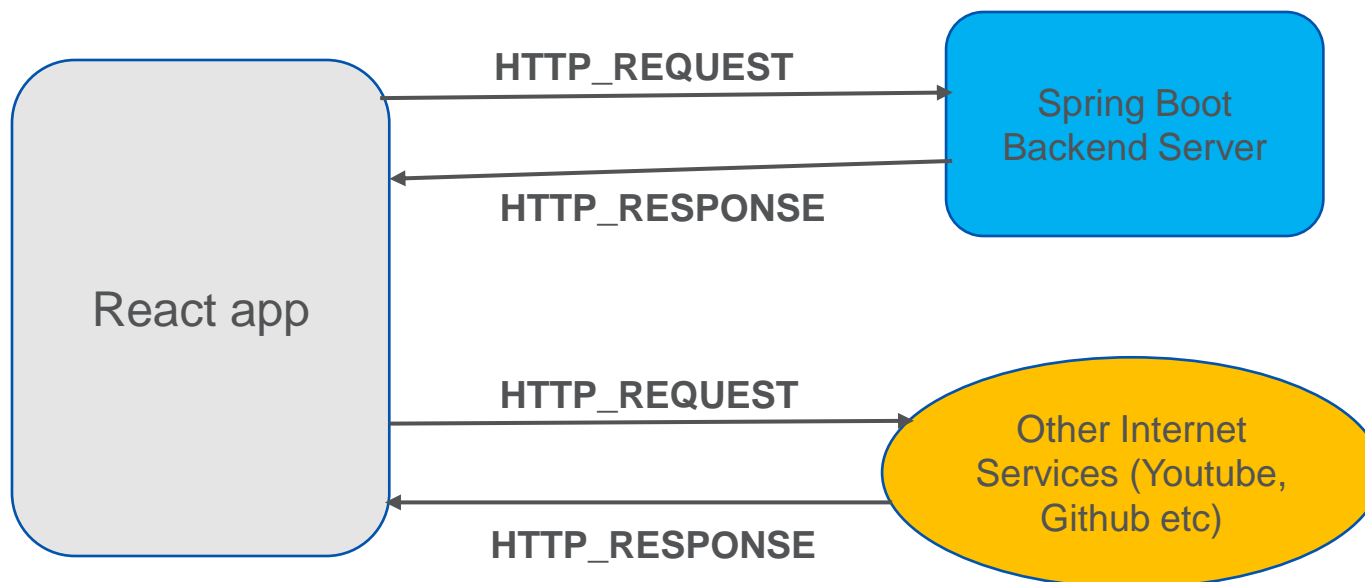
- The **<Link>** component has an attribute named **to**, which can be used to set the target navigation link

REST services



Consuming REST services

- Most of the modern web apps communicate with a REST backend to get the actual data
- Till now, we have only seen how to work with local JSON data
- Because all browser based apps rely on HTTP protocol, REST is the most popular architecture to consume data across the internet
- REST services are called through HTTP and data format used is JSON
- React apps can communicate with any internet services or our own backend server through HTTP and REST using JSON as the data format



How to consume HTTP services in React

- We can use the native browser methods such as `fetch()` in order to call http services from our react app
- But in order to simplify and reduce a lot of boilerplate code, some popular 3rd party libraries are used in order to consume http services
- The most common in the React ecosystem is a library named axios
- Please see <https://github.com/axios/axios> for complete details
- The main features of axios library are
 - Make **XMLHttpRequests** from the browser
 - Supports the **Promise** API
 - Intercept request and response
 - Automatic transforms for JSON data

Axios Installation and Usage

- Axios can be installed and added to our react app using the below command

- ***npm install axios***

```
import axios from 'axios';
```

- Axios can be imported using the syntax
- A sample axios usage is shown below

```
// Make a request for a user with a given ID
axios.get('/user?ID=12345')
  .then(function (response) {
    // handle success
    console.log(response);
  })
  .catch(function (error) {
    // handle error
    console.log(error);
  })
  .finally(function () {
    // always executed
  });
```

useEffect Hook

- useEffect hook lets us perform side-effects in function component.
- Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects.
- useEffect (callback, dependencies) takes two arguments:

```
export default function UseEffectExample () {  
  
  const [count, setCount] = useState(0)  
  
  useEffect(() => {  
    document.title = `You clicked ${count} times`  
  }, [count])  
  
}
```

