Full Stack Development

20ITL601/SO04

UNIT -1

- **Introduction:** MERN, MERN Components, Serverless Hello World Application, ES6, DOM, Virtual DOM, Installation.
- **Components** in React, using JSX, React Project Structure, State, create-react-app, Props, State, Component Life Cycle, Handling Events, Component Communication, Data Binding – One way, two way, SPA.
- Working with Forms & Third Party libraries, Routing.

List of Experiments:

1. Develop a Calculator React App
2. Develop a News feed application
3. Develop React application using Forms
4. Develop a website to demonstrate React Routing

UNIT – II

React.JS & node.JS:

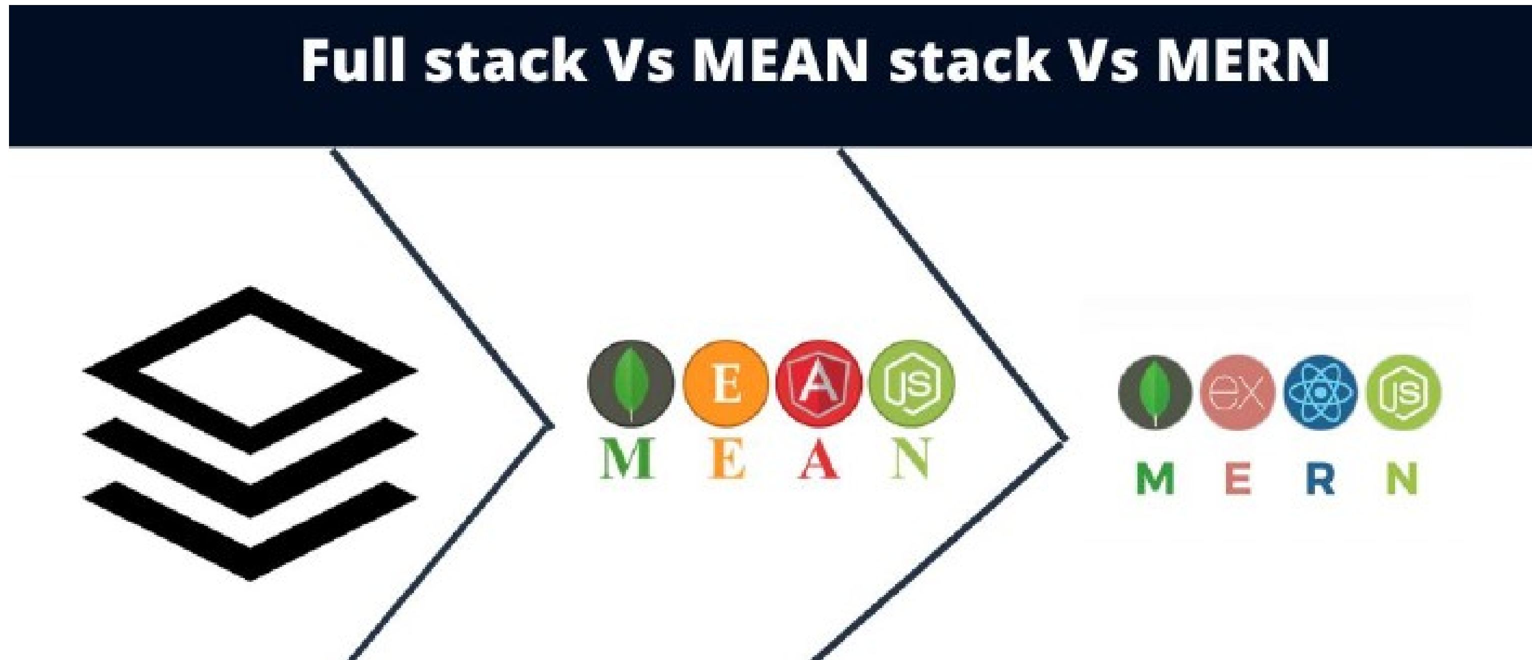
UNIT -III

- Accessing the File System from Node.js, Implementing HTTP Services in Node.js.
- **Express** with Node.js, Routes, Request and Response objects, Template engine.
- Understanding middleware, Query middleware, Serving static files, Handling POST body data, Cookies, Sessions, Authentication.

UNIT – IV

- **MongoDB:** Understanding NoSQL and MongoDB, Getting Started with MongoDB, Getting Started with MongoDB and Node.js, Manipulating MongoDB Documents from Node.js, Accessing MongoDB from Node.js.

Full Stack vs MERN vs MEAN Stack



MEAN → **M**ongo DB, **E**xpress, **A**ngular.js, **N**ode.js
MERN → **M**ongo DB, **E**xpress, **R**eact.js, **N**ode.js

Stack:

Stack consists of programming language, software products, and Technologies.

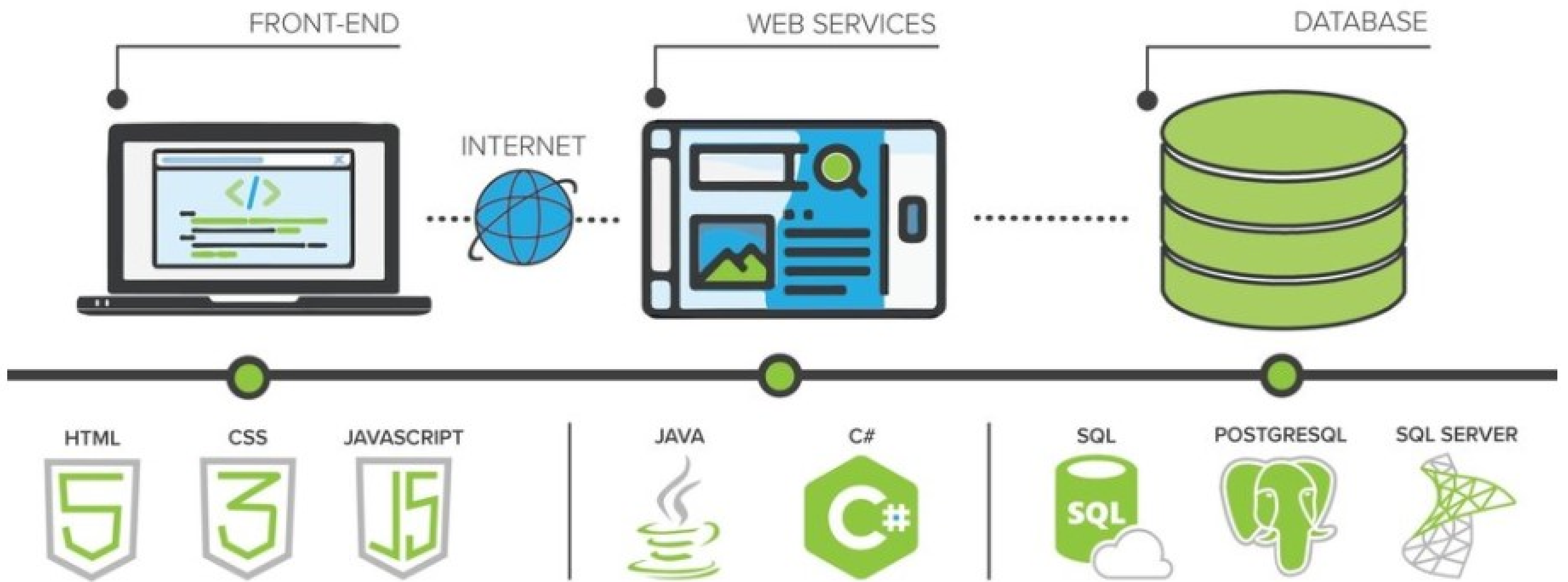
- A website or an app that you see has two aspects to it – the front-end and the back-end. Both aspects need to be handled professionally and properly to achieve the desired look and operational functionalities.

Full Stack:

- Full-stack refers to a set of tools and programming languages that a full stack developer uses to create both, the front-end and the back-end of an app or a website. So, the developer is experienced and trained to design the best UX/UI as well as handle multiple technologies at the back-end.
- A full-stack developer has the flexibility to switch from front-end to back-end, as and when required.
- A full-stack developer is so experienced that he can work on an entire app or website design structure and to different levels of involvement.

Adv: Companies do not have to hire separate teams for developing the front-end and the back-end.

Example: LAMP (Linux, Apache, MySql, PHP) stack.


FRONT-END
INTERNET
WEB SERVICES
DATABASE
HTML
CSS
JAVASCRIPT
JAVA
C#
SQL
POSTGRESQL
SQL SERVER
SQL

MEAN Stack

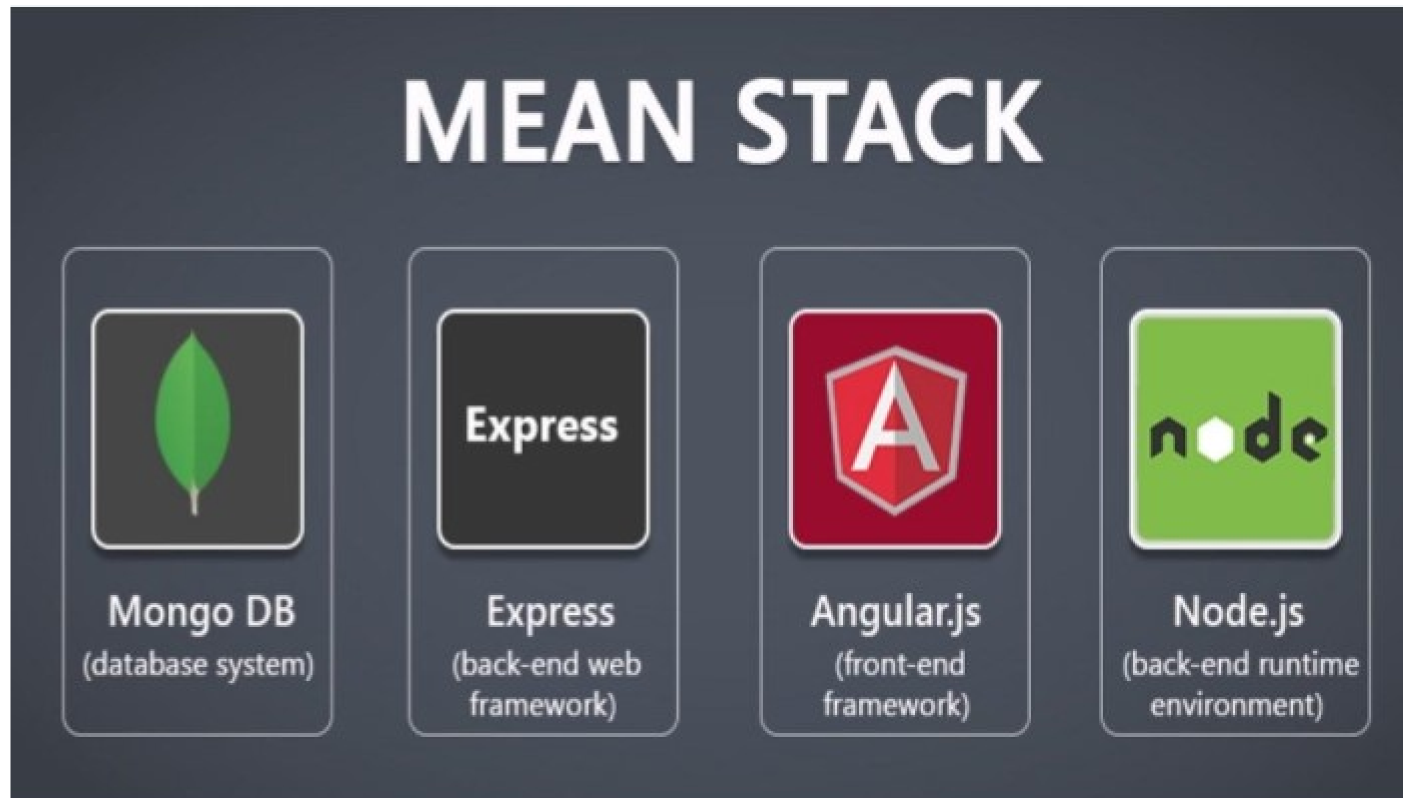
MEAN Stack:

The mean stack is a full-stack JavaScript framework. MEAN is the short form of MongoDB, Express.js, Angular, and Node.js.

This technology is used by Mean Stack developers to make hybrid mobile apps and web apps.

- **MongoDB** is used for storing JavaScript Object Notation documents;
- **Express.js** is used as a back-end web application framework;
- **Angular.js** is the front-end web app development framework, and
- **Node.js** is used for the implementation of the application back-end in JavaScript. Is an asynchronous event-driven JavaScript runtime.
  - **Allows JavaScript to be used to write server-side code.**
  - Node.js is a JavaScript server environment that runs code outside a browser i.e on the server.
  - The technology is a perfect fit for many websites like **streaming (Netflix, games, time trackers, social media applications – Twitter, Linkedin, Uber, ebay)**
  - It provides a scalable and fast solution for real-time applications.

MEAN Stack

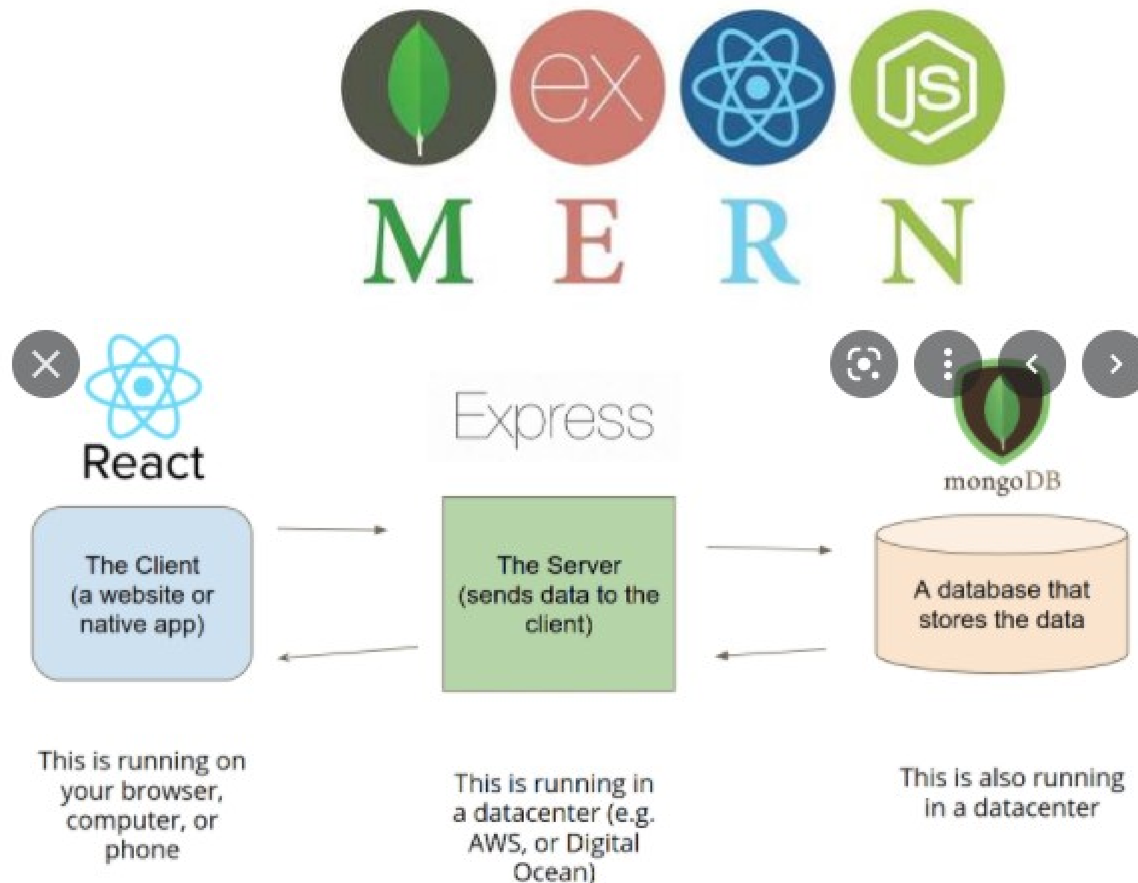


MEAN Stack:

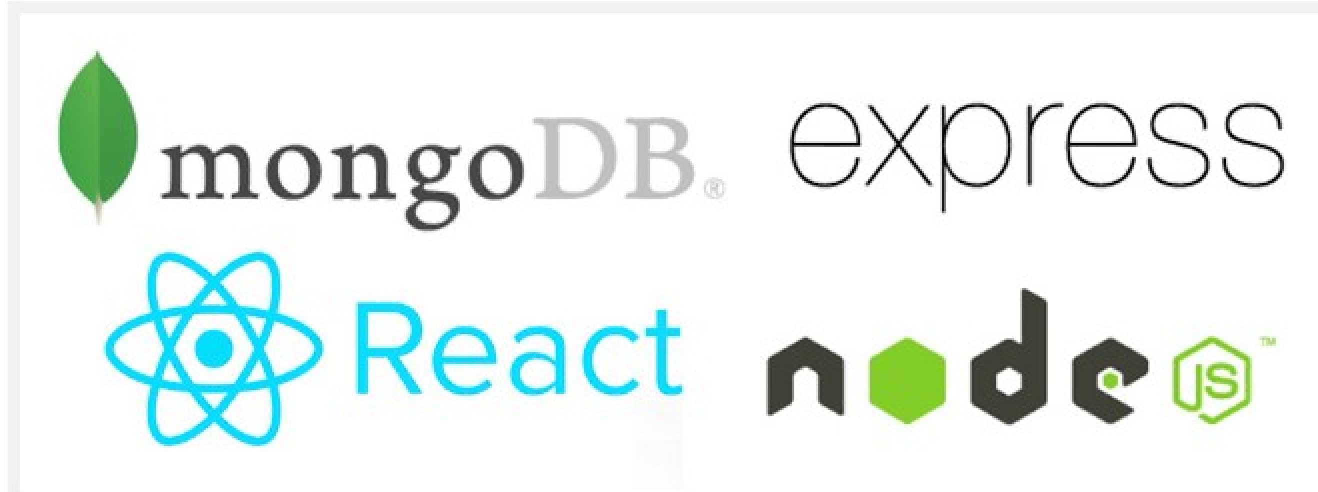
- It is one of the rapidly growing open-source development frameworks in stack technology.
- The availability of several plug-ins accelerates the development work,
- reduces system administration related tasks, and
- helps in the faster deployment of apps and websites.
- The complete web development cycle using JavaScript starting from client-side to server-side.
- Has a variety of plugin tools, testing tools for faster development and testing.

MERN Stack:

- MERN is also a full stack of web development, open-source technology.
- MERN stands for MongoDB, Express.js, React, and Node.js.
- This technology too can be used for making hybrid mobile and web apps.
- It is also a JavaScript-based framework for development.
- This stacking technology is growing in popularity because it offers an end-to-end runtime environment.



MERN Stack:



Benefits of MERN Stack:

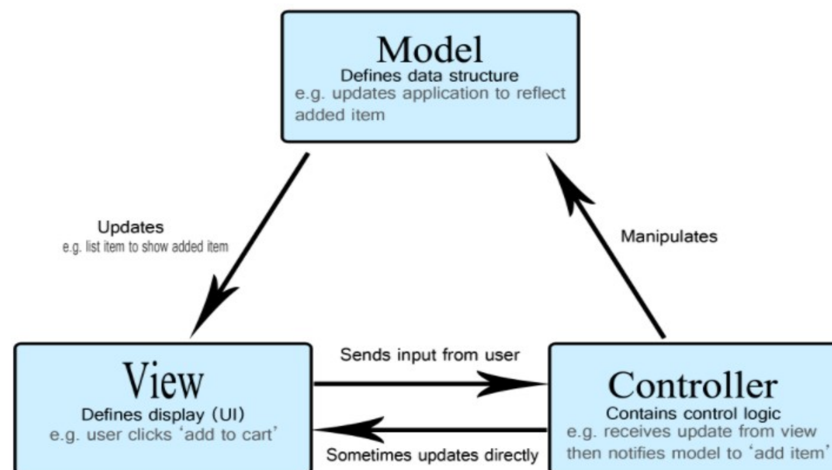
- A MERN stack developer uses JavaScript and JSON.
- Works in the Model View Controller architecture.
- There is a range of testing tool suites.
- Helps in the complete web development process, from front-end to back-end development.
- Is an open-source framework.

Similarities between MEAN & MERN:

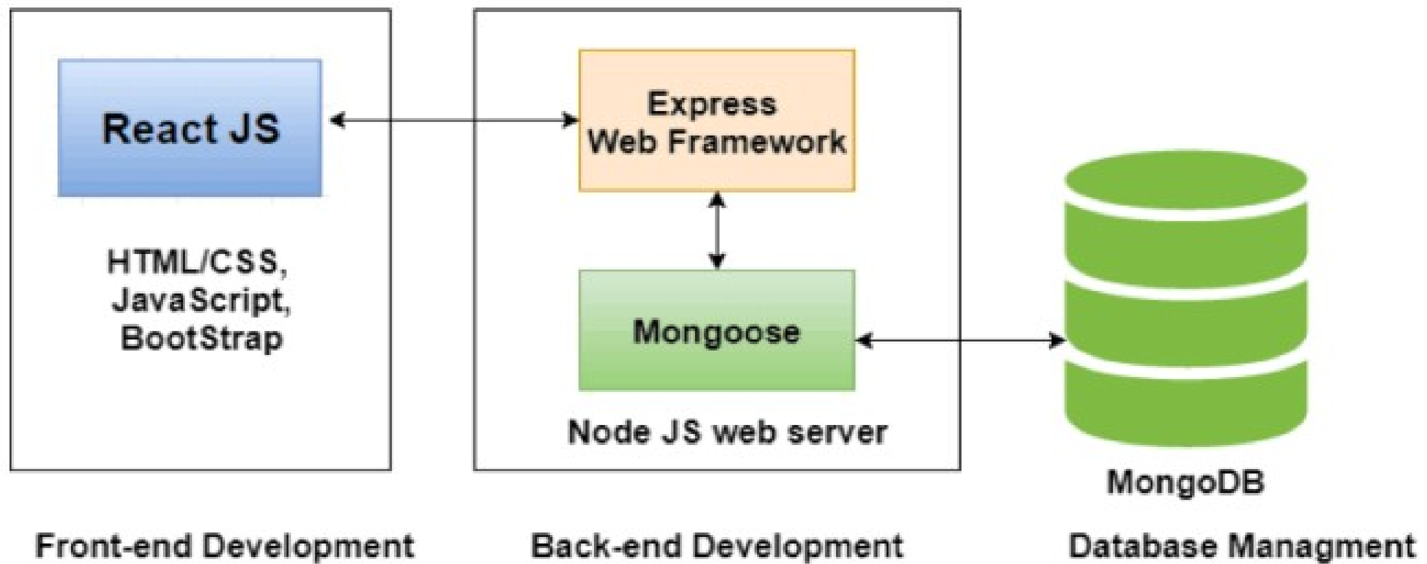
- Both are open-source,
- protect against XSS cross-site scripting,
- offer strong documentation support,
- helps in organizing the UI layer,
- supports MVC architecture,
- and have an extensive range of testing tools suite.
- The similarity between the two comes from the use of MongoDB, Express.js, and Node.js

MEAN is a better option for large-scale applications while MERN stack leads the race in the faster development of smaller applications.

To become a Full Stack Developer → Node.JS along with React or Angular + Database (MongoDB etc..)



MERN Stack



MERN Stack

MERN Stack:

MongoDB: **MongoDB is a NoSQL database component** of this full-stack and is used to store data of any type. It is an open-source, document-oriented database program. The data is stored in JSON (JavaScript Object Notation) format. MongoDB is a fast element that allows easy indexing of documents. It is a flexible component and can be integrated with document models such as tables, schemas, columns & SQL.

Express JS: Express JS is the web framework for Node JS. It helps in the creation of robust web applications and APIs. This component is responsible for the code reusability characteristic since it has a built-in router. It is asynchronous, single threaded, reliable, and fast when compared to its contenders. **IT is a Webserver built on Node.js**

React JS: React JS is the JavaScript library of MERN. It is used in the development of user interfaces. **React JS supports the creation of Single Page Applications (SPAs)** and mobile applications since it can deal with dynamic data. React JS's elements that make it a valuable part of MERN are virtual DOM, components, and JSX. Due to the presence of these elements, React JS is faster than all the other frameworks. A Front-end technology created by Facebook.

- React is not a full-fledged MVC framework. It is a JavaScript library for building user interfaces, so in some sense, it's the View part of the MVC.

Node JS: No**de JS is a Server Side JavaScript runtime environment;** unlike the others, this component of MERN allows the code to be run outside the browser, i.e., on the server. It is open-source and has fast execution speed since it was built on Google chrome's JavaScript Engine. Node JS is highly scalable and enables data streaming too.

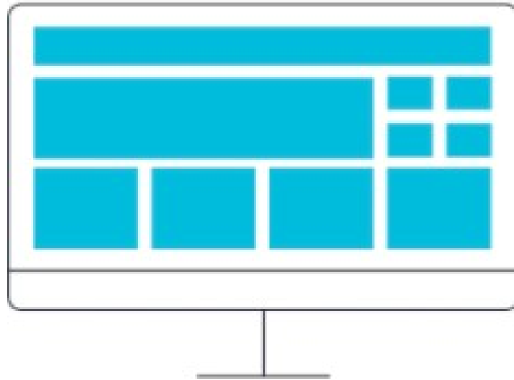
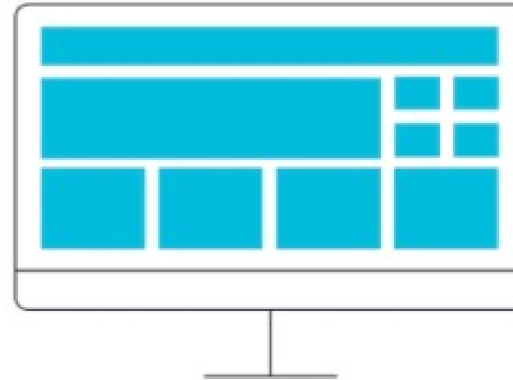
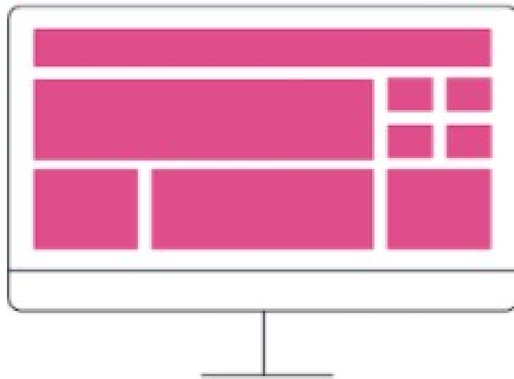
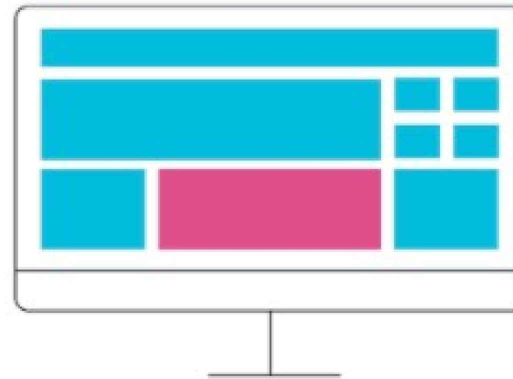
SPA (Single Page Application)

- An SPA is a web application paradigm that avoids fetching the contents of an entire web page from the server to display new contents.
- Single page application (SPA) is a single page (hence the name) where a lot of information stays the same and only a few pieces need to be updated at a time.
- The single page application is a web application or website that interacts with the user by dynamically rewriting the current page, rather than loading entire new pages from the server.
- This approach voids interruption of the user experience between successive pages, making the application behave more like a desktop application.
- On most websites there is a lot of repeating content. Some of it stays the same no matter where the user goes (headers, footers, logos, navigation bar, etc), some of it is constant in just a certain section (filter bars, banners), and there are many repeating layouts and templates (blogs, self-service, the google mail setup mentioned above).
- Some Single Page Application examples are like **Gmail, Google Maps, AirBNB, Netflix, Pinterest, Paypal, and many more.**
- For example, when you browse through your **email** you'll notice that not much changes during navigation - the sidebar and header remain untouched as you go through your inbox.

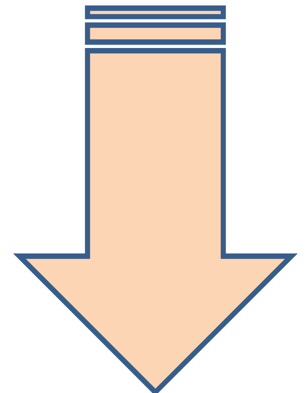
- The SPA only sends what you need with each click, and your browser renders that information. This is different to a traditional page load where the server re-renders a full page with every click you make and sends it to your browser.
- This piece by piece, client side method makes load time must faster for users and makes the amount of information a server has to send a lot less and a lot more cost efficient.

SPA (Single Page Application)

Traditional	SPA
Every request for new information gives you a new version of the whole page.	You request just the pieces you need.

VS Code & React



VS Code

- It is a code editor
- Other examples: Atom, Brackets, Sublime Text

Features:

- Has a built in Terminal
- Has built-in Source Control (Git)
- Has a built-in Debugger
- Has smart completions with Intellisense
- Is highly customizable
- Is fast
- Free
- Has lots of Powerful extensions to make your code editing easy and enjoyable.
- *Support for multiple programming languages: C#, Visual Basic, Java-Script, R, XML, Python, CSS*
- *GO, PERL*

- *Cross-Platform Support: (Windows, Linux and Mac*
- *Web support* → Comes with built-in support for Web applications. So web applications can be built and supported in VSC.
- **Hierarchy Structure:** The code files are located in files and folders. The required code files also have some files, which may be required for other complex projects. These files can be deleted as per convenience.
- **Multi-Projects:** Multiple projects containing multiple files/folders can be opened simultaneously. These projects/folders might or might not be related to each other.
-

VS Code

- It is a code editor
- Other examples: Atom, Brackets, Sublime Text

Features:

- Has a built in Terminal
- Has built-in Source Control (Git)
- Has a built-in Debugger
- Has smart completions with Intellisense
- Is highly customizable
- Is fast
- Has lotsof Powerful extensiosns to make your code editing easy and enjoyable.

VS code Extensions features:

Bracket Pair Colorizer → Bracket Pair Colorizer allows matching brackets to be identified with colors.

Highlight Matching Tag ➔ This extension highlights your opening and closing tags for you.

LiveServer → Launch a development server and run your code/page in the browser with live reloading on save!! Works with dynamic and static sites!

VS Code

VS code Extensions features contd..:

Prettier

Properly format your code with a single command. Highlight your text, open your Command Palette (discussed below) and choose Format Selection.

CSS Peek

This was inspired from the Brackets code editor, being able to peek at the CSS code from its associated HTML. You can also see the definition by hovering (hold Ctrl), or choose to Go To the Definition itself. A very useful extension when working in HTML.

Minify

Everyone should have a good Minifier. This is a good one to minify your JS, CSS, and HTML. Simply run your Command Palette and choose Minify!

Code Runner

Run your code right in Visual Studio Code and see the results in the Ouput. Highlight your text to only run specific code.

Regex Previewer

I no longer have to go to RegEx test sites to test my regular expressions.

VS Code

Shortcuts:

Command + /: Comment (and uncomment) out code.
**Command + ** : Split screen
Command + D: Select next occurrence of word (multi-replace)
Ctrl + ' : Show Terminal
Command + , : Show User Settings
Command + F: Find
Option + Up: Move line up
Option + Shift + Up : Copy the line up

Customizations

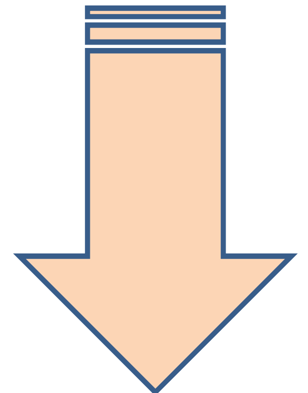
Font size: "editor.fontSize": 12,
Tab spacing: "editor.tabSize": 4,
Font family: "editor.fontFamily": "Menlo, Monaco, 'Courier New', monospace",
Word wrap: "editor.wordWrap": "off",

VS Code Terminal

- Visual Studio Code includes a full featured integrated terminal that provides integration with the editor to support features like links (Follow link, Open file in editor, Open folder in New window) and error detection and project build, run etc..
- *From the Terminal, required packages can be installed.*
- The integrated terminal can use various shells installed on your machine, with the default being pulled from your system defaults.
- *Example shells that can be used from the Terminal are:*
  - *Power Shell (ps or pwsh: default one)*
  - *Command Prompt*
  - *Java Script Debug Terminal*

  - *Supported shells:*
  - ***Linux/macOS:** bash, pwsh, zsh*
  - ***Windows:** pwsh*

Installation & Environment setup



React Installation & Environment setup

Steps:

1. Install NodeJS and npm (Node Package Manager)

- **NodeJS** is the platform needed for the ReactJS development.
- **NPM** is the package manager for the Node JavaScript platform. It puts modules in place so that node can find them, **and manages dependency conflicts intelligently.**
- The Node.js interpreter will be used to interpret and execute your javascript code.
- Node.js distribution comes as a binary installable for SunOS , Linux, Mac OS X, and Windows operating systems with the 32-bit (386) and 64-bit (amd64) x86 processor architectures.
- https://nodejs.org/en/ **-- install the latest version of Node.js (16.8.0)**
- **Check installation of node and npm by running command from power shell/command prompt:**

  node –v

  npm –v

  To install npm: from command prompt, or VS Code terminal command prompt, use command:

  npm install –g npm

2. Installing React:

we can install React in two ways:

1. Using Webpack and Babel
2. Using **create-react-app** command

React installation using ‘create-react-app’

Step 1: use command ‘npx create-react-app <appname> to install React app.

npx create-react-app reactapp1

Step 2: Delete all the source files

go to src folder and give command:

del *

Step 3: Add index.js and index.css files to the \src folder

> type nul >index.css

> type nul > index.js

Step 4: in index.js, add the following:

```
import React from 'react’;
import ReactDOM from 'react-dom’;
import './index.css';
```

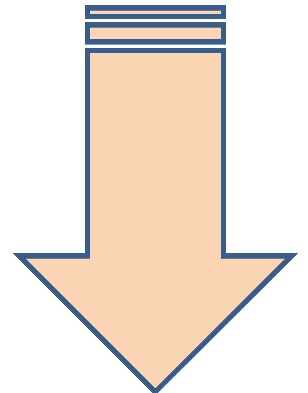
Step 5: run the project from command prompt /terminal using:

npm start

Step 6: use npm update to update the current app to latest version.

Use **npm install →** to install required node packages. (after deleting node_modules folder)

MERN Components, Libraries & Tools



codepen

MERN Components

MERN Components

1. React
2. Node.js
3. Express
4. Mongo DB

1. React:

- This is the main Component of the MERN stack.
- ReactJS is one of the most popular JavaScript front-end libraries which has a strong foundation and a large community.
- ReactJS is a **declarative**, **efficient**, and flexible **JavaScript library** for building reusable UI components. It is an open-source, component-based front end library which is responsible only for the view layer of the application. It was initially developed and maintained by Facebook and later used in its products like WhatsApp & Instagram.
- 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.
- **Example companies** using React are: Instagram, Facebook, Skype, Pinterest, Uber Eats, Bloomberg etc…
- ReactJS Tutorial – javatpoint

React

Features:

- ✓ **Open Source**

React Application:

- A React application is made of multiple components, each responsible for rendering a small, reusable piece of HTML.
- Components can be nested within other components to allow complex applications to be built out of simple building blocks.
- A component may also maintain an **internal state** – for example, a TabList component may store a variable corresponding to the currently open tab.
- The main objective of ReactJS is to develop User Interfaces (UI) that improves the speed of the apps. It uses virtual DOM (JavaScript object), which improves the performance of the app.
- The JavaScript virtual DOM is faster than the regular DOM. We can use ReactJS on the client and server-side as well as with other frameworks. It uses component and data patterns that improve readability and helps to maintain larger apps.

Creating a React Project:

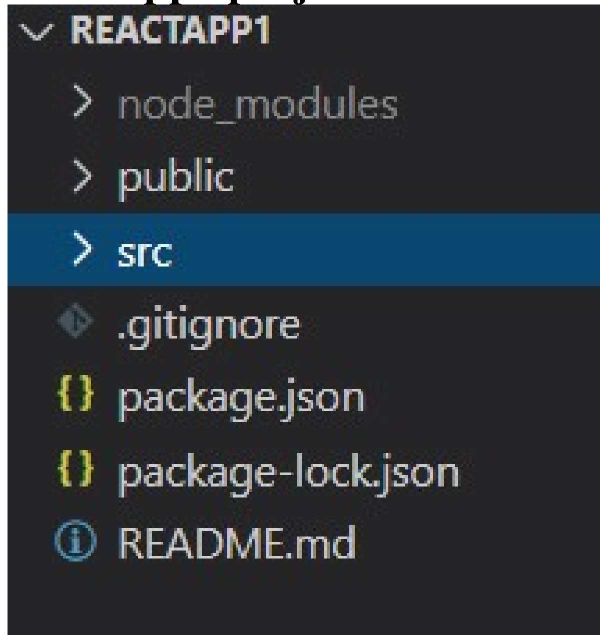
At command prompt: `npx create-react-app myapp`

Running: npm start

it can even produce a mobile application's UI using **React Native.**

React Project structure

reactapp1 project contains the following files and folders:



node_modules: All the "dependencies" and "devDependencies" required by our React app in node_modules.

This directory gets added to .gitignore so it does not really get uploaded/published as such.

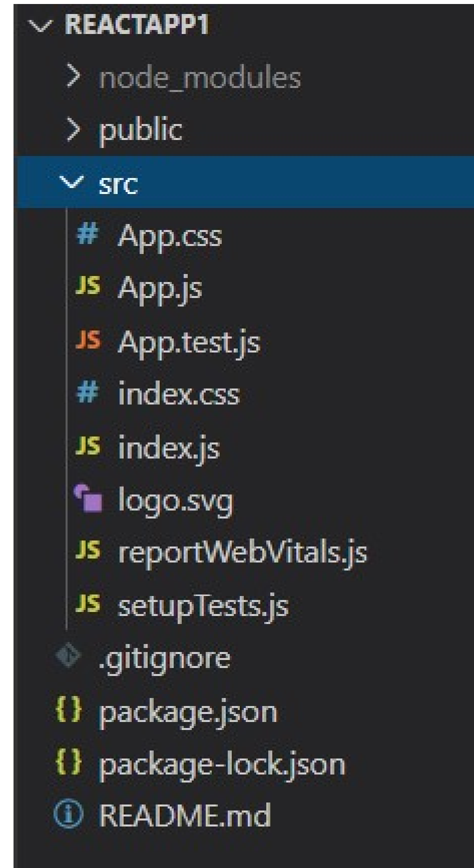
Public: static files are located in the public directory**. Files in this directory will retain the same name when deployed to production. Thus, they can be cached at the client-side and improve the overall download times.**

- **contains index.html →** main html file
- **manifest.json →** It's a Web App Manifest that describes your application (name, author, icon, and description) and it's used by e.g. mobile phones if a shortcut is added to the home screen
- **logos & icons**

src: All of the dynamic components will be located in the src.

React Project structure

src: All of the dynamic components will be located in the src.



App.js → Main JS component. Is the container of the app and serves as an application.

App.css → Main JS component css styles file

App.test.js → contains unit test code for the component.

Index.js → It is the default entry point of every react application. There are no changes in this file at all.

React Project structure

/assets :

- As the name suggests, all the static assets should reside here.
- Each asset should be registered and exported from the /index.js
- Thus, all assets will be accessible and imported from '/assets'
- This can include but not limited to images, logos, vector icons, fonts, etc.

/components:

- Only shared components used across features are placed here.
- All the components should be registered and exported from /index.js for a single access point.
- All the components should bear named export. This will avoid any conflicts.

React Project structure

package.json: The overall configuration for the React project is outlined in the package.json

Contains:

name - Represents the app name which was passed to **create-react-app.**

version - Shows the current version.

dependencies - List of all the required modules/versions for our app. By default, npm would install the most recent major version.

devDependencies - Lists all the modules/versions for running the app in a development environment.

scripts - List of all the aliases that can be used to access **react-scripts** commands in an efficient manner.

- For example, if we run **npm build** in the command line, it would run **"react-scripts build"** internally.

JSX (JavaScript XML)

• React is a declarative, efficient, and flexible JavaScript library for building user interfaces. But instead of using regular JavaScript, React code should be written in something called JSX.

• JSX stands for JavaScript XML. It is simply a syntax extension of JavaScript. ***It allows us to directly write HTML in React (within JavaScript code).*** It is easy to create a template using JSX in React, but it is not a simple template language instead it comes with the full power of JavaScript.

Let us see a sample JSX code:

```
const ele = <h1>This is sample JSX</h1>;
```

- The above code snippet somewhat looks like HTML and it also uses a JavaScript-like variable but is neither HTML nor JavaScript, it is JSX.
- **JSX is basically a syntax extension of regular JavaScript** and is used to create React elements. These elements are then rendered to the React DOM.

Why JSX?

- It is faster than normal JavaScript as it performs optimizations while translating to regular JavaScript.
- It makes it easier for us to create templates.
- Instead of separating the markup and logic in separated files, React uses *components* for this purpose.

JSX

Characteristics of JSX:

- JSX is not mandatory to use there are other ways to achieve the same thing but using JSX makes it easier to develop react application.
- JSX allows writing **expression in { }**. The expression can be any JS expression or React variable.
- To insert **a large block of HTML** we have to write it in a parenthesis i.e, **()**.
- JSX produces react elements.
- JSX follows XML rule.
- After compilation, JSX expressions become regular JavaScript function calls.
- JSX uses camelcase notation for naming HTML attributes. For example, tabindex in HTML is used as tabIndex in JSX.

JSX

Advantages of JSX:

- JSX makes it easier to write or add HTML in React.
- JSX can easily convert HTML tags to react elements.
- It is faster than regular JavaScript.
- JSX allows us to put HTML elements in DOM without using appendChild() or createElement() method.
- As JSX is an expression, we can use it inside of if statements and for loops, assign it to variables, accept it as arguments, or return it from functions.
- JSX prevents XSS (cross-site-scripting) attacks popularly known as injection attacks.
- It is type-safe, and most of the errors can be found at compilation time.

Disadvantages of JSX:

- JSX throws an error if the HTML is not correct.
- In JSX HTML code must be wrapped in one top-level element otherwise it will give an error.
- If HTML elements are not properly closed JSX will give an error.

JSX

Using JavaScript expressions in JSX: In React we are allowed to use normal JavaScript expressions with JSX.

- To embed any JavaScript expression in a piece of code written in JSX we will have to **wrap that expression in curly braces {}.**

Example: .js file

```
import React from 'react';
import ReactDOM from 'react-dom';

const name = "Students";

const element = <h1>Hello,
{ name }.Welcome to BEC.< /h1>;

ReactDOM.render(
    element,
    document.getElementById("root")
);
```

ES5 & ES6

- ECMAScript is a trademarked scripting language specification that is defined by ECMA International. It was created to standardize JavaScript.
- The ES scripting language has many implementations, and the popular one is JavaScript. Generally, ECMAScript is used for client-side scripting of the World Wide Web.
- **ES5** is an abbreviation of ECMAScript 5 and also known as ECMAScript 2009.
- The sixth edition of the ECMAScript standard is **ES6 or ECMAScript 6**. It is also known as ECMAScript 2015. ES6 is a major enhancement in the JavaScript language that allows us to write programs for complex applications.
- Although ES5 and ES6 have some similarities in their nature, there are also so many differences between them.
- React uses ES6, and you should be familiar with some of the new features like:
  - Classes
  - Arrow Functions
  - Variables (let, const, var)

ES5 & ES6

•

Based on	ES5	ES6
Definition	ES5 is the fifth edition of the ECMAScript (a trademarked scripting language specification defined by ECMA International)	ES6 is the sixth edition of the ECMAScript (a trademarked scripting language specification defined by ECMA International).
Release	It was introduced in 2009.	It was introduced in 2015.
Data-types	ES5 supports primitive data types that are **string, number, boolean, null,** and **undefined**.	In ES6, there are some additions to JavaScript data types. It introduced a new primitive data type **'symbol'** for supporting unique values.
Defining Variables	In ES5, we could only define the variables by using the **var** keyword.	In ES6, there are two new ways to define variables that are **let** and **const**.
Performance	As ES5 is prior to ES6, there is a non-presence of some features, so it has a lower performance than ES6.	Because of new features and the shorthand storage implementation ES6 has a higher performance than ES5.
Support	A wide range of communities supports it.	It also has a lot of community support, but it is lesser than ES5.
Object Manipulation	ES5 is time-consuming than ES6.	Due to destructuring and speed operators, object manipulation can be processed more smoothly in ES6.
Arrow Functions	In ES5, both **function** and **return** keywords are used to define a function.	An arrow function is a new feature introduced in ES6 by which we don't require the **function** keyword to define the function.
Loops	In ES5, there is a use of **for** loop to iterate over elements.	ES6 introduced the concept of **for...of** loop to perform an iteration over the values of the iterable objects.

ES6 - class

classes:

```
class Car {
  constructor(name) {
    this.brand = name;
  }
}
mycar = new Car("Ford");
```

Methods:

```
class Car {
  constructor(name) {
    this.brand = name;
  }

  present() {
    return 'I have a ' + this.brand;
  }
}

mycar = new Car("Ford");
mycar.present();
```

ES6 class example

```
<!DOCTYPE html>
<html>
<body>
<script>
class Car {
  constructor(name) {
    this.brand = name;
  }
  present() {
    return 'I have a ' + this.brand;
  } }
class Model extends Car {
  constructor(name, mod) {
    super(name);
    this.model = mod;
  }
  show() {
    return this.present() + ', it is a ' + this.model
  } }
mycar = new Model("Ford", "Mustang");
document.write(mycar.show());
</script></body></html>
```

ES6 Arrow Functions

- Arrow functions allow us to write shorter function syntax
- Arrow functions were introduced in ES6.

Function:

```
hello = function() {
  return "Hello World!";
}
```

Can be written as:

```
hello = () => {
  return "Hello World!";
}
```

ES6 Arrow Functions with parameters

hello = (val) => "Hello " + val;

Example:

```
<!DOCTYPE html>
<html>
<body>
<h1>Arrow Function</h1>
<p>A demonstration of an arrow function in one line, with parameters.</p>
<p id="demo"></p>
<script>
hello = (val) => "Hello " + val;
document.getElementById("demo").innerHTML = hello("World");
</script>
</body>
</html>
```

React render()

React's goal is in many ways to render HTML in a web page.

React renders html to the webpage using ReactDOM.render()

Render() function:

- The ReactDOM.render() function takes two arguments, HTML code and an HTML element.
- The purpose of the function is to display the specified HTML code inside the specified HTML element.

Example:

Index.html:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport"
      content="width=device-width, initial-scale=1" />
    <title>React App</title>
  </head>  <body>
    <div id="root"></div>


  </body></html>
```

index.js:

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(<p>Hello</p>,document.getElementById('root'));
```

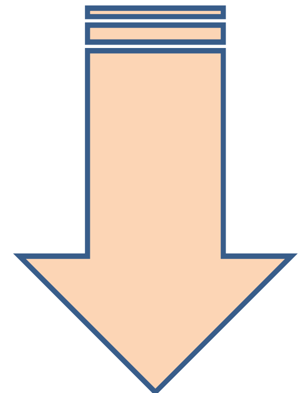
Using browser in VS Code

Step 1: First pick 'View' → 'Layout Editor' → 'Two Columns' layout

Step 2: press Ctrl + Shift + p to open Command Palette. Search for the ' Simple Browser' and add.

In the browser give url: **http://localhost:3000 to get rendered page**

React.JS Components



Component

- Class components
- Function components

Class Component

- 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.
- But **render()** is one that *must* be present, otherwise the component will have no screen presence. The render() function is supposed to return an element (which can be either a native HT ML element such as a <div> or an instance of another React component).

```
render() {
    return (
    <div title="Outer div">
    <h1>{message}</h1>
    </div>
);
```

Class Component – Example 1

Creating a class component is pretty simple; just define a class that extends Component and has a render function.

// MyComponent.js

```
import React, { Component } from 'react‘;
class MyComponent extends Component {
  render() {
    return (
      <div>This is my component.</div>
    );
  }
}
export default MyComponent;
```

```
// MyOtherComponent.js
import React, { Component } from 'react';
import MyComponent from
'./MyComponent';

class MyOtherComponent extends
Component {
  render() {
    return (
      <div>
        <div>This is my other
component.</div>
        <MyComponent />
      </div>
    );
  }
}

export default MyOtherComponent;
```

Class Component – Example 2

```
 class TestComponent extends Component {
render()
{
        return ( <div>This is my TestComponent.</div> ); }
}
export default TestComponent;
```

```
import TestComponent  from './TestComponent ';
class OtherComponent extends Component {
render() {
return (
<div>
<div>Here is other component.</div>
<TestComponent  />
</div>
);
}
}
export default OtherComponent;
```

Class Component – Example 3

```
 import React from "react“;
class Sample extends React.Component {
render() {
		return <h1>A Computer Science Portal
For Geeks</h1>;
}
}
class App extends React.Component {
render() {
		return <Sample />;
}
}
export default App;
```

Function Component– Example 1

Welcome.js:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
function Welcome() {
   const greeting = 'Hello Function Component!‘;
   return <h1>{greeting}</h1>;
 }
 export default Welcome;
```

In **Index.js:**

In **Index.js:**

```
import Welcome from './Welcome‘
const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <Welcome />
  </React.StrictMode>
);
```

Component

- Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML via a render() function.
- Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.
- Components are the fundamental unit of a clientside application written in React.
- Components come in two types, Class components and Function components.

Class Component:

Example 1:

```
class Welcome extends React.Component {
  render() {
    return <h1> Hello World </h1>;
  }
}
```

Example 2:

```
class Car extends React.Component {
  render() {
    return <h2>Hi, I am a Car!</h2>;
  }
}
```

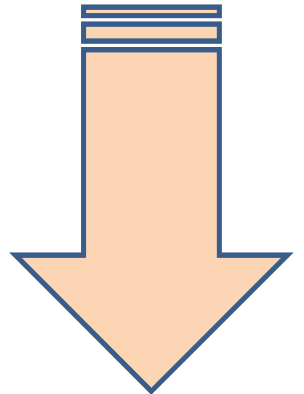
- When creating a React component, the **component's name must start with an upper case letter.**
- The component has to include the **extends React.Component** statement, this statement creates an inheritance to React.Component, and gives your component access to React.Component's functions.
- The component also requires a **render() method**, this method returns HTML.

Components

- you should prefer to use functional components over class components as they are simpler,
  - testable,
  - better in terms of performance,
  - requires less code,
  - and allow the use of best practices.
- But class components have their own advantages.

- Use named exports to export multiple functions in React, e.g. export function A() {} and export function B() {}   (in defining the function use export keyword)
- The exported components can be imported by using a named import as import {A, B} from './another-file'. You can have as many named exports as necessary in a single file.
-

props



props

- The term 'props' is an abbreviation for 'properties' which refers to the properties of an object.
- It is an object which stores the value of attributes of a tag and work similar to the HTML attributes.
- It gives a way *to pass data from one component to other components to get dynamic and unique outputs.*
- It is similar to function arguments.
- Props are passed to the component in the same way as arguments passed in a function.
- *Props are **immutable*** so we cannot modify the props from inside the component.
- When you need immutable data in the component, you have to add props to **reactDom.render()** method in the **main.js/index.js** file as an attribute (in the component tag) of your ReactJS project and used it inside the component in which you need using **this.props.<PropName>**
- Websites built with React like ***Facebook, Twitter, and Netflix*** use the same design patterns across many sections that just have different data. One of the main ways developers can achieve this functionality is by using props.

Default Props: You can also set default property values directly on the component constructor instead of adding it to the **reactDom.render()** element.

```
ComponentName.defaultProps = {
msg: "World!"
}
```

```
Ex:  App.defaultProps = {
msg: "World!"
}
```

props

- The props enable the component to access customized data, values, and pieces of information that the inputs hold.
- The term 'props' is an abbreviation for 'properties' which refers to the properties of an object.

Props – example 1

App.js:

```
 import React, { Component } from 'react';
class App extends React.Component {
  render() {
    return (  <div>  <h1> Welcome to { this.props.name } </h1>
         <p> <h4> BEC Bapatla. </h4> </p>
       </div>
    );    }  }
export default App;
```

Index.js:

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';

ReactDOM.render(<App name = “myappcomp”/>, document.getElementById('app'));
```

Props – example 2

propex1.js

```
import React, { Component } from 'react';
class PropEx1 extends React.Component {
render() {
return (
<div>
<h2> Hello { this.props.msg } </h2>
<p> Have a Great day. </p>
</div>
);
}
}

PropEx1.defaultProps = {
   msg: "World!"
   }
export default PropEx1;
```

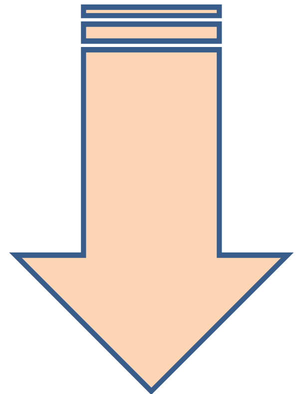
Index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client‘;
import PropEx1 from './propex1‘;

const root =
ReactDOM.createRoot(document.getElementById('
root'));
root.render(
  <React.StrictMode>
      <PropEx1 msg="BEC Students" />
      <PropEx1 />
  </React.StrictMode>
```

Output

Hello BEC Students
Have a Great day.
Hello World!
Have a Great day.

State



State

- **A State is an object** that stores the values of properties belonging to a component that could change over a period of time either by event handlers, server responses, or prop changes.
- React components has a built in state object.
- Every time the state of an object changes, React re-renders the component to the browser.

Example: Time Display in a page

- The state object can store multiple properties
- **this.setState()** is used to change the value of the **state** object.

- ~~Use **'rce'** template to add skeleton code in a .js file~~
- ~~Use **'rconst'** template to add constructor snippet.~~

State object creation

```
class Car extends React.Component {
 constructor(props) {
   super(props);
   this.state = {brand: "Ford"};
 }
 render() {
   return (
     <div>
       <h1>My Car</h1>
     </div>
   );
 }
}
```

State object with multiple properties

```
class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  render() {
    return (
      <div>
        <h1>My Car</h1>
      </div>
    );
  }
```

Changing state object

```
class Car extends React.Component {
 constructor(props) {
  super(props);
  this.state = {
   brand: "Ford",
   model: "Mustang",
   color: "red",
   year: 1964
  };
 }
 changeColor = () => {
  this.setState({color: "blue"});
 }
```

```
render() {
  return (
   <div>
    <h1>My {this.state.brand}</h1>
    <p>
     It is a {this.state.color}
     {this.state.model}
     from {this.state.year}.
    </p>
    <button
     type="button"
     onClick={this.changeColor}
    >Change color</button>
   </div>
  );
 }
```

The setState() Method

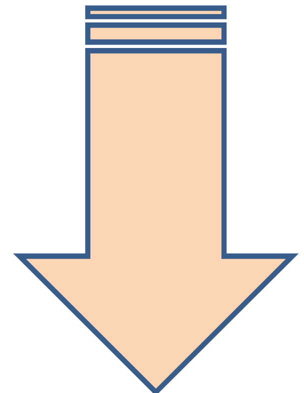
- State can be updated in response to event handlers, server responses, or prop changes. This is done using the **setState()** method.
- The **setState()** method enqueues all of the updates made to the component state and instructs React to re-render the component and its children with the updated state.

props vs state

area	Props	State
Usecase	Props are used to pass data and event handlers to its child components	State is used to store the data of the components that has to be rendered to the view
Mutability	Props are immutable – once set, props cannot be changed	State holds the data and can change over time
Component	Props can be used in both functional and class components	State can only be used in class components
Updation	Props are set by the parent component for the children components	State is generally updated by the event handlers.
ownership	props are immutable and owned by a component's parent	state is owned by the component. this.state is private to the component

Data binding in React.js

(one-way, two-way)



Data binding

- Data Binding is the process of connecting the view element or user interface, with the data which populates it.
- In **ReactJS**, components are rendered to the user interface and the component's logic contains the data to be displayed in the view(UI). The connection between the data to be displayed in the view and the component's logic is called data binding in ReactJS.
- **One-way Data Binding:** ReactJS uses one-way data binding. In one-way data binding one of the following conditions can be followed:
  - **Component to View:** Any change in component data would get reflected in the view.
  - **View to Component:** Any change in View would get reflected in the component's data.
-

Component to View Data binding

Any change in component data would get reflected in the view.

Example:

```
import React, { Component } from 'react';
class App extends Component {
  constructor() {
    super();
    this.state = {
      subject: "ReactJS"
    };
  }
  render() {
    return (
      <div style={{ textAlign: "center" }}>
        <h4 style={{ color: "#68cf48" }}>BEC</h4>
        <p><b>{this.state.subject}</b> Bapatla</p>

      </div>
    )  } }
export default App;
```

View to Component Data binding

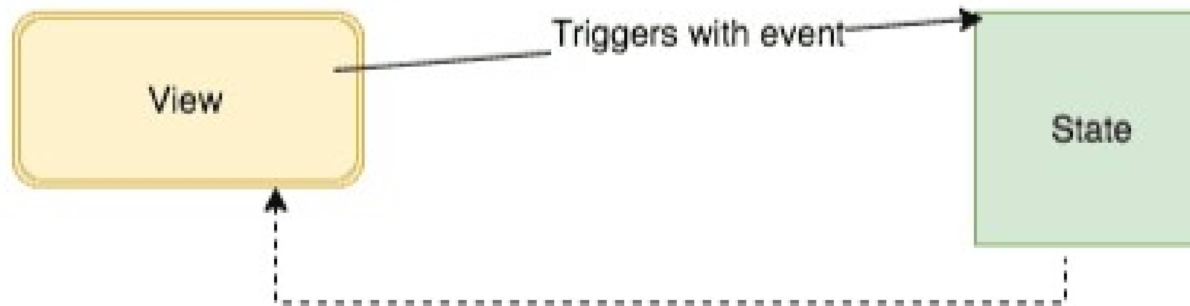
Any change in View would get reflected in the component's data.

```
import React, { Component } from 'react';
class App extends Component {
  constructor() {
    super();
    this.state = {
      subject: ""
    };  }

  handleChange = event => {
    this.setState({
      subject: event.target.value
    })  }

  render() {
    return (
      <div>
        <h4 style={{ color: "#68cf48" }}>BEC</h4>
        <input placeholder="Enter Subject"
        onChange={this.handleChange}></input>
        <p><b>{this.state.subject}</b> Bapatla</p>
      </div>
    )  } }
export default App;
```

Two Way Data binding

Two way data binding means

- The data we changed in the view will be updated in the state and the data in the state will be refected in the view automatically.



Two Way Data binding

Example:

```
class Twoway extends React.Component{
 state = {
    name:"reactgo"
 }
 handleChange = (e) =>{
  this.setState({
     name: e.target.value
  })  }
 render(){
  return(
    <div>
     <h1>{this.state.name}</h1>
     <input type="text"
      onChange={this.handleChange}
      value={this.state.name} />
    </div>
    )   }}
export default Twoway;
```

How data is Rendered in UI

- To render anything on the screen, we use the ReactDOM.render method in React.
- It has the following syntax:

  ReactDOM.render(element, container[, callback])

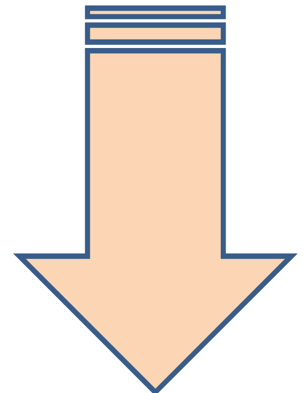
Here:

- **element** → can be any HTML element, JSX or a component that returns JSX
- **container** → is the element on the UI inside which we want to render the data
- **callback** → is the optional function we can pass which gets called once something is rendered or re-rendered on the screen

Rendering using JSX if large content need to be rendered:

```
import React from "react";
import ReactDOM from "react-dom“;
const rootElement = document.getElementById("root");
const content = (
  <div>
    <h1>Welcome to React!</h1>
    <p>React is awesome.</p>
  </div>
);
ReactDOM.render(content, rootElement);
```

Component Lifecycle



Component Lifecycle

- lifecycle of a component can be defined as the series of methods that are invoked in different stages of the component's existence.
- 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.
- Each component in React has a lifecycle which you can monitor and manipulate during **its Four main stages.**
  - The three phases are: Initializing, Mounting, Updating, and Unmounting.
- **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.

Component Lifecycle

- Mounting → Updating→ Unmounting stages

Mounting stage methods:

- constructor()
- getDerivedStateFromProps()
- render()
- componentDidMount()

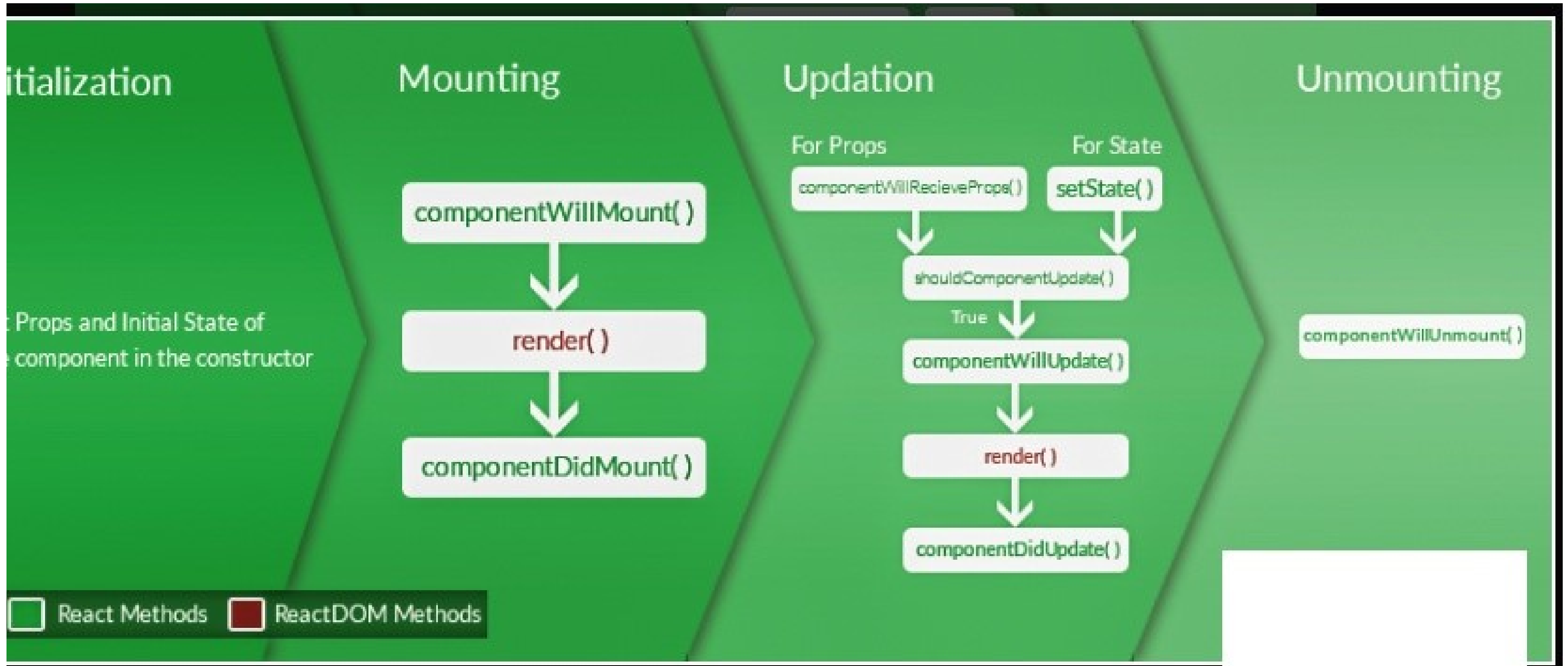
Updating Stage:

- getDerivedStateFromProps()
- shouldComponentUpdate()
- render()
- getSnapshotBeforeUpdate()
- componentDidUpdate()

Unmounting stage:

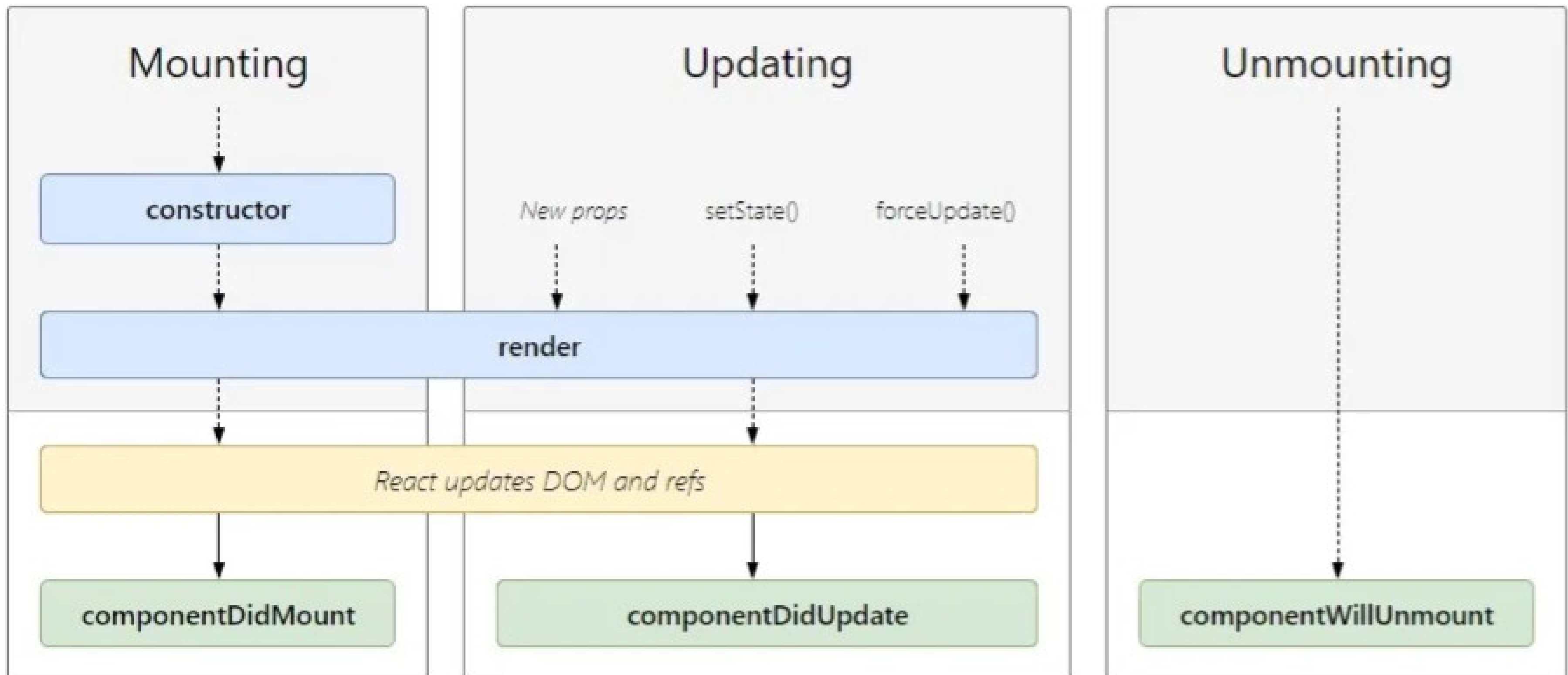
componentWillUnmount()

Component Lifecycle



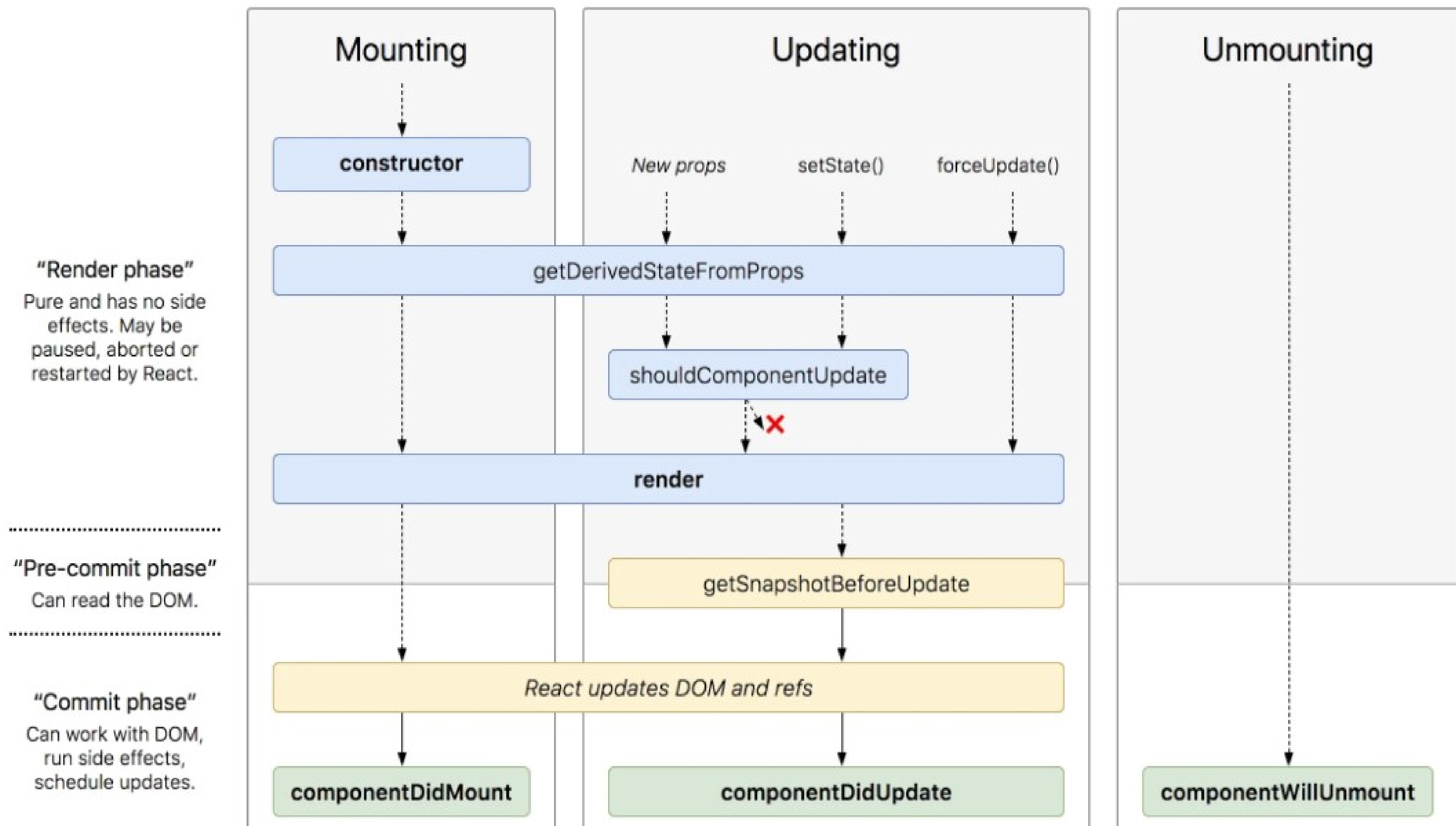
Component Lifecycle

- lifecycle of a component can be defined as the series of methods that are invoked in different stages of the component's existence.
- Each component in React has a lifecycle which you can monitor and manipulate during **its three main phases.**
  - The three phases are: Mounting, Updating, and Unmounting.

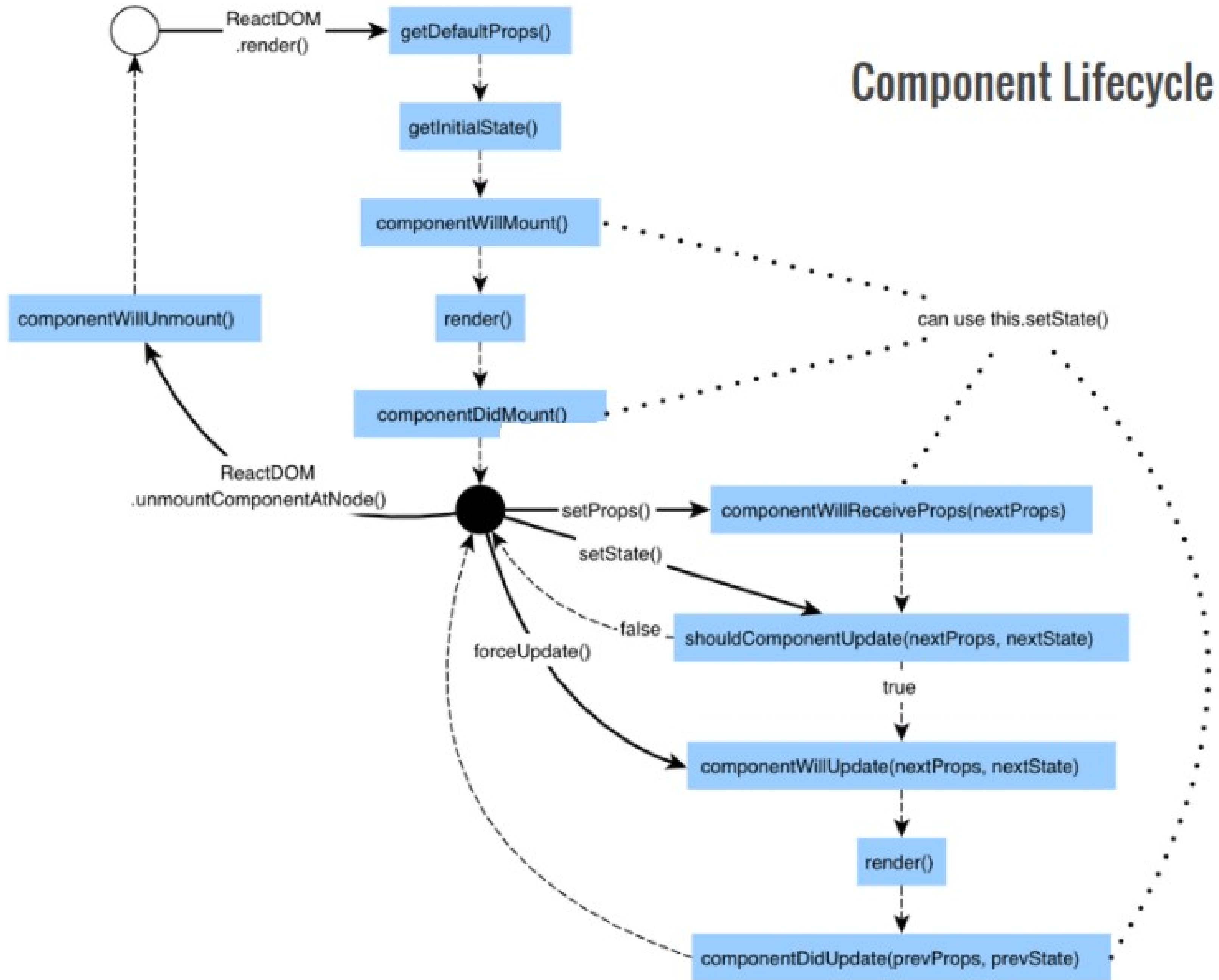


Component Lifecycle

- lifecycle of a component can be defined as the series of methods that are invoked in different stages of the component's existence.
- Each component in React has a lifecycle which you can monitor and manipulate during **its three main phases.**
  - The three phases are: Mounting, Updating, and Unmounting.



Component Lifecycle



Component Lifecycle - Initialization

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

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

Component Lifecycle – Mounting stage

- 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

Component Lifecycle

Mounting phase: Mounting means putting elements into the DOM.

React has **four built-in methods** that gets called, in this order, when mounting a component:

1. constructor()
2. getDerivedStateFromProps()
3. render()
4. componentDidMount()

Updating phase: five built-in methods

- A component is updated whenever there is a change in the component's state or props.
- React has **five built-in methods** that gets called, in this order, when a component is updated:

1. getDerivedStateFromProps()
2. shouldComponentUpdate()
3. render()
4. getSnapshotBeforeUpdate()
5. componentDidUpdate()

Component Lifecycle – unmounting phase

Unmounting phase: one built in method

- The next phase in the lifecycle is when a component is removed from the DOM, or unmounting as React likes to call it.
- React has only one built-in method that gets called when a component is unmounted. i.e **componentWillUnmount()**

componentWillUnmount():

- The componentWillUnmount method is called when the component is about to be removed from the DOM.

Component Lifecycle

- Each component in React has a lifecycle which you can monitor and manipulate during **its three main phases.**
  - The three phases are: Mounting, Updating, and Unmounting.

Mounting: Mounting means putting elements into the DOM.

React has **four built-in methods** that gets called, in this order, when mounting a component:

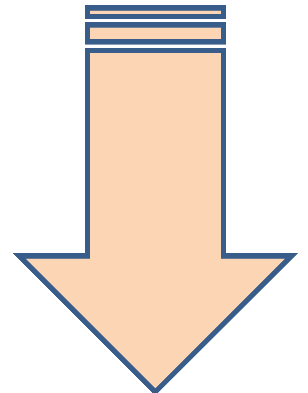
1. constructor()
2. getDerivedStateFromProps()
3. render()
4. componentDidMount()

The render() method is required and will always be called, the others are optional and will be called if you define them.

constructor( ):

- The **constructor()** method is called before anything else, when the component is initiated, and it is the natural place to set up the initial state and other initial values.
- The **constructor()** method is called with the props, as arguments, and you should always start by calling the super(props) before anything else, this will initiate the parent's constructor method and allows the component to inherit methods from its parent (React.Component).

Mounting phase



Mounting – constructor( ) method

Example:

```
class Header extends React.Component {
  constructor(props) {
    super(props);  //calling base class
    this.state = {favoritecolor: "red"};   //setting initial state
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

ReactDOM.render(<Header />, document.getElementById('root'));
```

Mounting - getDerivedStateFromProps( )

getDerivedStateFromProps( ):

- The getDerivedStateFromProps() method is called right before rendering the element(s) in the DOM.
- This is the natural place to set the state object based on the initial props.
- It takes state as an argument, and returns an object with changes to the state.
- The getDerivedStateFromProps method is called right before the render method

The example below starts with the favorite color being "red", but the getDerivedStateFromProps() method updates the favorite color based on the favcol attribute:

Mounting - getDerivedStateFromProps( )

The example below starts with the favorite color being "red", but the getDerivedStateFromProps() method updates the favorite color based on the favcol attribute:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol };
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );  }}
ReactDOM.render(<Header favcol="yellow"/>, document.getElementById('root'));
```

Mounting – render() method

- The render() method is required, and is the method that actually outputs the HTML to the DOM.

A simple component with a simple render() method:

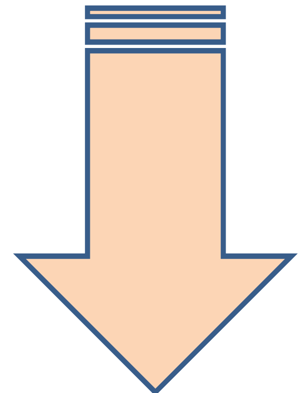
```
class Header extends React.Component {
  render() {
    return (
      <h1>This is the content of the Header component</h1>
    );
  }
}

ReactDOM.render(<Header />, document.getElementById('root'));
```

Mounting – componentDidMount( ) method

- The componentDidMount() method is called after the component is rendered.
- This is where you run statements that requires that the component is already placed in the DOM .
- At first my favorite color is red, but give me a second, and it is yellow instead:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)  }
  render() {
    return (<h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );  }}
ReactDOM.render(<Header />, document.getElementById('root'));
```

Update phase



Updating Phase

- **Updation**: React is a JS library that helps create Active web pages easily. Now active web pages are specific pages that behave according to their user. For example, let's take the GeeksforGeeks {IDE} webpage, the webpage acts differently with each user. User A might write some code in C in the Light Theme while another User may write a Python code in the Dark Theme all at the same time. This dynamic behavior that partially depends upon the user itself makes the webpage an Active webpage. Now how can this be related to Updation? Updation is the phase where the states and props of a component are updated followed by some user events such as clicking, pressing a key on the keyboard, 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.

Updating Phase

- **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.

Updating Phase

- The next phase in the lifecycle is when a component is updated.
- A component is updated whenever there is a change in the component's state or props.
- React has **five built-in methods** that gets called, in this order, when a component is updated:

1. getDerivedStateFromProps()
2. shouldComponentUpdate()
3. render()
4. getSnapshotBeforeUpdate()
5. componentDidUpdate()

- The render() method is required and will always be called, the others are optional and will be called if you define them.

Updating Phase - getDerivedStateFromProps()

1. getDerivedStateFromProps()

- Also at updates the getDerivedStateFromProps method is called.
- This is the first method that is called when a component gets updated.
- This is still the natural place to set the state object based on the initial props.

Updating Phase - getDerivedStateFromProps()

- The example below has a button that changes the favorite color to blue, but since the getDerivedStateFromProps() method is called, which updates the state with the color from the favcol attribute, the favorite color is still rendered as yellow:

```
class Header extends React.Component {
 constructor(props) {
  super(props);    this.state = {favoritecolor: "red"};
 }
 static getDerivedStateFromProps(props, state) {
  return {favoritecolor: props.favcol };  }
 changeColor = () => {
  this.setState({favoritecolor: "blue"});  }
 render() {    return (<div><h1>My Favorite Color is {this.state.favoritecolor}</h1>
    <button type="button" onClick={this.changeColor}>Change color</button>
    </div>
  );  }}
ReactDOM.render(<Header favcol="yellow"/>, document.getElementById('root'));
```

Updating Phase - shouldComponentUpdate()

2. shouldComponentUpdate()

- In the shouldComponentUpdate() method you can return a Boolean value that specifies whether React should continue with the rendering or not. The default value is true.

Stop the component from rendering at any update:

Updating Phase - shouldComponentUpdate()

Stop the component from rendering at any update:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};  }
  shouldComponentUpdate() {
    return false;  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});  }
  render() {
    return (
      <div>      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
      </div>    );  }}
ReactDOM.render(<Header />, document.getElementById('root'));
```

Updating Phase - render()

render():

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );  }}
ReactDOM.render(<Header />, document.getElementById('root'));
```

Updating Phase - getSnapshotBeforeUpdate()

1. getSnapshotBeforeUpdate():
2. n the getSnapshotBeforeUpdate() method you have access to the props and state before the update, meaning that even after the update, you can check what the values were before the update.

3. If the getSnapshotBeforeUpdate() method is present, you should also include the componentDidUpdate() method, otherwise you will get an error.

Updating Phase - getSnapshotBeforeUpdate()

1. getSnapshotBeforeUpdate():
2. n the getSnapshotBeforeUpdate() method you have access to the props and state before the update, meaning that even after the update, you can check what the values were before the update.

3. If the getSnapshotBeforeUpdate() method is present, you should also include the componentDidUpdate() method, otherwise you will get an error.

Updating Phase - getSnapshotBeforeUpdate()

```
class Header extends React.Component {
 constructor(props) {
  super(props);
  this.state = {favoritecolor: "red"};   }
 componentDidMount() {
  setTimeout(() => {
   this.setState({favoritecolor: "yellow"})
  }, 1000)  }
 getSnapshotBeforeUpdate(prevProps, prevState) {
  document.getElementById("div1").innerHTML =
  "Before the update, the favorite was " + prevState.favoritecolor;   }
 componentDidUpdate() {
  document.getElementById("div2").innerHTML =
  "The updated favorite is " + this.state.favoritecolor;
 }
```

Contd….

Updating Phase - getSnapshotBeforeUpdate() contd….

```
render() {
  return (
    <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <div id="div1"></div>
      <div id="div2"></div>
    </div>
  );
}
```

Updating Phase - componentDidUpdate()

The componentDidUpdate method is called after the component is updated in the DOM.

Example:

The example below might seem complicated, but all it does is this:

When the component is mounting it is rendered with the favorite color "red".

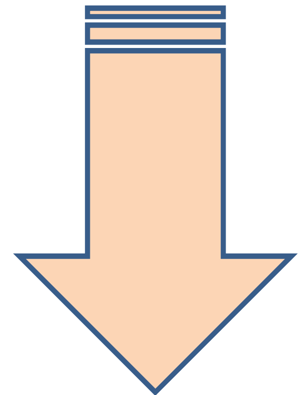
When the component has been mounted, a timer changes the state, and the color becomes "yellow".

This action triggers the update phase, and since this component has a componentDidUpdate method, this method is executed and writes a message in the empty DIV element:

Updating Phase - componentDidUpdate()

```
class Header extends React.Component {
  constructor(props) {    super(props);    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})    }, 1000)  }
  componentDidUpdate() {
    document.getElementById("mydiv").innerHTML =
    "The updated favorite is " + this.state.favoritecolor;
  }
  render() {
    return (        <div>        <h1>My Favorite Color is {this.state.favoritecolor}</h1>
<div id="mydiv"></div>      </div>    );  }
}
ReactDOM.render(<Header />, document.getElementById('root'));
```

Unmounting phase



Unmounting phase

- **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.

Unmounting phase - componentWillUnmount() method

- The next phase in the lifecycle is when a component is removed from the DOM, or unmounting as React likes to call it.
- React has only one built-in method that gets called when a component is unmounted.

componentWillUnmount():

- The componentWillUnmount method is called when the component is about to be removed from the DOM.

Unmounting phase - componentWillUnmount() method

Example:

```
class Container extends React.Component {
  constructor(props) {    super(props);    this.state = {show: true};
  }
  delHeader = () => {    this.setState({show: false});
  }
  render() {
    let myheader;
    if (this.state.show) {
      myheader = <Child />;
    };
    return (<div> {myheader}  <button type="button" onClick={this.delHeader}>Delete
Header</button>      </div>    );  }  }
class Child extends React.Component {
  componentWillUnmount() {
    alert("The component named Header is about to be unmounted.");
  }
  render() {    return (      <h1>Hello World!</h1>    );  }}
ReactDOM.render(<Container />, document.getElementById('root'));
```

Example

```
import React from 'react';
import ReactDOM from 'react-dom';
class Test extends React.Component {
   constructor(props)
   {
      super(props);
      this.state = { hello : "World!" };
   }
   componentWillMount()
   {
      console.log("componentWillMount()");
   }
   componentDidMount()
   {
      console.log("componentDidMount()");
   }
```

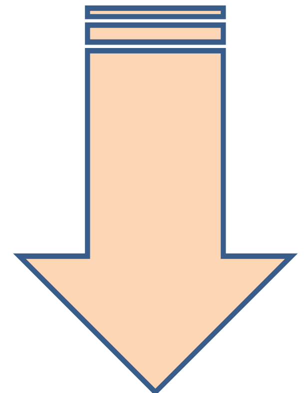
Example

```
changeState()
  {
    this.setState({ hello : "Geek!" });
  }
  render()
  {
    return (
      <div>
      <h1>GeeksForGeeks.org, Hello{ this.state.hello }</h1>
      <h2>
      <a onClick={this.changeState.bind(this)}>Press Here!</a>
      </h2>
      </div>);
  }
```

Example

```
  shouldComponentUpdate(nextProps, nextState)
   {
      console.log("shouldComponentUpdate()");
      return true;
   }
   componentWillUpdate()
   {
      console.log("componentWillUpdate()");
   }
   componentDidUpdate()
   {
      console.log("componentDidUpdate()");
   }
}
ReactDOM.render(
   <Test />,
   document.getElementById('root'));
```

Data & Data flow in React



- Mutable and immutable state
- Stateful and stateless components
- Component communication
- One-way data flow

State: All the information a program has access to at a given instant in time.

Example:

```
const letters = 'Letters’;
const splitLetters = letters.split(‘’);
console.log("Let's spell a word!");
splitLetters.forEach(letter => console.log(letter));
```

Mutable and immutable state:

- In React applications, there are two primary ways that you can work with state in components: through state that you can change, and through state that you shouldn’t.
- state and props are mutable and immutable respectively.
- In React components, state is generally mutable. props will not change.

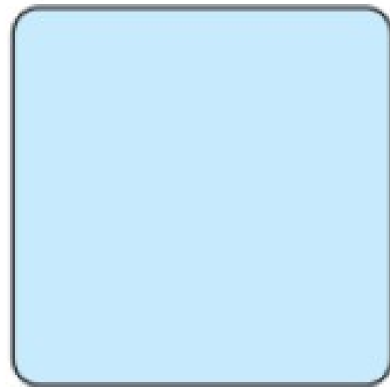
- we call state mutable we mean we can overwrite or update that data (for example, a variable that you can overwrite). Immutable state, on the other hand, can't be changed.
- There are also immutable data structures, which can be changed but only in controlled ways (this is sort of how the state API works in React).

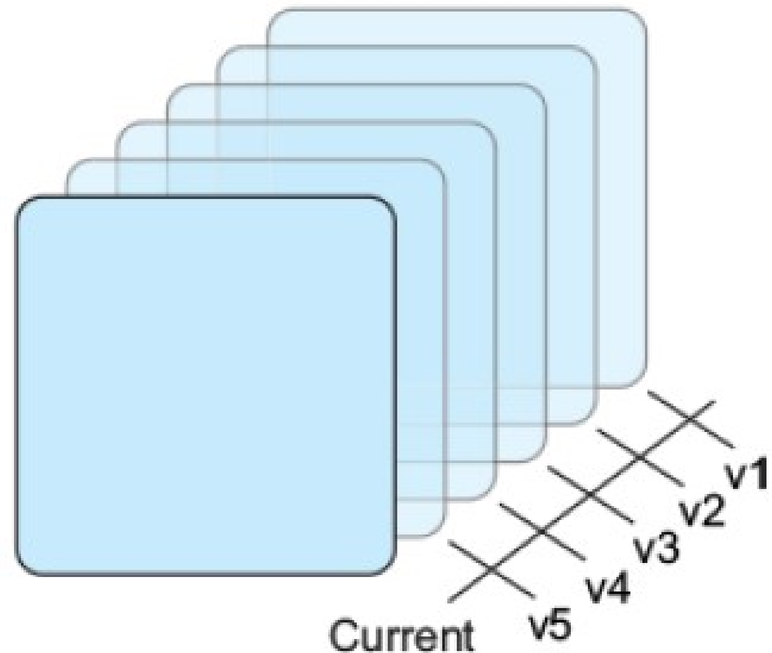
Immutable— An immutable, persistent data structure supports multiple versions over time but can't be directly overwritten; immutable data structures are generally persistent.

Mutable— A mutable, ephemeral data structure supports only a single version over time; mutable data structures are overwritten when they change and don't support additional versions.

Persistence and ephemerality in immutable and mutable data structures.

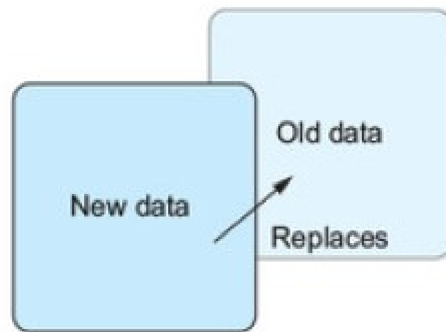
- Ephemeral (volatile) data structures only have the capacity to store a moment's worth of data, whereas persistent data structures can keep track of changes over time. This is where the immutability of immutable data structures becomes clearer: only copies of state are made—they're not replaced

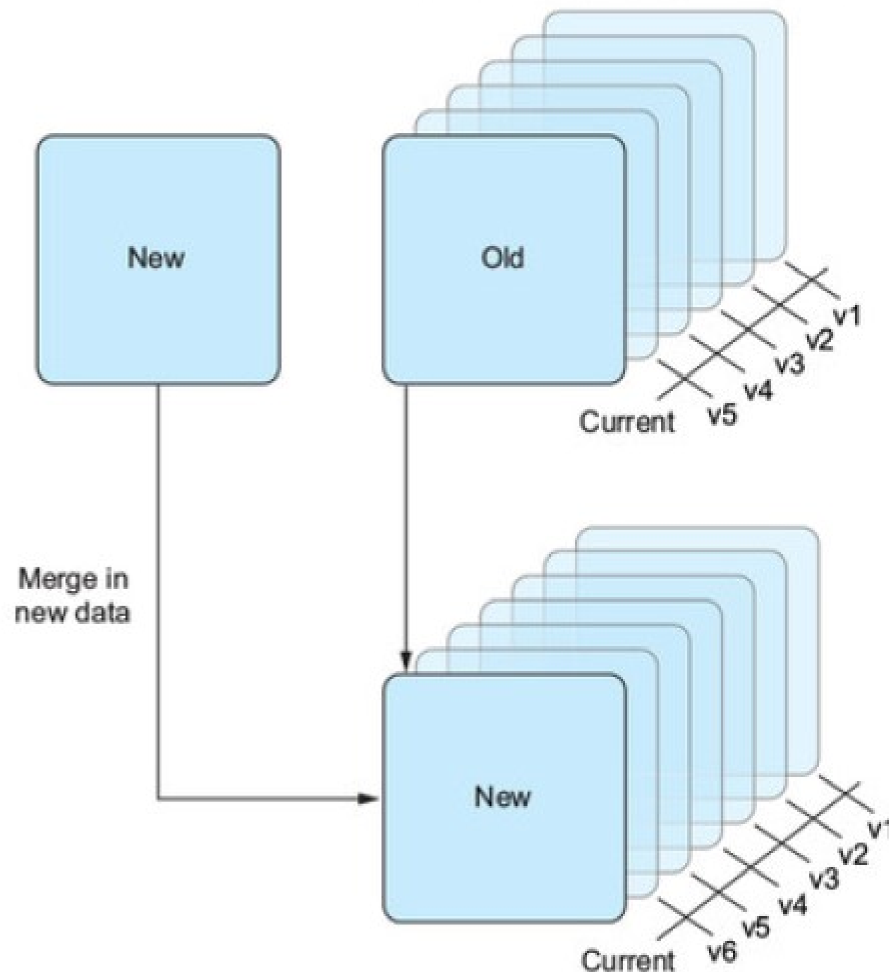




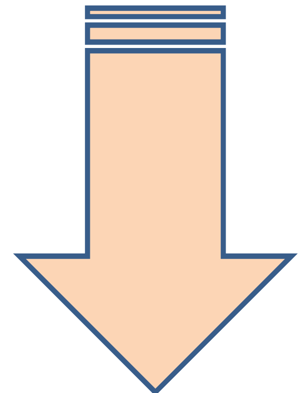
Handling changes with mutable and immutable data

- Immutable or persistent data structures usually record a history and don't change but rather make versions of what changed over time.
- Mutable (Ephemeral data structures), on the other hand, usually don't record history and get wiped out with each update.





Parent Child components communication



Communication between components

- ✓ React is a component-based UI library.
- ✓ In order to build up a system into something that can accomplish an interesting task, multiple components are needed. These components often need to work in coordination together and, thus, must be able to communicate with each other. Data must flow between them.

- ✓ Types: From parent to child → using props
- ✓ From child to parent → using callbacks

Parent to Child communication

- When you need to pass data from a parent to child class component, you do this by using props.
- For example, let's say you have two class components, Parent and Child, and you want to pass a state in the parent to the child. You would do something like this:

```
import React from 'react';
class Parent extends React.Component{
  constructor(props){  super(props);
    this.state = {        data: 'Data from parent'      }   }
  render(){
    const {data} = this.state;
    return(        <div>           <Child dataParentToChild = {data}/>          </div>       )   }}
class Child extends React.Component{   constructor(props){       super(props);
    this.state = {         data: this.props.dataParentToChild       }    }
  render(){
    const {data} = this.state;
    return(        <div>            {data}          </div>       )   }}
export default Parent;
```

Parent to Child communication

Parent:

```
function BookList() {
  const list = [
    { title: 'A Christmas Carol', author: 'Charles Dickens' },
    { title: 'The Mansion', author: 'Henry Van Dyke' },
    // …
]
  return (    <ul>
      {list.map((book, i) => <Book title={book.title} author={book.author} key={i} />)}
    </ul>
  )}
```

Child:

```
 function Book(props) {
  return (
    <li>
      <h2>{props.title</h2>
      <div>{props.author}</div>
    </li>
  )
}
```

Child to Parent communication

Steps:

1. Create a callback function in the parent component. This callback function will get the data from the child component.
2. Pass the callback function in the parent as a prop to the child component.
3. The child component calls the parent callback function using props.

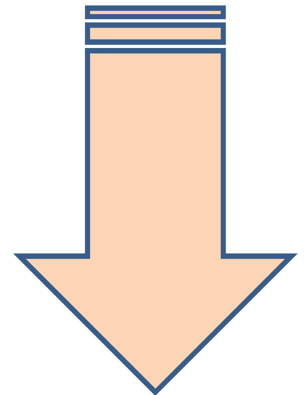
- When the Child component is triggered, it will call the Parent component's callback function with data it wants to pass to the parent. The Parent's callback function will handle the data it received from the child.

Child to Parent communication

```
class Parent extends React.Component{
   constructor(props){
      super(props);
      this.state = {
         data: null
      }   }
   handleCallback = (childData) =>{
      this.setState({data: childData})
   }
   render(){
      const {data} = this.state;
      return(<div> <Child parentCallback = {this.handleCallback}/>   {data} </div>
      )   }}
class Child extends React.Component{
   onTrigger = (event) => {
      this.props.parentCallback("Data from child");
      event.preventDefault();   }
   render(){return(<div>  <form onSubmit = {this.onTrigger}>
            <input type = "submit" value = "Submit"/>     </form> </div> )  }}
export default Parent;
```

Child to Parent communication

Forms



Forms

- Just like in HTML, React uses forms to allow users to interact with the web page.
- Handling forms is about how you handle the data when it changes value or gets submitted.
- In HTML, form data is usually handled by the DOM. In React, form data is usually handled by the components.
- When the data is handled by the components, all the data is stored in the component state.
- You can control changes by adding event handlers in the **onChange** attribute

Forms

```
class MyForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = { username: '' };
  }
  myChangeHandler = (event) => {
    this.setState({username: event.target.value});
  }
  render() {    return (
      <form>
              <h1>Hello {this.state.username}</h1>
              <p>Enter your name:</p>
              <input type='text' onChange={this.myChangeHandler} />
      </form>
    );  }
}
ReactDOM.render(<MyForm />, document.getElementById('root'));
```

- You must initialize the state in the constructor method before you can use it. You get access to the field value by using the **event.target.value** syntax.

Forms – Multiple Input fields

- You can control the values of more than one input field by adding a **name attribute** to each element.
- When you initialize the state in the constructor, use the field names.
- To access the fields in the event handler use the **event.target.name** and **event.target.value** syntax.
- To update the state in the this.setState method, use square brackets [bracket notation] around the property name.

useState hook

- The React useState Hook allows us to track state in a function component.
- State generally refers to data or properties that need to be tracking in an application.
- To use useState, first import it into the application

```
import { useState } from "react";
```

Initialize the useState:

useState accepts an initial state and returns two values:

- ✓ The current state.
- ✓ A function that updates the state.

```
import { useState } from "react";
function FavoriteColor()
        {
          const [color, setColor] = useState("");
        }
```

- The first value, color, is our current state.
- The second value, setColor, is the function that is used to update our state.

useState("") → set the initial state to an empty string:

useState hook

- Hooks allow function components to have access to state and other React features. Because of this, class components are generally no longer needed.

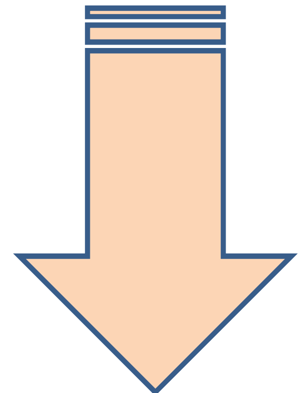
```
 import { useState } from "react";
import ReactDOM from "react-dom/client";

function FavoriteColor() {
 const [color, setColor] = useState("red");
const [color2, setColor2] = useState("");
const [color3, setColor3] = useState("");



  return <h1>My favorite color is {color}!</h1>
}

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

Update state -- useState

To update our state, we use our state updater function.

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

function FavoriteColor() {
  const [color, setColor] = useState("red");

  return (
    <>
      <h1>My favorite color is {color}!</h1>
      <button
        type="button"
        onClick={() => setColor("blue")}
      >Blue</button>
    </>
  )
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<FavoriteColor />);
```

Integrating Third Party Libraries



Integrating Third Party Libraries

- At their core, third-party libraries and plugins are pieces of code developed outside the primary React team but designed to work within the React ecosystem. Examples include Redux for state management, React Router for routing, and Styled Components for styling.
- there are libraries like Chart.js or Recharts
- Managing these third-party integrations requires a package manager. The most common in the React ecosystem is npm (Node Package Manager). Using commands like npm install <library-name>, developers can quickly add, update, or remove third-party code from their projects.

Benefits:

- **Rapid Development:** Instead of building functionality from scratch, third-party integrations allow developers to stand on the shoulders of giants, speeding up the development process.
- **Community Support:** Popular third-party libraries often come with robust community support. This means plenty of tutorials, resources, and an active community to help troubleshoot issues.
- **Optimized Performance:** These libraries and plugins are often optimized for performance and regularly updated to remain in sync with the latest web standards and best practices.

Integrating Third Party Libraries

Drawbacks:

- **Overhead:** Not all third-party solutions are lightweight. Incorporating hefty libraries can increase your application's bundle size, affecting page load times.
- **Learning Curve:** Every library or plugin has its API and way of doing things. Developers might need to invest time in learning these before they can effectively integrate them.
- **Dependence on External Code:** Using third-party solutions means relying on external code, which can be a concern if that library is deprecated or isn't regularly maintained.

Guidelines to use third party library:

1. **Research the Library/Plugin:** Before integrating any third-party library, ensure that it's well-maintained, has good community support, and is compatible with your React version. You can check for the number of downloads, last update, and issues on its GitHub repository.
2. **Installation:** Use npm or yarn to install the library.

   ```
   npm install <library-name>
   ```

3. **Import the installed library from the package**

   ```
   I	mport {package-name} from 'file-name';
   ```

Integrating Third Party Libraries

4. Consult the Documentation: Every library/plugin usually comes with its set of configurations and props. Always refer to the official documentation for setup and usage guidelines.

5. Test the Implementation: After integrating, test to ensure that there are no conflicts or issues in the app. This includes visual rendering, functionality, and performance

Examples:

1. **Ant Design (antd)** → Ant Design is a popular design system with a set of high-quality React components.

```
npm install antd
Import {Button} from 'antd';
```

2. Chart.js:

Chart.js is a powerful data visualization library. You can use the react-chartjs-2 wrapper to use Chart.js in your React applications.

```
npm install react-chartjs-2 chart.js
import { Bar } from 'react-chartjs-2';
```

3. Redux:

Redux is a popular state management library. You can install it with the react-redux package:

```
npm install redux react-redux
        import { createStore } from 'redux';
import { Provider } from 'react-redux';
```

Integrating Third Party Libraries

Pagination:

A long list can be divided into several pages using Pagination, and only one page will be loaded at a time.

AutoComplete:

AutoComplete is an input box with text hints, and users can type freely.

Integrating Third Party Libraries

- The good part of using this third party library is it boost the application development process and helps to gain the goal.
- You'll build React applications in a context that involves nonReact libraries that also work with the DOM. These might include things like jQuery, jQuery plugins, or even other frontend frameworks.
- We've seen that React manages the DOM for you and that this can simplify how you think about user interfaces.
- There are many libraries that are written in plain Javascript or as a JQuery plugin, an example is Datatable.js. There is no need to reinvent the wheel, consume a lot of time and energy, and re-create those libraries.
- **Third-party libraries can be integrated with class components,** also **with functional components using Hooks.**
- A React.js component may update the DOM elements multiple times during its lifecycle after component props or states update.
- Some libraries need to know when the DOM is updated. Some other libraries need to prevent the DOM elements from updating.
- **Datatables.js is a free JQuery plugin** that adds advanced controls to HTML tables like searching, sorting, and pagination.
- **Refs:** React provides a way for developers to access DOM elements or other React elements. Refs are very handy when integrating with third-party libraries.

Integrating Third Party Libraries

- We need to know some lifecycle methods. These lifecycle methods are important for initializing other libraries, destroying components, subscribing and unsubscribing events.

React Lifecycle methods required to know:

We need to know some lifecycle methods. These lifecycle methods are important for initializing other libraries, destroying components, subscribing and unsubscribing events

1. **componentDidMount**

it is fired when the element is mounted on the DOM. It is like jquery's $(document).ready().

Usage:

- ✓ fetching data from the server.
- ✓ initializing third-party libraries.

Example:

```
componentDidMount() {
  this.$el = $(this.el);
  this.currentTable = this.$el.DataTable({});
}
```

Integrating Third Party Libraries

2. componentDidUpdate

it is fired when the props passed to the component are updated or the method this.setState is called to change the state of the component. This method is not called for the initial render().

Usage:

- ✓ reload third-party library if props is updated.

Example:

```
componentDidUpdate(prevProps) {
  if (prevProps.children !== this.props.children) {
    // update third-party library based on prop change
  }
}
```

3. componentWillUnmount: it is fired before the React component is destroyed and unmounted on the DOM.

Usage:

- ✓ Unsubscribing from events
- ✓ Destroying third-party library

Integrating Third Party Libraries

4. shouldComponentUpdate: it is used to avoid the React component from re-rendering. It prevents to update the DOM even if the state or props are updated.

Usage:

Some libraries require an un-changeable DOM.

Example:

```
shouldComponentUpdate() {
  return false;
}
```

Integrating Third Party Libraries - Jquery Datatables

- Datatables.js is a free JQuery plugin that adds advanced controls to HTML tables like searching, sorting, and pagination.

Steps:

1. Need to install a couple of dependencies from npm: jquery and datatables.net

   npm i -S jquery datatables.net

2. Add a link to DataTable.css file in index.html.

```
<link rel="stylesheet" href="https://cdn.datatables.net/1.10.23/css/jquery.dataTables.min.css" />
```

3. Create a class component named DataTable inside

   components/DataTable.js.

4. Import the libraries:

```
var $ = require("jquery");
$.DataTable = require("datatables.net");
```

5. Inside the render() method, we need to have a table element with a ref. It looks like an html ID, we use it for selecting (referencing) it.

Integrating Third Party Libraries - Jquery Datatables

6. We need to **render children props inisde the tbody** which is passed by the parent element.

```
render() {
  return (
    <table ref={(el) => (this.el = el)}>
      <thead>
        <tr>
          <th>#</th>
          <th>Title</th>
          <th>Completed</th>
          <th></th>
        </tr>
      </thead>
      <tbody>{this.props.children}</tbody>
    </table>
  );
}
```

Integrating Third Party Libraries

7. Inside the **componentDidMount**() method, we need to get the ref and call jquery method DataTable()

```
componentDidMount() {
  this.$el = $(this.el);
  this.currentTable = this.$el.DataTable();
}
```

8. Inside the **componentDidUpdate**(prevProps), we refresh the datatable by calling ajax.reload() when the props are updated. According to datatable.js, this method refreshes the table.

```
componentDidUpdate(prevProps) {
  // It means that only when props are updated
  if (prevProps.children !== this.props.children) {
    this.currentTable.ajax.reload();
  }
}
```

9. Finally, inside **componentWillUnmount**() we destroy the table.

```
componentWillUnmount() {
  this.currentTable.destroy();
}
```

Integrating Third Party Libraries

10. Using the DataTable component in our react application.

```
import React from "react";
import DataTable from "./components/DataTable";

class App extends React.Component {
  state = {
    todos: [],
  };
  componentDidMount() {
    fetch("https://jsonplaceholder.typicode.com/todos")
      .then((res) => res.json())
      .then((data) =>
        this.setState({
          todos: data,
        })
      );
  }
```

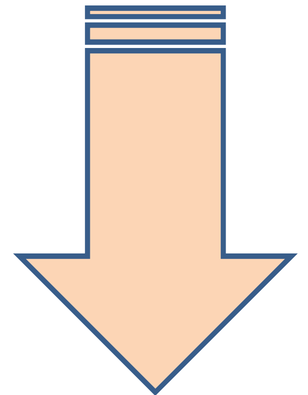
Integrating Third Party Libraries

```
 render() {
    return (
      <DataTable>
        {this.state.todos.map((todo) => (
          <tr key={todo.id}>
            <td>{todo.id}</td>
            <td>{todo.title}</td>
            <td>{todo.completed ? "Yes" : "No"}</td>
            <td>
              <button>Edit</button>
              <button>Delete</button>
            </td>
          </tr>
        ))}
      </DataTable>
    );
  }
}
export default App;
```

Integrating Third Party Libraries

•

Routing



Routing

- Routing is a process in which a user is directed to different pages based on their action or request. ReactJS Router is mainly used for developing Single Page Web Applications.
- Routing is the ability to move between different parts of an application when a user enters a URL or clicks an element (link, button, icon, image etc) within the application.
- To add routing capabilities, you will use the popular React-Router library. It's worth noting that this library has three variants:
  1. **react-router:** the core library
  2. **react-router-dom:** a variant of the core library meant to be used for web applications
  3. **react-router-native:** a variant of the core library used with react native in the development of Android and iOS applications.

- Both **react-router-dom** and **react-router-native** import all the functionality of the core react-router library.
- To install **react-router-dom** as part of the current project:

  npm install --save react-router-dom @6
-

Routing

- The **react-router** package includes a number of routers that we can take advantage of depending on the platform we are targeting. These include
  - BrowserRouter,
  - HashRouter, and
  - MemoryRouter.

- For the browser-based applications we are building, the BrowserRouter and HashRouter are a good fit.
- The **BrowserRouter** is used for applications which have a dynamic server that knows how to handle any type of URL
- The **HashRouter** is used for static websites with a server that only responds to requests for files that it knows about.

Router components

- The Main Components of React Router are:
- **BrowserRouter**: BrowserRouter is a router implementation that uses the HTML5 history API(pushState, replaceState, and the popstate event) to keep your UI in sync with the URL. It is the parent component that is used to store all of the other components.
- **Routes**: Used to specify the routes i.e for a given path what is the element to be selected.
- It's a new component introduced in the v6 and an upgrade of the component. The main advantages of Routes over Switch are:
  - Relative s and s
  - Routes are chosen based on the best match instead of being traversed in order.
- **Route:** Route is the conditionally shown component that renders some UI when its path matches the current URL.
- **Link:** The link component is used to create links to different routes and implement navigation in the application. It works like an HTML anchor tag.

Routing

Example using BrowserRouter: In this example, the <App/> component is the child to the <BrowserRouter> and should be the only child. Now, the routing can happen anywhere within the <App/> component.

```
ReactDOM.render(
  <BrowserRouter>
    <App/>
  </BrowserRouter>,
  document.getElementById('root'));
```

BrowserRouter

```
<BrowserRouter>
        <div>
            <ul>
             <li>
               <Link to="/">Home</Link>   </li>
             <li>
               <Link to="/About“>  About Us   </Link>   </li>
             <li>
               <Link to="/Contact“>     Contact Us     </Link>   </li>
          </ul>
          <Routes>
             <Route path="/" element={<Home />}>   </Route>
             <Route path="/About" element={<About />}>   </Route>
             <Route path="/Contact" element={<Contact />}>  </Route>
          </Routes>
        </div>
</ BrowserRouter >
```

React router

Steps:

1. Install react-router in the project folder

C:\Users\username\Desktop\reactApp>npm install –save react-router-dom@6

2. Create Components

In this step, we will create four components. The **App** component will be used as a tab menu. The other three components **(Home), (About) and (Contact)** are rendered once the route has changed

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, Route, Link, browserHistory, IndexRoute } from 'react-router'
class App extends React.Component {
  render() {
    return (<div>   <ul>          <li>Home</li>          <li>About</li>          <li>Contact</li>
        </ul>
        {this.props.children}
      </div>
    )
  }
}
export default App;
```

React router

Step2: contd…

```
class Home extends React.Component {
  render() {      return (
      <div>          <h1>Home...</h1>        </div>
    )  }}
export default Home;
class About extends React.Component {
  render() {
    return (       <div>          <h1>About...</h1>        </div>
    )  }}
export default About;
class Contact extends React.Component {
  render() {
    return (       <div>          <h1>Contact...</h1>        </div>
    )  }
}
export default Contact;
```

https://www.geeksforgeeks.org/reactjs-router/

React Router – Example 2

{this.props.children} → means that render my children here.

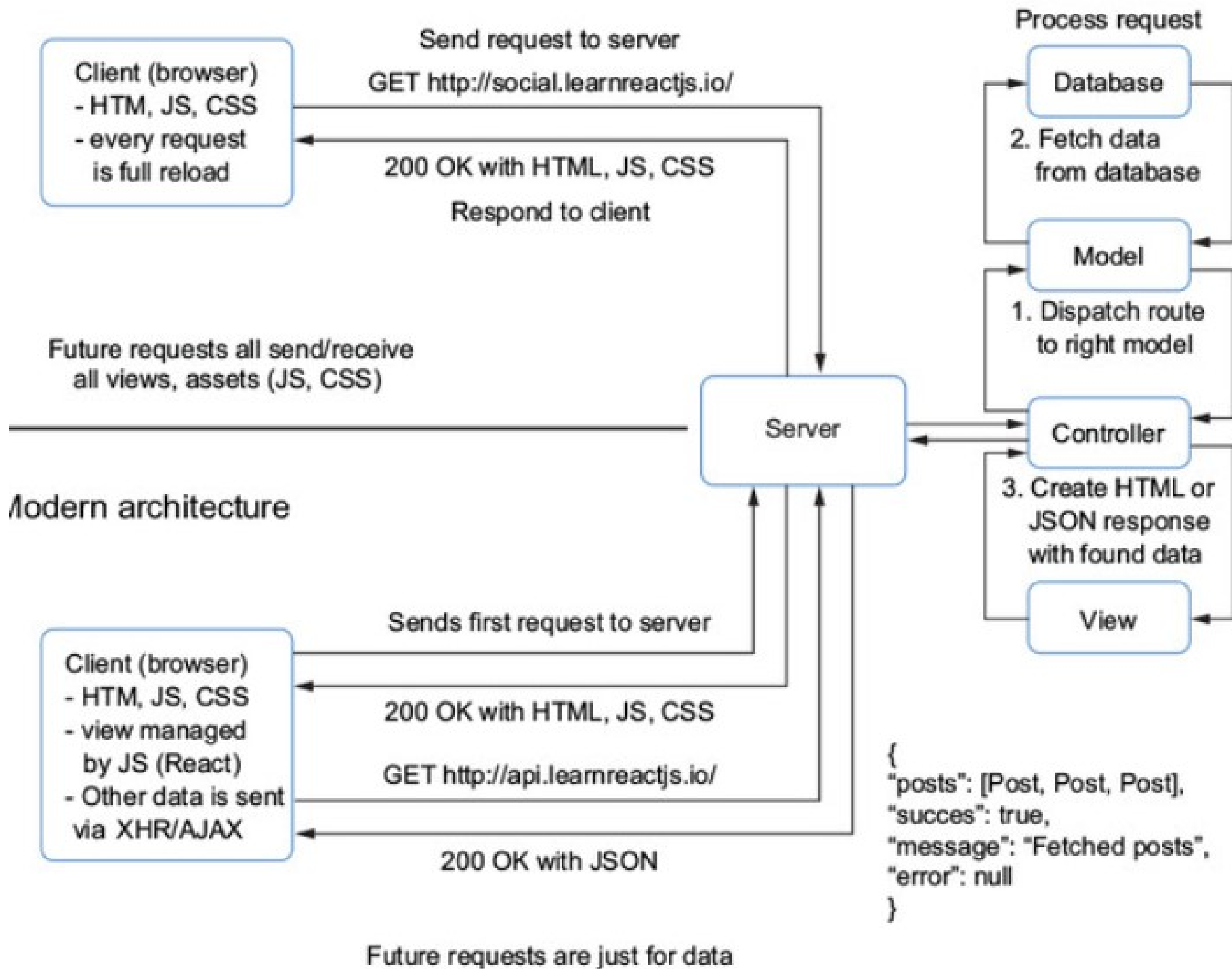
<IndexRoute> → If you want a child route to be used as the default when no other child matches, this special route is used.

- To avoid error:03000086:digital envelope routines::initialization error' in package.json, add
- "start": "react-scripts --openssl-legacy-provider start",
- "build": "react-scripts --openssl-legacy-provider build",

Old & New Web Appl. Architecture

- In the old way, dynamic content would be generated on the server. The server would usually fetch data from a database and use it to populate an HTML view that would be sent down to the client.
- Now there is more application logic on the client that gets managed by JavaScript (in this case, React). The server initially sends down the HTML, JavaScript, and CSS assets, but after that, the client React app takes over. From there, unless a user manually refreshes the page, the server will only have to send down raw JSON data.

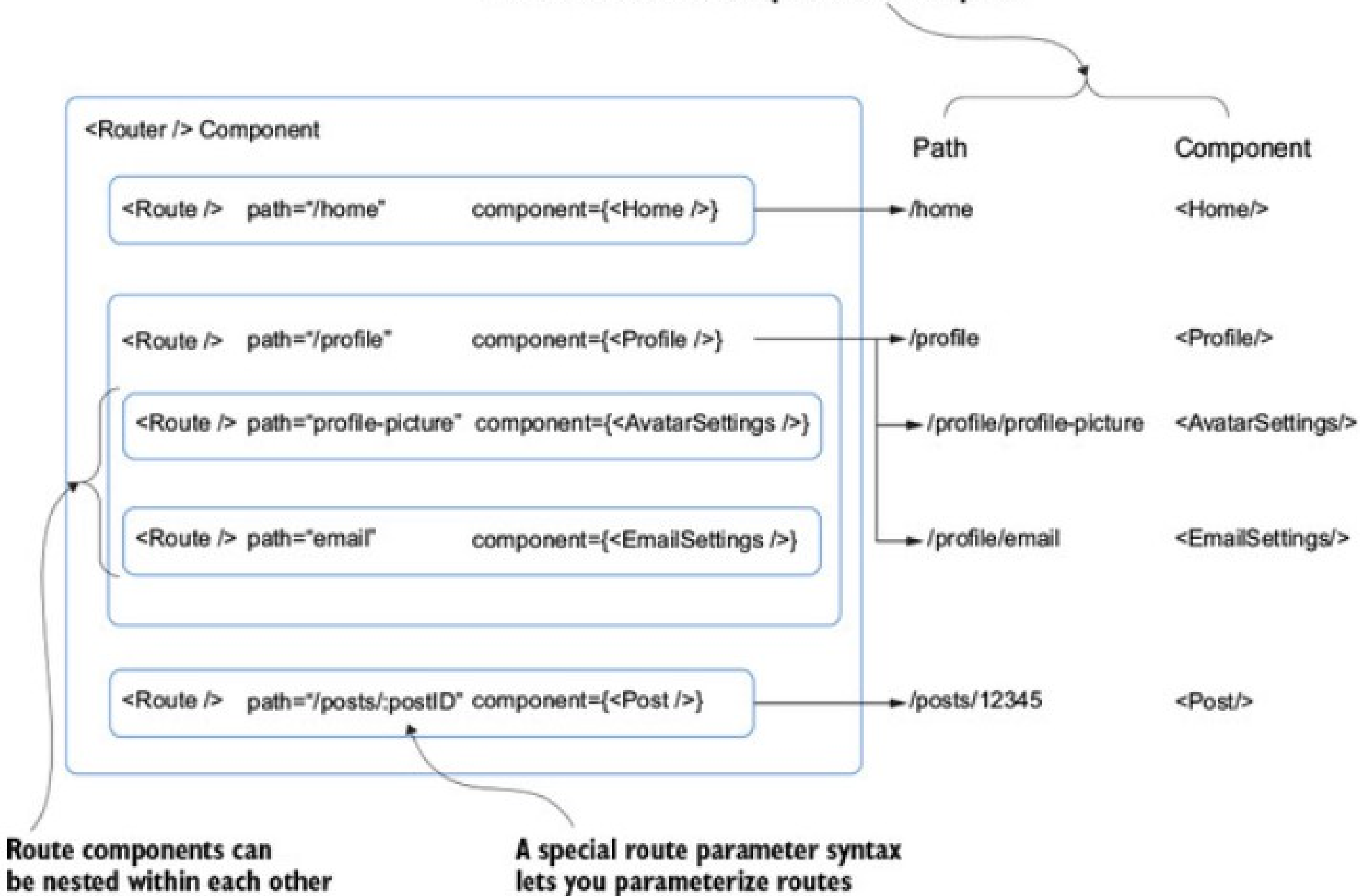
React Router



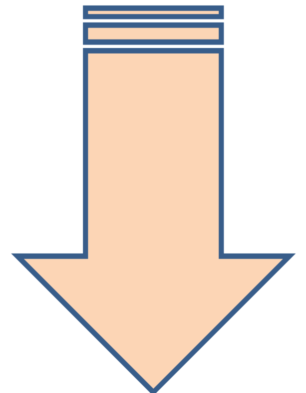
Router - Route

- The Router has Route components as its children. Each of these components uses **two props: a `path` string and a component.** The `<Router/>` will use each `<Route/>` to match a URL and render the right component.

Router – Route Components working



Node.js



Node.js

2. **Node.js:**

- Node is a "a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an **event-driven, non-blocking I/O ,single threaded, asynchronous model** that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices."
- It is a runtime which runs the java script programs outside the browser.
- The creators of Node.js just took Chrome's V8 JavaScript engine and made it run independently as a JavaScript runtime.
- Node.js has a set of built-in modules which you can use without any further installation.
- Node.js has single-threaded, event driven, non-blocking I/O, asynchronous model which is very memory efficient.
- Uses events and callback functions to achieve asynchronous mode of operation.

Node.js tasks:

- Node.js can generate dynamic page content
- Node.js can create, open, read, write, delete, and close files on the server
- Node.js can collect form data
- Node.js can add, delete, modify data in your database

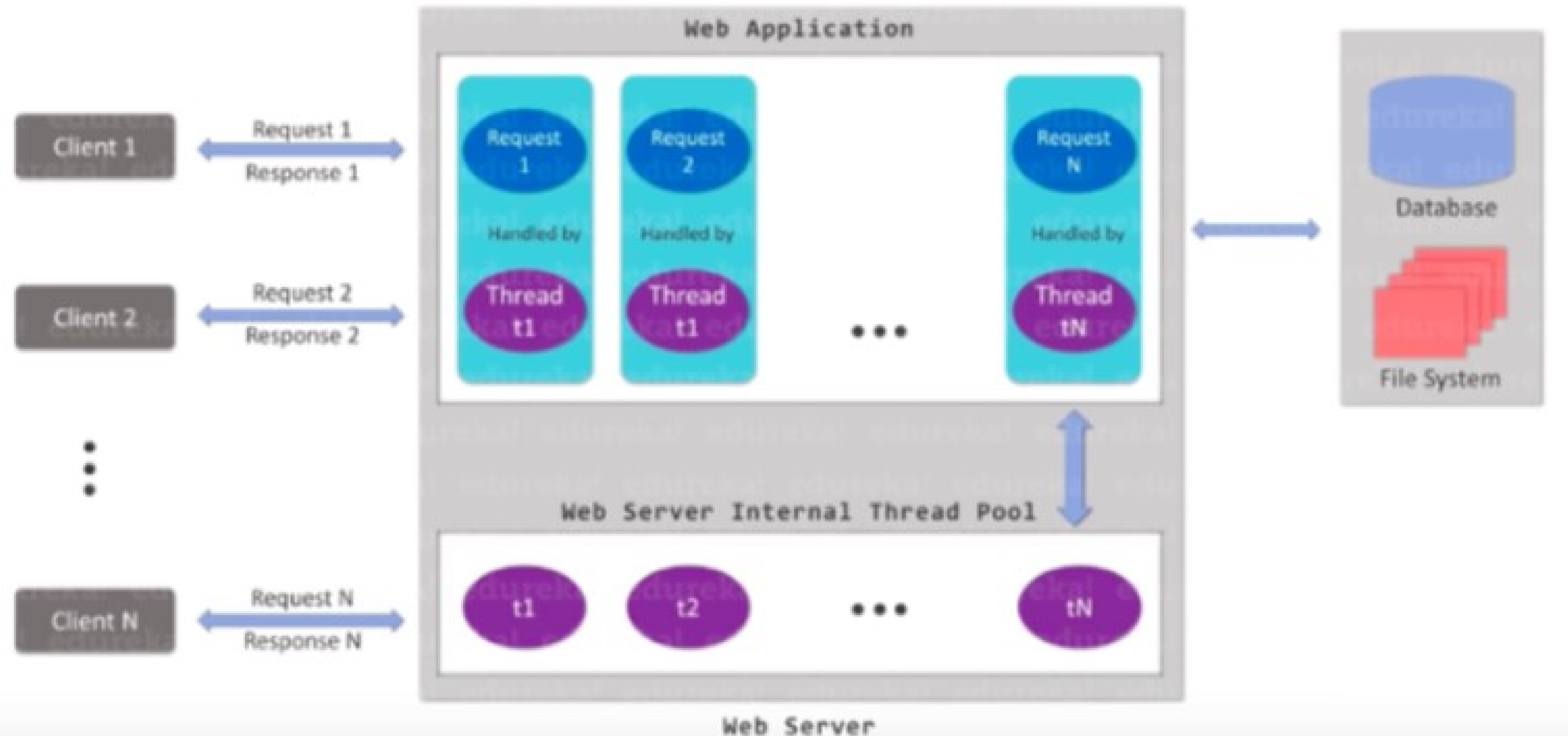
Node.js modules:

- Module is a set of functions.
- Node.js ships with a bunch of core modules compiled into the binary. These modules provide access to the operating system elements such as the fThey also provide some utility functions that are commonly required by most programsile system, networking, input/output, etc..

Node.js

Event-Driven, Non Blocking-I/O Asynchronous, single-threaded Model of node.js:

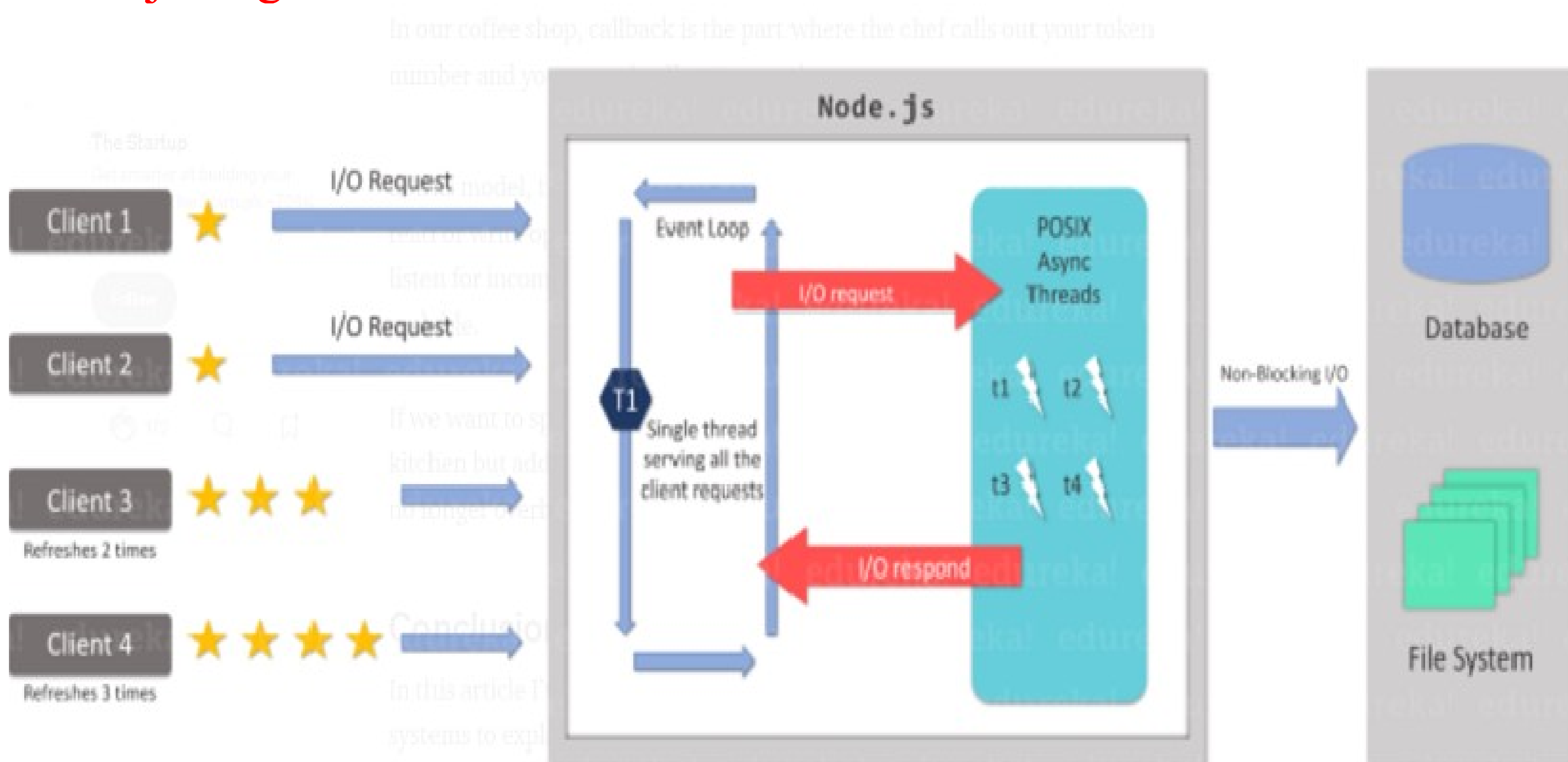
Traditional thread based model:



In a traditional thread-based model when the server receives a connection, it holds the connection open until it has performed the request which can either be a page request or a costly transaction like writing something to a database.

Node.js

Node.js single thread based model:



In this model, the web server does not have to wait for the completion of read or write operations of previous requests. Its only task is to continuously listen for incoming requests, thereby making the model highly efficient and scalable.

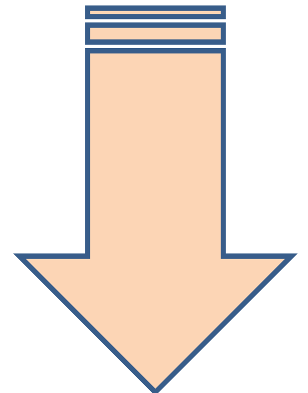
Node.js

Installation:

Node.js: https://nodejs.org

Node documentation: https://nodejs.org/en/docs/

Express.js



Express

3. Express:

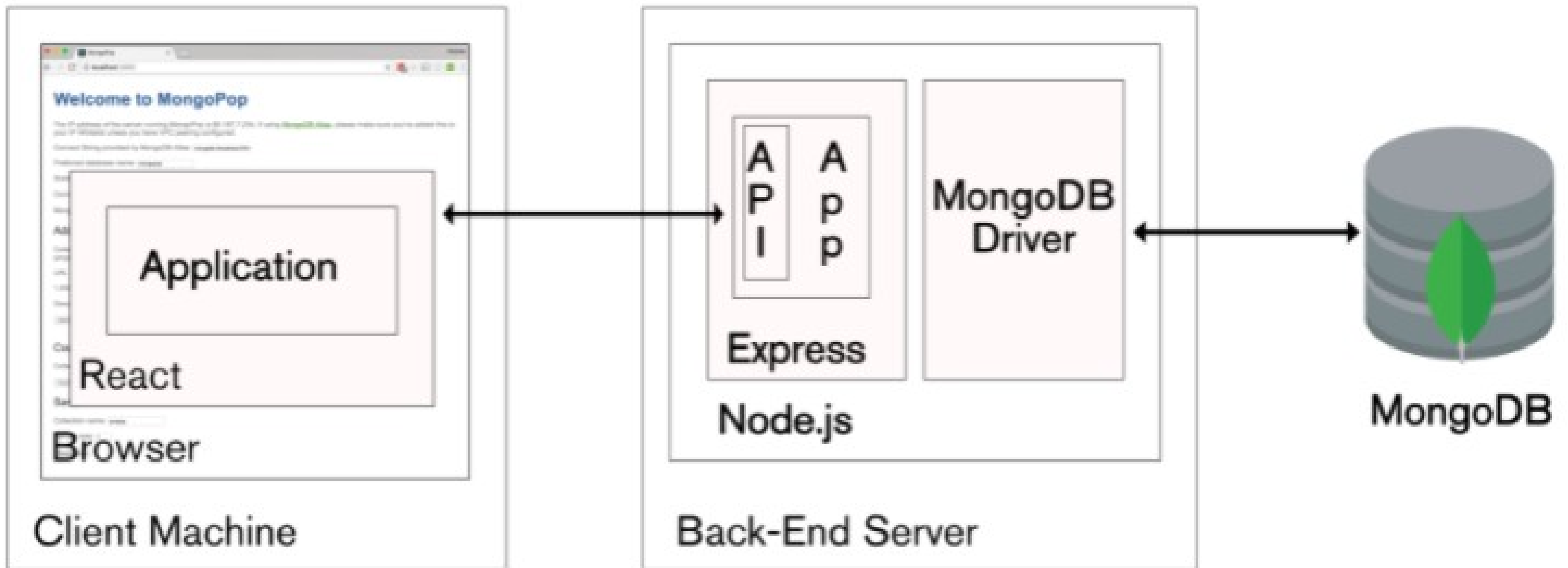
- Express is a Web Server framework for Node.js
- **Node.js is just a runtime environment that can run JavaScript.** To write a full-fledged web server by hand on Node.js directly is not easy, neither is it necessary.
- Express is a framework that simplifies the task of writing the server code.
- Express
  - **parses the request URL, headers, and parameters for you**.
  - On the response side, it has, as expected, all functionality required by web applications. This includes **determining response codes, setting cookies, sending custom headers, etc**.
  - Further, you can write Express *middleware*, custom pieces of code that can be inserted in any request/response processing path to achieve common functionality such as **logging, authentication**, etc.

Installation: npm install express --save

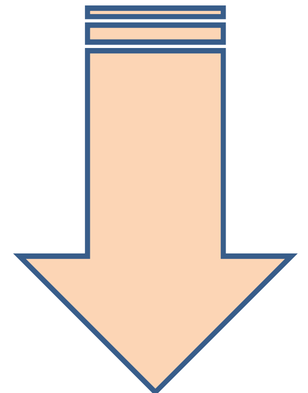
NPM (Node Package Manager): npm is the default package manager for Node.js.

- You can use npm **to install third-party libraries** (packages) and manage dependencies between them.

Express



Mongo DB



MongoDB

- MongoDB is the database used in the MERN stack. It is a NoSQL document-oriented database, with a flexible schema and a JSON-based query language.
- Not only do many modern companies (including **Facebook and Google**) use MongoDB in production, but some older established companies such as **SAP** and **Royal Bank of Scotland** have adopted MongoDB.

NOSQL:

- NoSQL stands for "non-relational," no matter what the acronym expands to.
- It's essentially *not* a conventional database where you have tables with columns and rows and strict relationships among them.
- The first is their ability to **horizontally scale** by distributing the load over multiple servers.

Document – Oriented:

- Compared to relational databases where data is stored in the form of relations, or tables, MongoDB is a document-oriented database.
- The unit of storage (comparable to a row) is a *document*, or an object,
- multiple documents are stored in *collections* (comparable to a table). Every document in a collection has a unique identifier, using which it can be accessed. The identifier is indexed automatically.
- data is stored de-normalized. This means that data is sometimes duplicated, requiring more storage space

MongoDB

Schema-Less:

- Storing an object in a MongoDB database does not have to follow a prescribed schema. All documents in a collection need not have the same set of fields.
  –

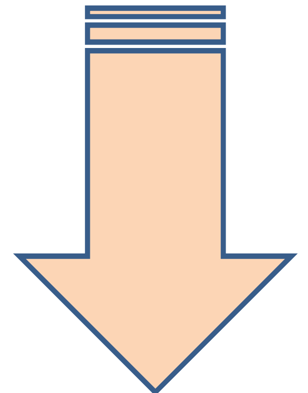
JavaScript Based:

– MongoDB's language is JavaScript.

- For relational databases, we had a query language called SQL. For MongoDB, the query language is based on JSON.
- You create, search for, make changes, and delete documents by specifying the operation in

  a JSON object. The query language is not English-like (you don't SELECT or say WHERE), and therefore much easier to construct programmatically.

Tools & Libraries



Tools & Libraries

Tools:

- React-Router
- Webpack
- Npm (node package manager)

Libraries:

- React-Bootstrap
- Redux
- *Mongoose*
- *Jest*

React-Router:

- To perform transitioning between different views of the component and keeping the browser URL in sync with the current state of the view. This managing URLs and history is called ***routing****.*
- *Managing Browser's Back button.*
- It is a very easy-to-use library

React-Bootstrap:

- Bootstrap, the most popular CSS framework, has been adapted to React and the project is called React-Bootstrap. This library gives us most of the Bootstrap functionality

Tools & Libraries

- There are other component/CSS libraries built for React (such **as Material-UI, MUI, Elemental UI**, etc.) and also individual components (such as **react-select, react-treeview, and react-date-picker**).

Webpack:

- **Used For modularizing the code, building & compiling** the client-side code into a bundle to deliver to the browser.

Other Libraries:

Body-parser, ESLint → server-side library

Body-parser: to parse POST data in the form of JSON, or form data

ESLint: for ensuring code follows conventions

React-select → client-side library

Other Popular Libraries:

Redux: This is a state management library that also combines the Flux programming pattern.

It's typically used in larger projects where even for a single screen, managing the state becomes complex.

Mongoose: If you are familiar with Object Relational Mapping layers, you may find Mongoose somewhat similar. This library **adds a level of abstraction over the MongoDB database layer and lets the developer see objects as such**. The library also provides other useful conveniences when dealing with the MongoDB database.

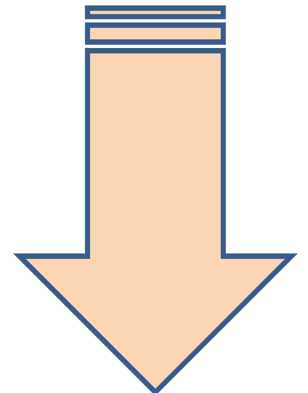
Jest: This is a testing library that can be used to test React applications easily.

Versions

Table 1-1. Versions of Various Tools and Libraries

Component	Major Version	Remarks
Node.js	10	This is an LTS (long term support) release
Express	4	--
MongoDB	3.6	Community Edition
React	16	--
React Router	4	--
React Bootstrap	0.32	There is no 1.0 release yet, things may break with 1.0
Bootstrap	3	This is compatible with React Bootstrap 0 (and 1 when released)
Webpack	4	--
ESLint	5	--
Babel	7	--

Why MERN?



Why MERN?

- **MERN** is ideally suited for web applications that have a large amount of interactivity built into the front-end.

MERN features:

1. JavaScript Everywhere
2. JSON Everywhere
3. Node.js Performance
4. The npm Ecosystem
5. Isomorphic
6. It's not a Framework

JavaScript Everywhere:

- The best part about MERN that I like is that there is a single language used everywhere. We use JavaScript for client-side code as well as server-side code.
- Even if you have database scripts (in MongoDB), you write them in JavaScript. So, the only language you need to know and be comfortable with is JavaScript.
- You don't even need to know a template language that generates pages.
- Only you will need to know is HTML and CSS and Java Script.
- Having a single language across tiers also lets you share code between these tiers.

Why MERN?

JSON Everywhere:

- When using the MERN stack, object representation is JSON (JavaScript Object Notation) everywhere—in the database, in the application server and on the client, and even on the wire.
- Saves time because of **no object transformations.**
- **No Object Relational Mapping (ORM),** not having to force fit an object model into rows and columns
- **No special serializing and de-serializing code.**
- **You save a *lot* of data transformation code.**

Node.js Performance:

- Due to its event-driven architecture and non-blocking I/O, the claim is that Node.js is very fast and a resilient web server.
- When your application starts scaling and receiving a lot of traffic, this will play an important role in cutting costs and savings in terms of time spent in trouble-shooting server CPU and I/O problems.

Why MERN?

The npm Ecosystem:

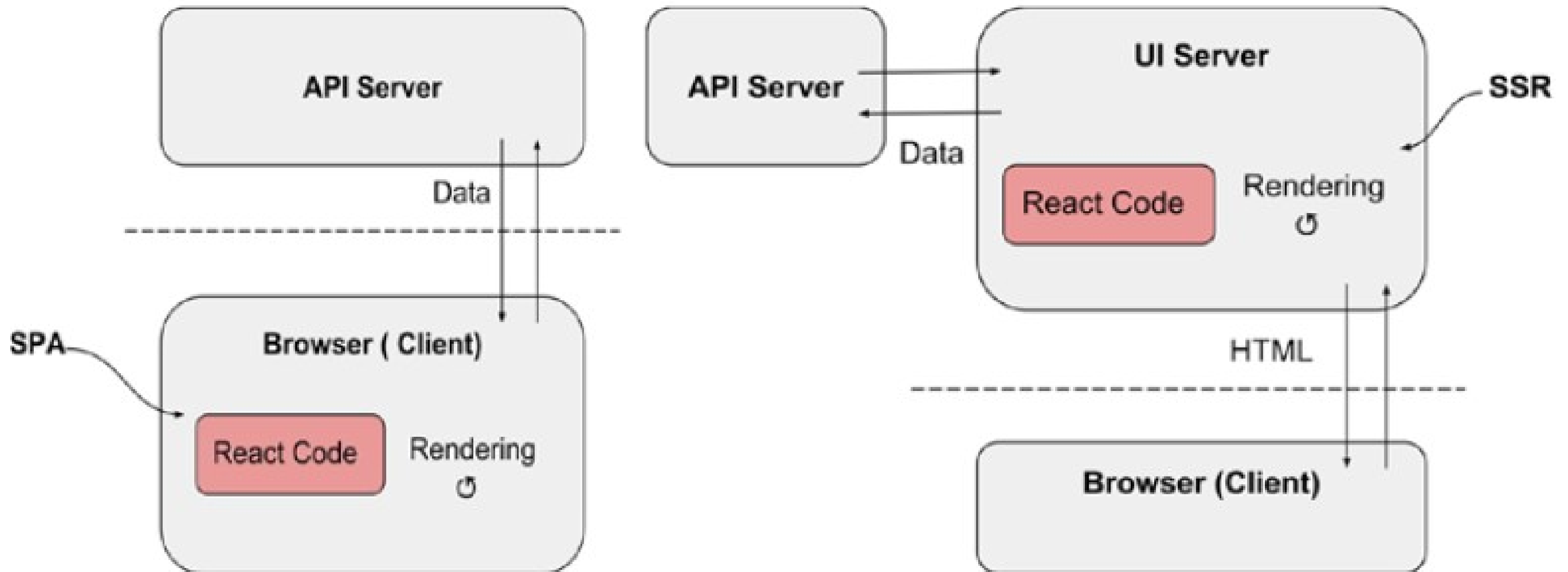
- huge number of npm packages available freely for everyone to use. Most problems that you face will have an npm package already as a solution. Even if it doesn't fit your needs exactly, you can fork it and make your own npm package.
- npm is by far the easiest to use and fastest package manager.
- Most npm packages are so small, due to the compact nature of JavaScript code.

Isomorphic:

- SPAs are not SEO (Search Engine Optimization) friendly ecause a search engine would not make Ajax calls to fetch data or run JavaScript code to render the page. PhantomJS is used on the server to pseudo-generate HTML pages.
- With the MERN stack, serving complete pages out of the server is natural and doesn't require tools that are after-thoughts. This is possible because React runs on JavaScript, which is the same on the client or the server. When React-based code runs on the browser, it gets data from the server and constructs the page (DOM) in the browser. This is the SPA way of rendering the UI. If we wanted to *generate* the same page in the server for search engine bots, the same React-based code can be used to get the data from an API server and construct the page (this time, as an HTML) and stream that back to the client. This is called *server-side rendering* (SSR).
- The same language is used to run the UI construction code in the server and the browser: **JavaScript.** This is what is meant by the term isomorphic: the same code can be run on the browser or the server.

Why MERN?

Isomorphic:



Why MERN?

It's not a Framework:

- React is a library, not a framework.
- A framework helps a lot by getting most of the standard stuff out of the way. A library, on the other hand, gives you tools to construct your application.
- With a library, experienced architects can design their own applications with complete freedom to pick and choose from the library's functions and build their own frameworks that fit their application's unique needs and vagaries. So, for an experienced architect or very unique application needs, a library is better, even though a framework can get you started quickly.

const, let, var

const:

- Const variables are cannot be updated or redeclared. This way is used to declare constants

```
const fruit = "Apple";
console.log(fruit); // prints Apple

(function () {
  const fruit = "Mango";
  console.log(fruit); // prints Mango
})();

const fruit = "Grape"; // Error

fruit = "Grape"; // Error
console.log(fruit); //prints Apple
```

const, let, var

```
const market = {
  name: "Fruit Market",
  location: "Somewhere",
  isOpen: "true",
};

market.isOpen = false;

console.log(market.isOpen); // prints false
```

let

- This is the improved version of var declarations. Variables declaration using this way eliminates all the issues that we discussed earlier. let creates variables that are block-scoped.
- Also, they can not be redeclared and can be updated. The below example shows it is the best choice to use let than var.

```
let fruit = "Apple";
console.log(fruit); // prints Apple

(function () {
  let fruit = "Mango";
  console.log(fruit); // prints Mango
})();

let fruit = "Grape"; // Error

fruit = "Grape";
console.log(fruit); //prints Grape
```

Var

- Variables declared using var keyword scoped to the current execution context. This means If they are inside a function we only can access them inside the function. and If they are not they are part of the global scope which we can access anywhere

```
var fruit = "Apple";
console.log(fruit); // prints Apple

(function someFunction() {
  var vegetable = "Carrot";
  console.log(vegetable); // prints Carrot
  console.log(fruit); // prints Apple
})();

console.log(vegetable); // Error
```

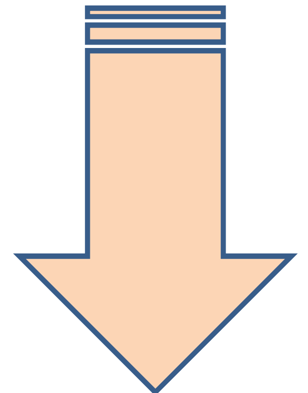
Component Constructor

- If there is a constructor() function in your component, this function will be called when the component gets initiated.
- The constructor function is where you initiate the component's properties.
- In React, component properties should be kept in an object called state.
- The constructor function is also where you honor the inheritance of the parent component by including the super() statement, which executes the parent component's constructor function, and your component has access to all the functions of the parent component (React.Component).

Example:

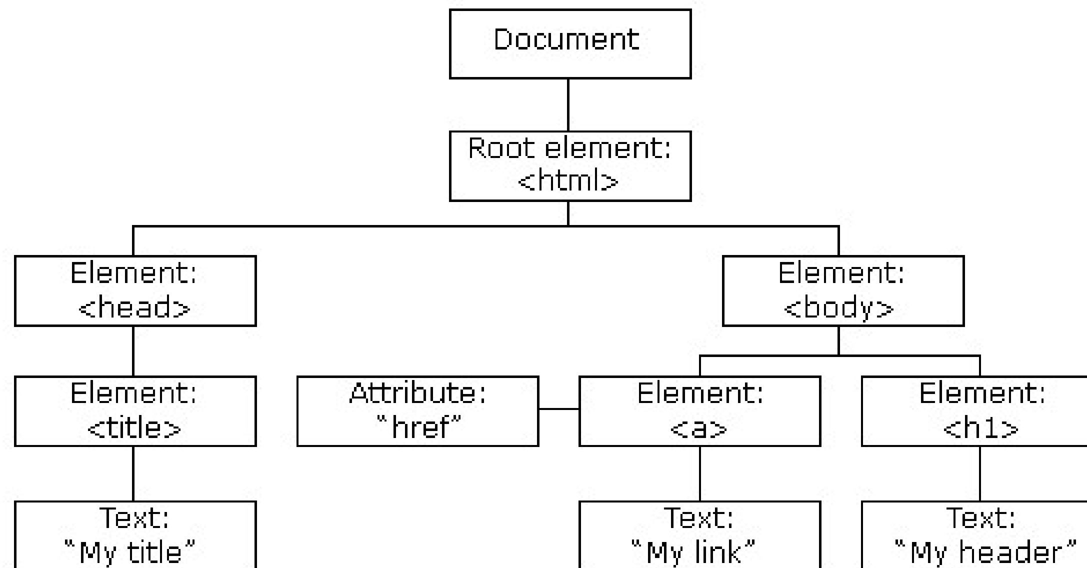
```
class Car extends React.Component {
  constructor() {
    super();
    this.state = {color: "red"};
  }
  render() {
    return <h2>I am a {this.state.color} Car!</h2>;
  }
}
```

DOM, VDOM & JSON

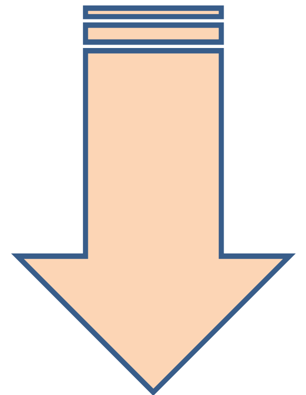


DOM

- When a web page is loaded, the browser creates a Document Object Model of the page.
- With the HTML DOM, JavaScript can access and change all the elements of an HTML document.
- The HTML DOM model is constructed as a tree of Objects:



Node.js



DOM

The HTML DOM is a standard object model and programming interface for HTML.

It defines

- The HTML elements as objects
- The properties of all HTML elements
- The methods to access all HTML elements
- The events for all HTML elements

With the object model, JavaScript gets all the power it needs to create dynamic HTML:

- JavaScript can change all the HTML elements in the page
- JavaScript can change all the HTML attributes in the page
- JavaScript can change all the CSS styles in the page
- JavaScript can remove existing HTML elements and attributes
- JavaScript can add new HTML elements and attributes
- JavaScript can react to all existing HTML events in the page
- JavaScript can create new HTML events in the page

In other words: The HTML DOM is a standard for how to get, change, add, or delete HTML elements.

DOM

Example:

The following example changes the content (the innerHTML) of the <p> element with id="demo":

```
<!DOCTYPE html>
<html>
<body>
<h2>My First Page</h2>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = "Hello World!";
</script>
</body>
</html>
```

- to get the content of an element, innerHTML property is used

Html DOM Methods

- HTML DOM methods are actions you can perform (on HTML Elements).
- HTML DOM properties are values (of HTML Elements) that you can set or change.

- The HTML DOM can be accessed with JavaScript (and with other programming languages).
- In the DOM, all HTML elements are defined as objects.
- The programming interface is the properties and methods of each object.
- A **property** is a value that you can get or set (like changing the content of an HTML element).
- A **method** is an action you can do (like add or deleting an HTML element).

HTML DOM document object

- The document object represents your web page.
- If you want to access any element in an HTML page, you always start with accessing the document object.
- Below are some examples of how you can use the document object to access and manipulate HTML.

Finding HTML Elements:

document.getElementById(*id*)	Find an element by element id
document.getElementsByTagName(*name*)	Find elements by tag name
document.getElementsByClassName(*name*)	Find elements by class name

HTML DOM document object

Changing HTML Elements:

Property	Description
element.innerHTML = *new html content*	Change the inner HTML of an element
element.attribute = new value	Change the attribute value of an HTML element
element.style.*property = new style*	Change the style of an HTML element
Method	Description
element.setAttribute*(attribute, value)*	Change the attribute value of an HTML element

Adding and Deleting Elements:

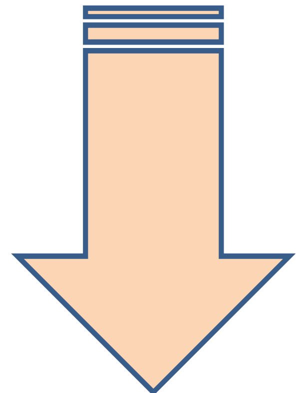
Method	Description
document.createElement(*element*)	Create an HTML element
document.removeChild(*element*)	Remove an HTML element
document.appendChild(*element*)	Add an HTML element
document.replaceChild(*new, old*)	Replace an HTML element

HTML DOM document object

Adding Events Handlers:

Method	Description
document.getElementById(*id*).onclick = function(){*code*}	Adding event handler code to an onclick event

VDOM



VDOM

DOM drawback:

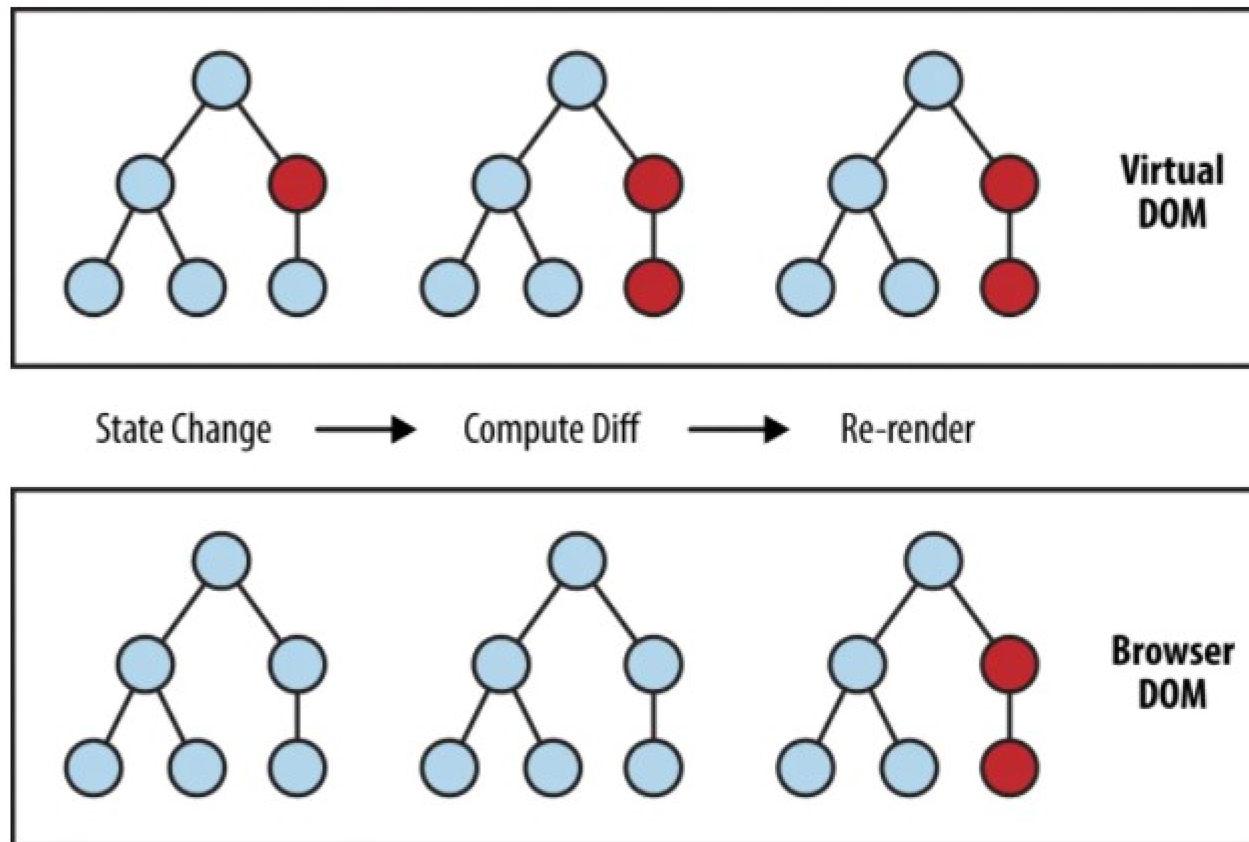
- The DOM is represented as a tree data structure. Because of that, the changes and updates to the DOM are fast. But after the change, the updated element and it’s children have to be re-rendered to update the application UI. The re-rendering or re-painting of the UI is what makes it slow. Therefore, the more UI components you have, the more expensive the DOM updates could be, since they would need to be re-rendered for every DOM update.

- **React creates a VIRTUAL DOM in memory.**
- Instead of manipulating the browser's DOM directly, React creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM.
- React only changes what needs to be changed!
- React finds out what changes have been made, and changes only what needs to be changed.

Virtual DOM is fast:

How VDOM is faster?

- When new elements are added to the UI, a virtual DOM, which is represented as a tree is created. Each element is a node on this tree. If the state of any of these elements changes, a new virtual DOM tree is created. This tree is then compared or "diffed" with the previous virtual DOM tree.
- Once this is done, the virtual DOM calculates the best possible method to make these changes to the real DOM. This ensures that there are minimal operations on the real DOM. Hence, reducing the performance cost vof updating the real DOM.



VDOM

- The red circles represent the nodes that have changed. These nodes represent the UI elements that have had their state changed. The difference between the previous version of the virtual DOM tree and the current virtual DOM tree is then calculated. The whole parent subtree then gets re-rendered to give the updated UI. This updated tree is then batch updated to the real DOM.

- In React every UI piece is a component, and each component has a state. React follows the observable pattern and listens for state changes. When the state of a component changes, React updates the virtual DOM tree. Once the virtual DOM has been updated, React then compares the current version of the virtual DOM with the previous version of the virtual DOM. This process is called “diffing”.

- Once React knows which virtual DOM objects have changed, then React updates only those objects, in the real DOM. This makes the performance far better when compared to manipulating the real DOM directly. **This makes React standout as a high performance JavaScript library.**

VDOM -- React render() function

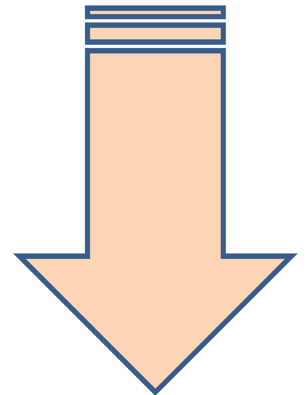
- render() is where the UI gets updated and rendered.
- render() function is the point of entry where the tree of React elements are created. When a state or prop within the component is updated, the render() will return a different tree of React elements. If you use setState() within the component, React immediately detects the state change and re-renders the component.
- React then figures out how to efficiently update the UI to match the most recent tree changes.
- This is when React updates its virtual DOM first and updates only the object that have changed in the real DOM.

Batch Update:

- React follows a batch update mechanism to update the real DOM. Hence, leading to increased performance. This means that updates to the real DOM are sent in batches, instead of sending updates for every single change in state.

- The repainting of the UI is the most expensive part, and React efficiently ensures that the real DOM receives only batched updates to repaint the UI.

JSON



JSON (Java Script Object Notation)

- JSON (JavaScript Object Notation) is a lightweight data-interchange format.
- It is easy for humans to read and write. It is easy for machines to parse and generate.
- It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999.
- JSON is a string format.
- JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others.
- These properties make JSON an ideal data-interchange language.

JSON string:

'{"name":"John", "age":30, "car":null}'

Keys and values are separated by a colon.

Keys must be strings, and values must be a valid JSON data type:

- string
- number
- object
- array
- boolean
- Null

JSON values **cannot** be one of the following data types:
•a function
•a date
•*undefined*

JavaScript objects can be created from JSON string: myObj = {"name":"John", "age":30,

JSON

- **Json strings:** Strings in JSON must be written in double quotes.

  {"name":"John"}
- **Numbers** in JSON must be an integer or a floating point.

  {"age":30}
- **Values** in JSON can be objects.

```
{
        "employee":{"name":"John", "age":30, "city":"New York"}
}
```

Json Arrays:

```
{
"employee":{"name":"John", "age":30, "city":"New York"}
}
```

Json Booleans: Values in JSON can be true/false.

{"sale":true}

Json null: Values in JSON can be null.

{"middlename":null}

JSON (Java Script Object Notation)

JavaScript objects can be created from JSON string:

```
myObj = {"name":"John", "age":30, "car":null};
```

(OR)

```
myJSON = '{"name":"John", "age":30, "car":null}';
myObj = JSON.Parse(myJSON);
```

Accessing Object Values: using either . or [ ]

```
const myJSON = '{"name":"John", "age":30, "car":null}';
const myObj = JSON.parse(myJSON);
x = myObj.name;
```

Using [ ] accessing object values:

```
const myJSON = '{"name":"John", "age":30, "car":null}';
const myObj = JSON.parse(myJSON);
x = myObj["name"];
```

Parsing JSON:

```
const obj = JSON.parse('{"name":"John", "age":30, "city":"New York"}’);
```

Array as JSON:

```
const text = '["Ford", "BMW", "Audi", "Fiat"]’;
const myArr = JSON.parse(text);
```

JSON vs XML

- Both JSON and XML can be used to receive data from a web server.
- The following JSON and XML examples both define an employees object, with an array of 3 employees:

JSON example:

```
{"employees":[
  { "firstName":"John", "lastName":"Doe" },
  { "firstName":"Anna", "lastName":"Smith" },
  { "firstName":"Peter", "lastName":"Jones" }
]}
```

XML example:

```
<employees>
  <employee>
    <firstName>John</firstName> <lastName>Doe</lastName>
  </employee>
  <employee>
    <firstName>Anna</firstName> <lastName>Smith</lastName>
  </employee>
  <employee>
    <firstName>Peter</firstName> <lastName>Jones</lastName>
  </employee>
</employees>
```

JSON vs XML

- Both JSON and XML are "self describing" (human readable)
- Both JSON and XML are hierarchical (values within values)
- Both JSON and XML can be parsed and used by lots of programming languages
- Both JSON and XML can be fetched with an XMLHttpRequest

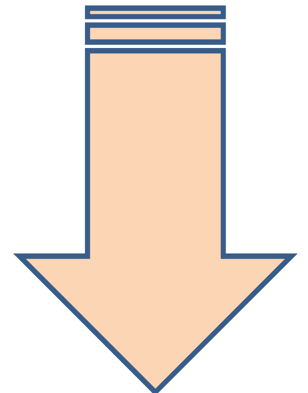
JSON is Unlike XML Because:

- JSON doesn't use end tag
- JSON is shorter
- JSON is quicker to read and write
- JSON can use arrays

✓ XML is much more difficult to parse than JSON.

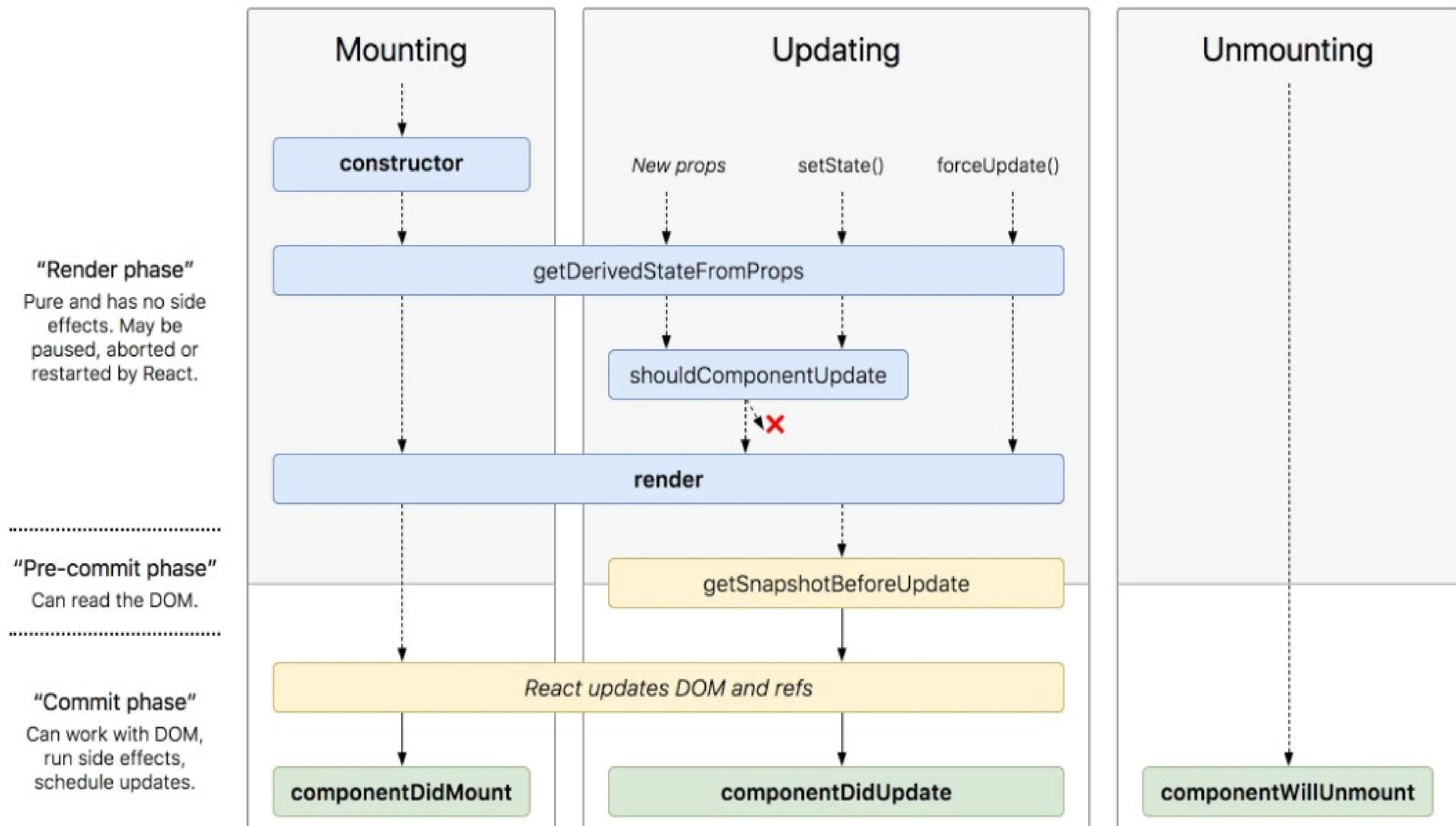
✓ JSON is parsed into a ready-to-use JavaScript object.

Component Lifecycle

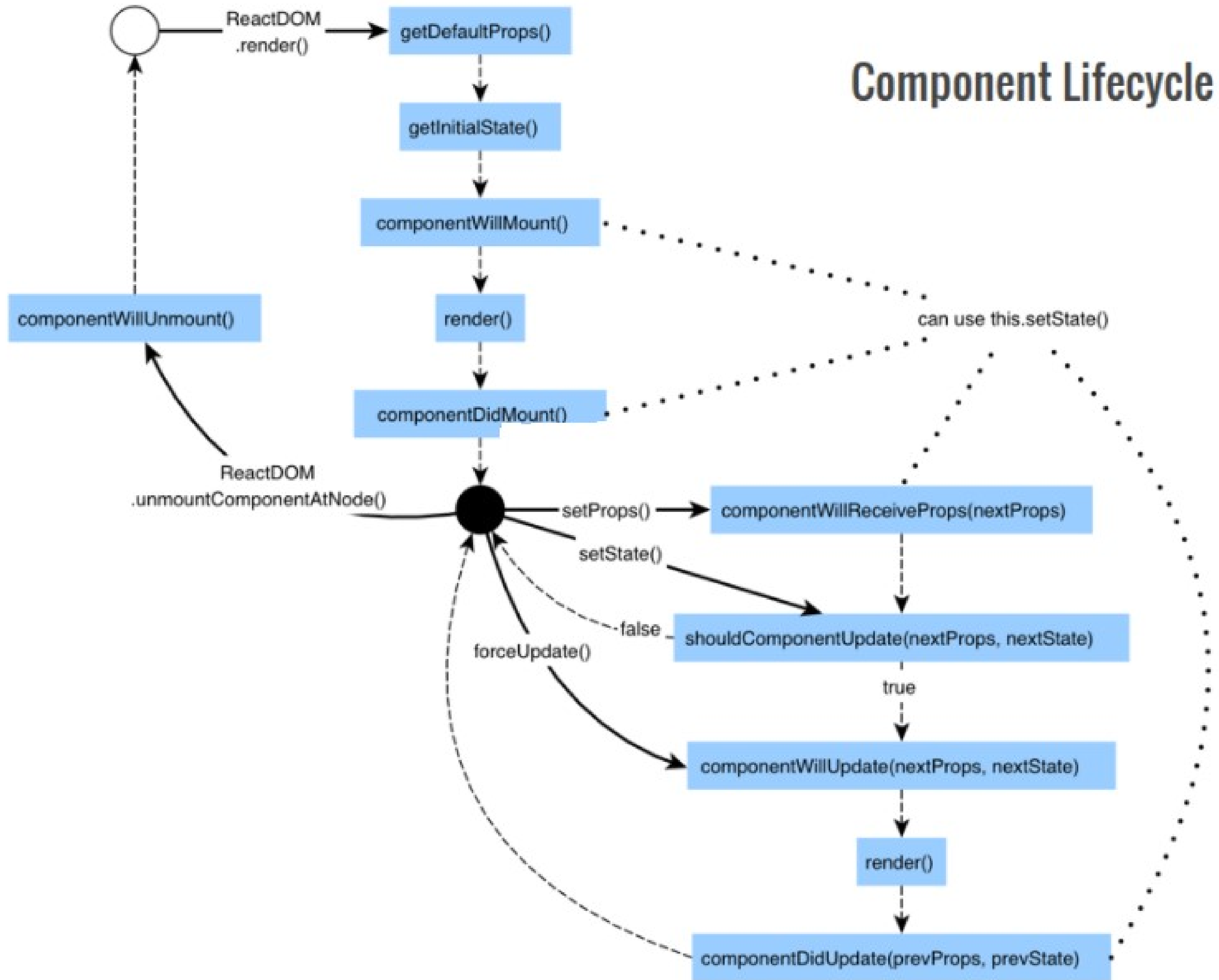


Component Lifecycle

- lifecycle of a component can be defined as the series of methods that are invoked in different stages of the component's existence.
- Each component in React has a lifecycle which you can monitor and manipulate during **its three main phases.**
  – The three phases are: Mounting, Updating, and Unmounting.



Component Lifecycle



Component Lifecycle

Mounting phase: Mounting means putting elements into the DOM.

React has **four built-in methods** that gets called, in this order, when mounting a component:

1. constructor()
2. getDerivedStateFromProps()
3. render()
4. componentDidMount()

Updating phase: five built-in methods

- A component is updated whenever there is a change in the component's state or props.
- React has **five built-in methods** that gets called, in this order, when a component is updated:

1. getDerivedStateFromProps()
2. shouldComponentUpdate()
3. render()
4. getSnapshotBeforeUpdate()
5. componentDidUpdate()

Component Lifecycle – unmounting phase

Unmounting phase: one built in method

- The next phase in the lifecycle is when a component is removed from the DOM, or unmounting as React likes to call it.
- React has only one built-in method that gets called when a component is unmounted.

componentWillUnmount():

- The componentWillUnmount method is called when the component is about to be removed from the DOM.

Component Lifecycle

- Each component in React has a lifecycle which you can monitor and manipulate during **its three main phases.**
  - The three phases are: Mounting, Updating, and Unmounting.

Mounting: Mounting means putting elements into the DOM.

React has **four built-in methods** that gets called, in this order, when mounting a component:

1. constructor()
2. getDerivedStateFromProps()
3. render()
4. componentDidMount()

The render() method is required and will always be called, the others are optional and will be called if you define them.

constructor( ):

- The **constructor()** method is called before anything else, when the component is initiated, and it is the natural place to set up the initial state and other initial values.
- The **constructor()** method is called with the props, as arguments, and you should always start by calling the super(props) before anything else, this will initiate the parent's constructor method and allows the component to inherit methods from its parent (React.Component).

Mounting – constructor( ) method

Example:

```
class Header extends React.Component {
  constructor(props) {
    super(props);  //calling base class
    this.state = {favoritecolor: "red"};   //setting initial state
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

ReactDOM.render(<Header />, document.getElementById('root'));
```

Mounting - getDerivedStateFromProps( )

getDerivedStateFromProps( ):

- The getDerivedStateFromProps() method is called right before rendering the element(s) in the DOM.
- This is the natural place to set the state object based on the initial props.
- It takes state as an argument, and returns an object with changes to the state.
- The getDerivedStateFromProps method is called right before the render method

The example below starts with the favorite color being "red", but the getDerivedStateFromProps() method updates the favorite color based on the favcol attribute:

Mounting - getDerivedStateFromProps( )

The example below starts with the favorite color being "red", but the getDerivedStateFromProps() method updates the favorite color based on the favcol attribute:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol };
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );  }}
ReactDOM.render(<Header favcol="yellow"/>, document.getElementById('root'));
```

Mounting – render() method

- The render() method is required, and is the method that actually outputs the HTML to the DOM.

A simple component with a simple render() method:

```
class Header extends React.Component {
  render() {
    return (
      <h1>This is the content of the Header component</h1>
    );
  }
}

ReactDOM.render(<Header />, document.getElementById('root'));
```

Mounting – componentDidMount( ) method

- The componentDidMount() method is called after the component is rendered.
- This is where you run statements that requires that the component is already placed in the DOM .
- At first my favorite color is red, but give me a second, and it is yellow instead:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)  }
  render() {
    return (<h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );  }}
ReactDOM.render(<Header />, document.getElementById('root'));
```

Updating Phase

- The next phase in the lifecycle is when a component is updated.
- A component is updated whenever there is a change in the component's state or props.
- React has **five built-in methods** that gets called, in this order, when a component is updated:

1. getDerivedStateFromProps()
2. shouldComponentUpdate()
3. render()
4. getSnapshotBeforeUpdate()
5. componentDidUpdate()

- The render() method is required and will always be called, the others are optional and will be called if you define them.

Updating Phase - getDerivedStateFromProps()

1. getDerivedStateFromProps()

- Also at updates the getDerivedStateFromProps method is called.
- This is the first method that is called when a component gets updated.
- This is still the natural place to set the state object based on the initial props.

Updating Phase - getDerivedStateFromProps()

- The example below has a button that changes the favorite color to blue, but since the getDerivedStateFromProps() method is called, which updates the state with the color from the favcol attribute, the favorite color is still rendered as yellow:

```
class Header extends React.Component {
 constructor(props) {
  super(props);     this.state = {favoritecolor: "red"};
 }
 static getDerivedStateFromProps(props, state) {
  return {favoritecolor: props.favcol };  }
 changeColor = () => {
  this.setState({favoritecolor: "blue"});  }
 render() {    return (<div><h1>My Favorite Color is {this.state.favoritecolor}</h1>
    <button type="button" onClick={this.changeColor}>Change color</button>
    </div>
  );  }}
ReactDOM.render(<Header favcol="yellow"/>, document.getElementById('root'));
```

Updating Phase - shouldComponentUpdate()

2. shouldComponentUpdate()

- In the shouldComponentUpdate() method you can return a Boolean value that specifies whether React should continue with the rendering or not. The default value is true.

Stop the component from rendering at any update:

Updating Phase - shouldComponentUpdate()

Stop the component from rendering at any update:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};  }
  shouldComponentUpdate() {
    return false;  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});  }
  render() {
    return (
      <div>     <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
      </div>    );  }}
ReactDOM.render(<Header />, document.getElementById('root'));
```

Updating Phase - render()

render():

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  changeColor = () => {
    this.setState({favoritecolor: "blue"});
  }
  render() {
    return (
      <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <button type="button" onClick={this.changeColor}>Change color</button>
      </div>
    );  }}
ReactDOM.render(<Header />, document.getElementById('root'));
```

Updating Phase - getSnapshotBeforeUpdate()

1. **getSnapshotBeforeUpdate()**
2. in the getSnapshotBeforeUpdate() method you have access to the props and state before the update, meaning that even after the update, you can check what the values were before the update.

3. **If the getSnapshotBeforeUpdate() method is present, you should also include the componentDidUpdate() method, otherwise you will get an error.**

4. getSnapshotBeforeUpdate(prevProps, prevState)

   Parameters: It accepts two parameters, they are *prevProps* and *prevState* which are just the props or state before the component in question is re-rendered.

Updating Phase - getSnapshotBeforeUpdate()

```
class Header extends React.Component {
 constructor(props) {
   super(props);
   this.state = {favoritecolor: "red"};   }
 componentDidMount() {
   setTimeout(() => {
     this.setState({favoritecolor: "yellow"})
   }, 1000)  }
 getSnapshotBeforeUpdate(prevProps, prevState)
{
   document.getElementById("div1").innerHTML =
   "Before the update, the favorite was " + prevState.favoritecolor;
}
 componentDidUpdate() {
   document.getElementById("div2").innerHTML =
   "The updated favorite is " + this.state.favoritecolor;
 }
```

Contd….

Updating Phase - getSnapshotBeforeUpdate() contd….

```
render() {
  return (
    <div>
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
      <div id="div1"></div>
      <div id="div2"></div>
    </div>
  );
}
```

Updating Phase - componentDidUpdate()

The componentDidUpdate method is called after the component is updated in the DOM.

Example:

The example below might seem complicated, but all it does is this:

When the component is mounting it is rendered with the favorite color "red".

When the component has been mounted, a timer changes the state, and the color becomes "yellow".

This action triggers the update phase, and since this component has a componentDidUpdate method, this method is executed and writes a message in the empty DIV element:

Updating Phase - componentDidUpdate()

```
class Header extends React.Component {
  constructor(props) {    super(props);    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})    }, 1000)  }
  componentDidUpdate() {
    document.getElementById("mydiv").innerHTML =
    "The updated favorite is " + this.state.favoritecolor;
  }
  render() {
    return (        <div>        <h1>My Favorite Color is {this.state.favoritecolor}</h1>
<div id="mydiv"></div>      </div>    );  }
}
ReactDOM.render(<Header />, document.getElementById('root'));
```

Unmounting phase - componentWillUnmount() method

- The next phase in the lifecycle is when a component is removed from the DOM, or unmounting as React likes to call it.
- React has only one built-in method that gets called when a component is unmounted.

componentWillUnmount():

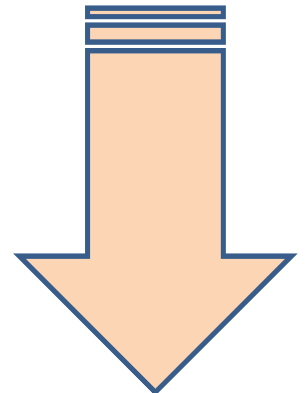
- The componentWillUnmount method is called when the component is about to be removed from the DOM.

Unmounting phase - componentWillUnmount() method

Example:

```
class Container extends React.Component {
  constructor(props) {    super(props);    this.state = {show: true};
  }
  delHeader = () => {    this.setState({show: false});
  }
  render() {
    let myheader;
    if (this.state.show) {
      myheader = <Child />;
    };
    return (<div> {myheader}  <button type="button" onClick={this.delHeader}>Delete
Header</button>      </div>    );  }  }
class Child extends React.Component {
  componentWillUnmount() {
    alert("The component named Header is about to be unmounted.");
  }
  render() {    return (      <h1>Hello World!</h1>    );  }}
ReactDOM.render(<Container />, document.getElementById('root'));
```

Data & Data flow in React



- Mutable and immutable state
- Stateful and stateless components
- Component communication
- One-way data flow

State: All the information a program has access to at a given instant in time.

Example:

```
const letters = 'Letters';
const splitLetters = letters.split('');
console.log("Let's spell a word!");
splitLetters.forEach(letter => console.log(letter));
```

Mutable and immutable state:

- In React applications, there are two primary ways that you can work with state in components: through state that you can change, and through state that you shouldn't.
- state and props are mutable and immutable respectively.
- In React components, state is generally mutable. props will not change.

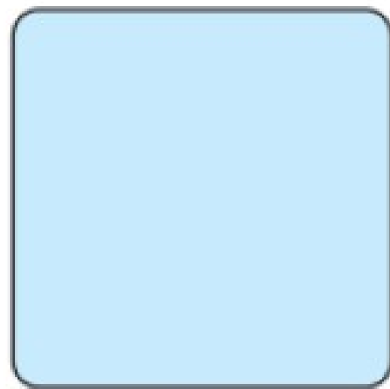
- we call state mutable we mean we can overwrite or update that data (for example, a variable that you can overwrite). Immutable state, on the other hand, can't be changed.
- There are also immutable data structures, which can be changed but only in controlled ways (this is sort of how the state API works in React).

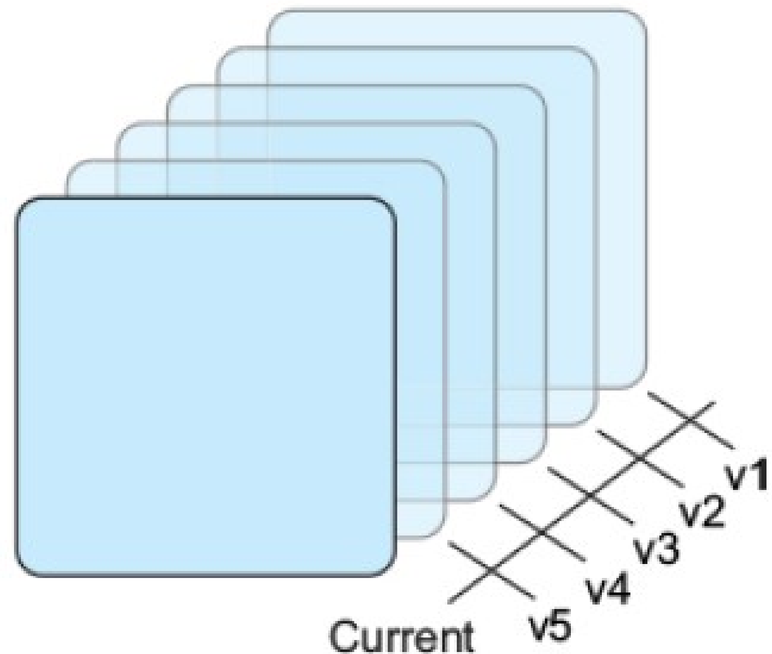
***Immutable*—** An immutable, persistent data structure supports multiple versions over time but can't be directly overwritten; immutable data structures are generally persistent.

***Mutable*—** A mutable, ephemeral data structure supports only a single version over time; mutable data structures are overwritten when they change and don't support additional versions.

Persistence and ephemerality in immutable and mutable data structures.

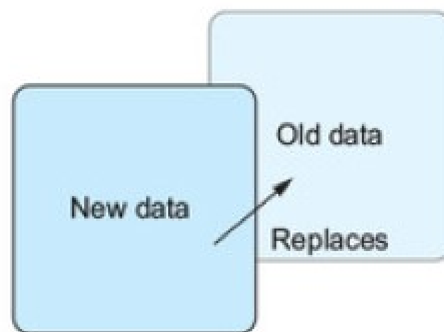
- Ephemeral (volatile) data structures only have the capacity to store a moment's worth of data, whereas persistent data structures can keep track of changes over time. This is where the immutability of immutable data structures becomes clearer: only copies of state are made—they're not replaced

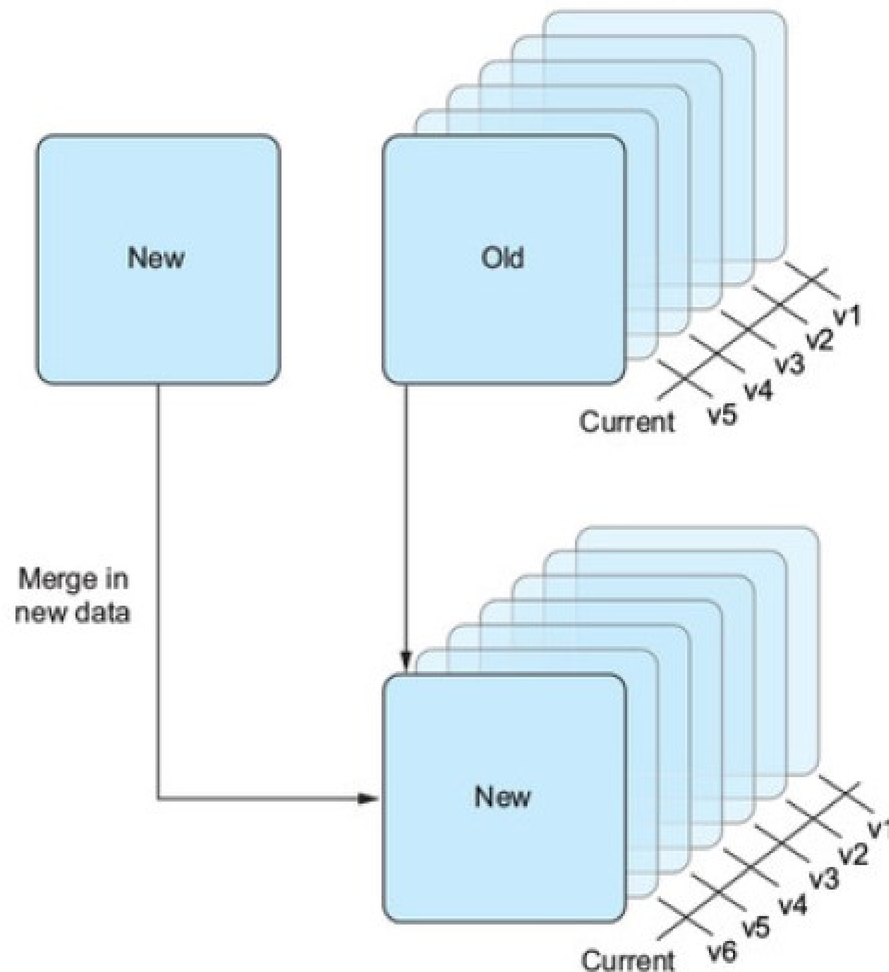




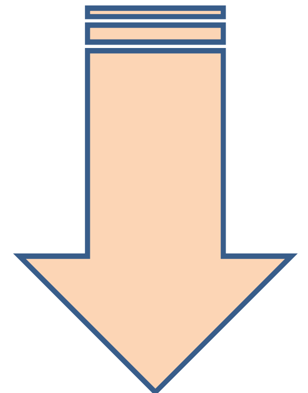
Handling changes with mutable and immutable data

- Immutable or persistent data structures usually record a history and don't change but rather make versions of what changed over time.
- Mutable (Ephemeral data structures), on the other hand, usually don't record history and get wiped out with each update.





defaultProps



defaultProps

- The defaultProps is a React component property that allows you to set default values for the props argument.
- If the prop property is passed, it will be changed.
- The defaultProps can be defined as a property on the component class itself, to set the default props for the class.

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

class App extends Component {
  render() {
    return (
      <div >
        <Person name="kapil" eyeColor="blue" age="23"></Person>
        <Person name="Sachin" eyeColor="blue" ></Person>
        <Person name="Nikhil" age="23"></Person>
        <Person eyeColor="green" age="23"></Person>
      </div>
    );
  }
}
```

defaultProps

```
 class Person extends Component {
  render() {
    return (
      <div>
        <p> Name: {this.props.name} </p>
        <p>EyeColor: {this.props.eyeColor}</p>
        <p>Age : {this.props.age} </p>
      </div>
    )
  }
}

Person.defaultProps = {
  name: "Rahul",
  eyeColor: "deepblue",
  age: "45"
}

export default App;
```

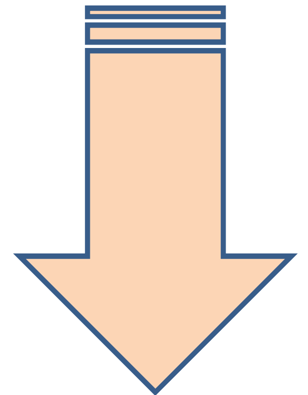
defaultProps with function component

```
import React from 'react';

function App(props) {
  return (
    <div >
      <Person name="kapil" eyeColor="blue" age="23"></Person>
      <Person name="Sachin" eyeColor="blue" ></Person>
      <Person name="Nikhil" age="23"></Person>
      <Person eyeColor="green" age="23"></Person>
    </div>
  );
}
```

defaultProps with function component

```
function Person(props) {
  return (
    <div>
      <p> Name: {props.name} </p>
      <p>EyeColor: {props.eyeColor}</p>
      <p>Age : {props.age} </p>
      <hr></hr>
    </div>
  )
}

Person.defaultProps = {
  name: "Rahul",
  eyeColor: "deepblue",
  age: "45"
}

export default App;
```

PropTypes



PropTypes

- You can catch a lot of bugs with typechecking
- React has some built-in typechecking abilities. To run typechecking on the props for a component, you can assign the special propTypes property.
- PropTypes is React's internal mechanism for adding type checking to components.
- React components use a special property named propTypes to set up type checking.
- If default props are set for the React component, the values are first resolved before type checking against propTypes. Therefore, default values are also subject to the prop type definitions.
- propTypes type checking only happens in development mode, enabling you to catch bugs in your React application while developing. For performance reasons, it is not triggered in the production environment.
- npm install prop-types --save → to install prop-types package

Example:

```
import PropTypes from 'prop-types';
class Greeting extends React.Component {
  render() {    return (
      <h1>Hello, {this.props.name}</h1>
    );  }  }
Greeting.propTypes = {
  name: PropTypes.string
```

PropTypes - example

```
class People extends React.Component {
  // ...component class body here
}
People.propTypes = {
  label: Proptypes.string,
  score: Proptypes.number
}
```

PropTypes Validators

The PropTypes utility exports a wide range of validators for configuring type definitions.

Different types of validators are:

- Basic types
- Renderable types
- instance types
- Multiple types
- Collection types
- required prop types.

PropTypes Validators

Basic types:

➢ PropTypes.any	The prop can be of any data type
➢ PropTypes.bool	The prop should be a Boolean
➢ PropTypes.number	The prop should be a number
➢ PropTypes.string	The prop should be a string
➢ PropTypes.func	The prop should be a function
➢ PropTypes.array	The prop should be an array
➢ PropTypes.object	The prop should be an object
➢ PropTypes.symbol	The prop should be a symbol

Renderable Types:

PropTypes also exports the following validators for ensuring that the value passed to a prop can be rendered by React.

PropTypes.node: The prop should be anything that can be rendered by React — a number, string, element, or array (or fragment) containing these types

PropTypes.element: The prop should be a React element

PropTypes Validators

```
Component.propTypes = {
 nodeProp: PropTypes.node,
 elementProp: PropTypes.element
}
```

- One common usage of the PropTypes.element validator is in ensuring that a component has a single child. If the component has no children or multiple children, a warning is displayed on the JavaScript console.

- Component.propTypes = {

```
 children: PropTypes.element.isRequired
}
```

Prop validators

PropTypes – InstanceTypes:

In cases where you require a prop to be an instance of a particular JavaScript class, you can use the PropTypes.instanceOf validator.

```
Component.propTypes = {
  personProp: PropTypes.instanceOf(Person)
}
```

Multiple types:

PropTypes.oneOf: The prop is limited to a specified set of values, treating it like an enum

PropTypes.oneOfType: The prop should be one of a specified set of types, behaving like a union of types

```
Component.propTypes = {
  enumProp: PropTypes.oneOf([true, false, 0, 'Unknown']),
  unionProp: PropTypes.oneOfType([    PropType.bool,
    PropType.number,
    PropType.string,
    PropType.instanceOf(Person)
  ])
}
```

Prop validators – collection types

PropTypes – CollectionTypes:

Besides thePropTypes.array and PropTypes.object validators, PropTypes also provides validators for more fine-tuned validation of arrays and objects.

PropTypes.arrayOf

PropTypes.arrayOf ensures that the prop is an array in which all items match the specified type.

```
Component.propTypes = {
 peopleArrayProp: PropTypes.arrayOf(
  PropTypes.instanceOf(Person)
 ),
 multipleArrayProp: PropTypes.arrayOf(
  PropTypes.oneOfType([
   PropType.number,
   PropType.string
  ])
 ) }
```

Prop validators – collection types

PropTypes.objectOf

PropTypes.objectOf ensures that the prop is an object in which all property values match the specified type.

```
Component.propTypes = {
  booleanObjectProp: PropTypes.objectOf(
    PropTypes.bool
  ),
  multipleObjectProp: PropTypes.objectOf(
    PropTypes.oneOfType([
      PropType.func,
      PropType.number,
      PropType.string,
      PropType.instanceOf(Person)
    ])
  )
}
```

Prop validators – collection types

PropTypes.shape:

You can use PropTypes.shape when a more detailed validation of an object prop is required. It ensures that the prop is an object that contains a set of specified keys with values of the specified types.

```
Component.propTypes = {
  profileProp: PropTypes.shape({
    id: PropTypes.number,
    fullname: PropTypes.string,
    gender: PropTypes.oneOf(['M', 'F']),
    birthdate: PropTypes.instanceOf(Date),
    isAuthor: PropTypes.bool
  })
}
```

PropTypes.exact:

For strict (or exact) object matching, you can use PropTypes.exact as follows:

```
Component.propTypes = {
  subjectScoreProp: PropTypes.exact({
    subject: PropTypes.oneOf(['Maths', 'Arts', 'Science']),
    score: PropTypes.number
  })
```

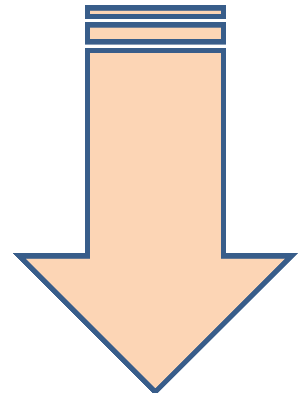
Prop validators – collection types

Required types:

you can chain isRequired to any prop validator to ensure that a warning is shown whenever the prop is not provided.

```
Component.propTypes = {
  requiredAnyProp: PropTypes.any.isRequired,
  requiredFunctionProp: PropTypes.func.isRequired,
  requiredSingleElementProp: PropTypes.element.isRequired,
  requiredPersonProp: PropTypes.instanceOf(Person).isRequired,
  requiredEnumProp: PropTypes.oneOf(['Read', 'Write']).isRequired,

  requiredShapeObjectProp: PropTypes.shape({
    title: PropTypes.string.isRequired,
    date: PropTypes.instanceOf(Date).isRequired,
    isRecent: PropTypes.bool
  }).isRequired

}
```

Parent Child components communication



Parent to Child communication

- When you need to pass data from a parent to child class component, you do this by using props.
- For example, let's say you have two class components, Parent and Child, and you want to pass a state in the parent to the child. You would do something like this:

```
import React from 'react';
class Parent extends React.Component{
  constructor(props){  super(props);
    this.state = {          data: 'Data from parent'       }
  }
  render(){
    const {data} = this.state;
    return(          <div>              <Child dataParentToChild = {data}/>           </div>       )    }}

class Child extends React.Component{
  constructor(props){        super(props);
    this.state = {           data: this.props.dataParentToChild        }    }
  render(){
    const {data} = this.state;
    return(          <div>              {data}          </div>       )    }}

export default Parent;
```

Child to Parent communication

Steps:

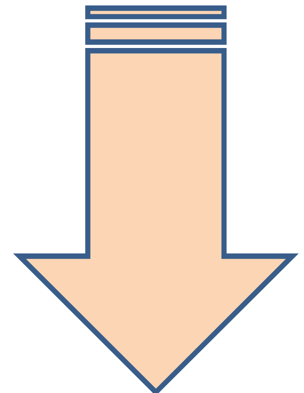
1. Create a callback function in the parent component. This callback function will get the data from the child component.
2. Pass the callback function in the parent as a prop to the child component.
3. The child component calls the parent callback function using props.

- When the Child component is triggered, it will call the Parent component's callback function with data it wants to pass to the parent. The Parent's callback function will handle the data it received from the child.

Child to Parent communication

```
class Parent extends React.Component{
   constructor(props){
      super(props);
      this.state = {
         data: null
      }   }
   handleCallback = (childData) =>{
      this.setState({data: childData})
   }
   render(){
      const {data} = this.state;
      return(<div> <Child parentCallback = {this.handleCallback}/>   {data} </div>
      )   }}
class Child extends React.Component{
   onTrigger = (event) => {
      this.props.parentCallback("Data from child");
      event.preventDefault();   }
   render(){return(<div>  <form onSubmit = {this.onTrigger}>
            <input type = "submit" value = "Submit"/>    </form> </div> )  }}
export default Parent;
```

Data Binding –
One Way, Two way



Data Binding in React.js applications

- ✓ **Data Binding** is the process of connecting the view element or user interface, with the data which populates it.
- ✓ In **ReactJS**, components are rendered to the user interface and the component's logic contains the data to be displayed in the view(UI). The connection between the data to be displayed in the view and the component's logic is called data binding in ReactJS.
- **One-way Data Binding:** ReactJS uses one-way data binding. In one-way data binding one of the following conditions can be followed:

  Component to View: Any change in component data would get reflected in the view.

  View to Component: Any change in View would get reflected in the component's data.

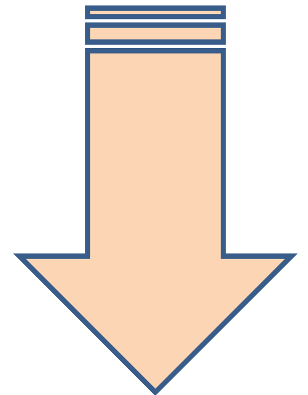
useState( )

- ✓ The **useState** hook is a special function that takes the initial state as an argument and returns an array of two entries. UseState encapsulate only singular value from the state, for multiple state need to have useState calls.

  const [state, setState] = useState(initialstate)

- **Syntax:** The first element is the **initial state** and the second one is a **function** that is used for updating the state.
- const [state, setState] = useState(initialstate)We can also pass a function as an argument if the initial state has to be computed. And the value returned by the function will be used as the initial state.
- const [sum, setsum] = useState(function generateRandomInteger(){5+7);})The above function is one line function which computes the sum of two numbers and will be set as the initial state.

Forms



Forms

- Just like in HTML, React uses forms to allow users to interact with the web page.
- Handling forms is about how you handle the data when it changes value or gets submitted.
- In HTML, form data is usually handled by the DOM. In React, form data is usually handled by the components.
- When the data is handled by the components, all the data is stored in the component state.
- You can control changes by adding event handlers in the **onChange** attribute

Forms

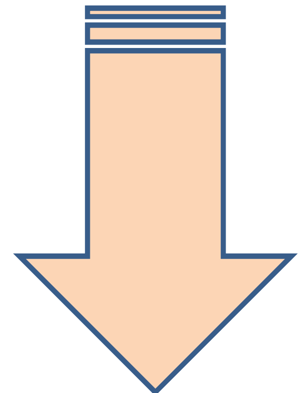
```
class MyForm extends React.Component {
 constructor(props) {
  super(props);
  this.state = { username: '' };
 }
 myChangeHandler = (event) => {
  this.setState({username: event.target.value});
 }
 render() {   return (
    <form>
          <h1>Hello {this.state.username}</h1>
          <p>Enter your name:</p>
          <input type='text’ onChange={this.myChangeHandler} />
    </form>
  );  }
}
ReactDOM.render(<MyForm />, document.getElementById('root'));
```

- You must initialize the state in the constructor method before you can use it. You get access to the field value by using the **event.target.value** syntax.

Forms – Multiple Input fields

- You can control the values of more than one input field by adding a **name attribute** to each element.
- When you initialize the state in the constructor, use the field names.
- To access the fields in the event handler use the **event.target.name** and **event.target.value** syntax.
- To update the state in the this.setState method, use square brackets [bracket notation] around the property name.

Integrating Third Party Libraries



Integrating Third Party Libraries

- The good part of using this third party library is it boost the application development process and helps to gain the goal.
- You’ll build React applications in a context that involves nonReact libraries that also work with the DOM. These might include things like jQuery, jQuery plugins, or even other frontend frameworks.
- We’ve seen that React manages the DOM for you and that this can simplify how you think about user interfaces.
- There are many libraries that are written in plain Javascript or as a JQuery plugin, an example is Datatable.js. There is no need to reinvent the wheel, consume a lot of time and energy, and re-create those libraries.
- **Third-party libraries can be integrated with class components,** also **with functional components using Hooks.**
- A React.js component may update the DOM elements multiple times during its lifecycle after component props or states update.
- Some libraries need to know when the DOM is updated. Some other libraries need to prevent the DOM elements from updating.
- **Datatables.js is a free JQuery plugin** that adds advanced controls to HTML tables like searching, sorting, and pagination.
- **Refs:** React provides a way for developers to access DOM elements or other React elements. Refs are very handy when integrating with third-party libraries.

Integrating Third Party Libraries

- We need to know some lifecycle methods. These lifecycle methods are important for initializing other libraries, destroying components, subscribing and unsubscribing events.

React Lifecycle methods required to know:

We need to know some lifecycle methods. These lifecycle methods are important for initializing other libraries, destroying components, subscribing and unsubscribing events

1. **componentDidMount**

it is fired when the element is mounted on the DOM. It is like jquery's $(document).ready().

Usage:

- ✓ fetching data from the server.
- ✓ initializing third-party libraries.

Example:

```
componentDidMount() {
  this.$el = $(this.el);
  this.currentTable = this.$el.DataTable({});
}
```

Integrating Third Party Libraries

2. componentDidUpdate

it is fired when the props passed to the component are updated or the method this.setState is called to change the state of the component. This method is not called for the initial render().

Usage:

- ✓ reload third-party library if props is updated.

Example:

```
componentDidUpdate(prevProps) {
  if (prevProps.children !== this.props.children) {
    // update third-party library based on prop change
  }
}
```

3. componentWillUnmount: it is fired before the React component is destroyed and unmounted on the DOM.

Usage:

- ✓ Unsubscribing from events
- ✓ Destroying third-party library

Integrating Third Party Libraries

4. shouldComponentUpdate: it is used to avoid the React component from re-rendering. It prevents to update the DOM even if the state or props are updated.

Usage:

Some libraries require an un-changeable DOM.

Example:

```
shouldComponentUpdate() {
  return false;
}
```

Integrating Third Party Libraries - Jquery Datatables

- Datatables.js is a free JQuery plugin that adds advanced controls to HTML tables like searching, sorting, and pagination.

Steps:

1. Need to install a couple of dependencies from npm: jquery and datatables.net

npm i -S jquery datatables.net

2. Add a link to DataTable.css file in index.html.

<link rel="stylesheet" href="https://cdn.datatables.net/1.10.23/css/jquery.dataTables.min.css" />

3. Create a class component named DataTable inside

components/DataTable.js.

4. Import the libraries:

var $ = require("jquery");

$.DataTable = require("datatables.net");

5. Inside the render() method, we need to have a table element with a ref. It looks like an html ID, we use it for selecting (referencing) it.

Integrating Third Party Libraries - Jquery Datatables

6. We need to **render children props inisde the tbody** which is passed by the parent element.

```
render() {
  return (
    <table ref={(el) => (this.el = el)}>
      <thead>
        <tr>
          <th>#</th>
          <th>Title</th>
          <th>Completed</th>
          <th></th>
        </tr>
      </thead>
      <tbody>{this.props.children}</tbody>
    </table>
  );
}
```

Integrating Third Party Libraries

7. Inside the **componentDidMount**() method, we need to get the ref and call jquery method DataTable()

```
componentDidMount() {
  this.$el = $(this.el);
  this.currentTable = this.$el.DataTable();
}
```

8. Inside the **componentDidUpdate**(prevProps), we refresh the datatable by calling ajax.reload() when the props are updated. According to datatable.js, this method refreshes the table.

```
componentDidUpdate(prevProps) {
  // It means that only when props are updated
  if (prevProps.children !== this.props.children) {
    this.currentTable.ajax.reload();
  }
}
```

9. Finally, inside **componentWillUnmount**() we destroy the table.

```
componentWillUnmount() {
  this.currentTable.destroy();
}
```

Integrating Third Party Libraries

10. Using the DataTable component in our react application.

```
import React from "react";
import DataTable from "./components/DataTable";

class App extends React.Component {
  state = {
    todos: [],
  };
  componentDidMount() {
    fetch("https://jsonplaceholder.typicode.com/todos")
      .then((res) => res.json())
      .then((data) =>
        this.setState({
          todos: data,
        })
      );
  }
```

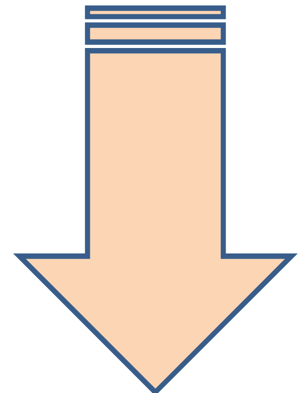
Integrating Third Party Libraries

```
 render() {
   return (
     <DataTable>
       {this.state.todos.map((todo) => (
        <tr key={todo.id}>
          <td>{todo.id}</td>
          <td>{todo.title}</td>
          <td>{todo.completed ? "Yes" : "No"}</td>
          <td>
            <button>Edit</button>
            <button>Delete</button>
          </td>
        </tr>
       ))}
     </DataTable>
   );
  }
}
export default App;
```

Integrating Third Party Libraries

-

Routing



Routing

- Routing is a process in which a user is directed to different pages based on their action or request. ReactJS Router is mainly used for developing Single Page Web Applications.
- Routing is the ability to move between different parts of an application when a user enters a URL or clicks an element (link, button, icon, image etc) within the application.
- To add routing capabilities, you will use the popular React-Router library. It's worth noting that this library has three variants:
  1. react-router: the core library
  2. react-router-dom: a variant of the core library meant to be used for web applications
  3. react-router-native: a variant of the core library used with react native in the development of Android and iOS applications.

- Both react-router-dom and react-router-native import all the functionality of the core react-router library.
- To install react-router-dom as part of the current project:

```
npm install --save react-router-dom
```

-

Routing

- The react-router package includes a number of routers that we can take advantage of depending on the platform we are targeting. These include BrowserRouter, HashRouter, and MemoryRouter.
- For the browser-based applications we are building, the BrowserRouter and HashRouter are a good fit.
- The **BrowserRouter** is used for applications which have a dynamic server that knows how to handle any type of URL
- The **HashRouter** is used for static websites with a server that only responds to requests for files that it knows about.

Routing

Example using BrowserRouter: In this example, the <App/> component is the child to the <BrowserRouter> and should be the only child. Now, the routing can happen anywhere within the <App/> component.

```
ReactDOM.render(
  <BrowserRouter>
    <App/>
  </BrowserRouter>,
  document.getElementById('root'));
```

React router

Steps:

1. **Install react-router in the project folder**

   C:\Users\username\Desktop\reactApp>npm install react-router

2. **Create Components**

In this step, we will create four components. The **App** component will be used as a tab menu. The other three components **(Home), (About) and (Contact)** are rendered once the route has changed

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, Route, Link, browserHistory, IndexRoute } from 'react-router'
class App extends React.Component {
  render() {
    return (<div>   <ul>          <li>Home</li>          <li>About</li>          <li>Contact</li>
        </ul>
        {this.props.children}
      </div>
    )
  }
}
export default App;
```

React router

Step2: contd…

```
class Home extends React.Component {
  render() {      return (
      <div>          <h1>Home...</h1>        </div>
    )  }}
export default Home;
class About extends React.Component {
  render() {
    return (       <div>          <h1>About...</h1>        </div>
    )  }}
export default About;
class