- callback function - This function we pass is our effect. When React renders our component, it will remember the effect we used, and then run our effect after updating the DOM.
- dependencies is an optional array of dependencies. `useEffect()` *executes callback only* if the dependencies have changed between renderings.

useEffect() Dependencies

- dependencies argument of `useEffect(callback, dependencies)` lets you control when the side-effect runs. When dependencies are:
 - Not provided: the side-effect runs after every rendering.

```
export default function UseEffectExample () {  
  
  const [count, setCount] = useState(0)  
  
  useEffect(() => {  
    document.title = `You clicked ${count} times`  
  })  
  
}
```

- An empty array `[]`: the side-effect runs once after the initial rendering.

```
export default function UseEffectExample () {  
  
  const [count, setCount] = useState(0)  
  
  useEffect(() => {  
    document.title = `You clicked ${count} times`  
  }, [])  
  
}
```

useEffect() Dependencies continued

- Has props or state values [prop1, prop2, ..., state1, state2]: the side-effect runs only when any dependency value changes.

```
export default function UseEffectExample () {  
  
  const [count, setCount] = useState(0)  
  
  useEffect(() => {  
    document.title = `You clicked ${count} times`  
  }, [count])  
  
}
```


Axios – HTTP GET operation

- The HTTP GET operation is done using **axios.get()** method. For eg, we can use this method to fetch a list of customers or a single customer details (for anything that is read only)
- The CRUD operations could be triggered on various user events or interaction
- When we are loading initial data for a functional component from the backend, its important to make the HTTP get calls in the **useEffect Hook**, so that the component is ready and mounted

```
useEffect(() => {  
  axios.get(API_URL)  
    .then(response => {  
      setUsers(response.data)  
    })  
    .catch(error => {  
      setUsers([])  
    })  
}, [])
```

- The above **axios.get()** method is making a HTTP GET operation on the URL **http://localhost:8080/api/customers**

Promises Usage

- In the modern day JavaScript development, Promises are very common
- Functional programming JavaScript are achieved through callbacks and promises provide an easy way to program with callbacks
- A typical promise usage, as used in axios is shown below

```
axios.get("http://localhost:8080/api/customers/" + custId)  
  .then(res => console.log(res))  
  .catch(err => console.log(err));
```

- Axios get method is returning a promise object and hence we are able to do a **.then()** and a **.catch()** call on it

Promises Usage – More clarification

- If we split the **axios.get()** call code into multiple statements, we can understand it much better
- First receive the promise object

```
// the axios.get() method is returning a promise object
const getPromiseObj = axios.get("http://localhost:8080/api/customers/" + custId);
```

- Now, define the actions for success using .then() and error scenario using .catch() methods

```
// when .then() is called on promise, the promise is resolved
getPromiseObj.then(res => console.log(res)) // handling success
| | | | | | | .catch(err => console.log(err)); // handling error
```

Axios – HTTP POST operation

- POST is used for creating a new customer entity in the backend database
- The HTTP POST operation is done using **axios.post()** method
- POST operations are generated from html forms where user input various types of data
- A sample axios.post() call is below

```
axios
  .post("http://localhost:8080/customer/addCustomer", newCustomer)
  .then((response) => {
    console.log(response.data);
  })
  .catch((error) => {
    console.log(error.data);
  });
```

- In a post call, the first parameter is the URL to which the POST is done and the 2nd parameter is the POST data, in this case it's the new customer data

Axios – PUT operation

- HTTP PUT operation is used for updating entities
- When the user does edit/update operation on any entity(e.g.customer), PUT is used
- A sample PUT call could be done using axios as below

```
axios
  .put("http://localhost:8080/customer/addCustomer"+custId, customer)
  .then((response) => {
    console.log(response.data);
  })
  .catch((error) => {
    console.log(error.data);
  });
```

- PUT operation is similar to POST, where the 1st parameter is the URL and 2nd parameter is the data that is updated
- **PUT** on **http://localhost:8080/api/customers/123**

Axios – delete operation

- HTTP DELETE method is used for delete operation
- The HTTP POST operation is done using **axios.delete()** method.

axios

```
.delete('http://localhost:8080/customer/addCustomer/${custId}')  
.then((response) => {  
  console.log(response.data);  
})  
.catch((error) => {  
  console.log(error.data);  
});
```

Thank You

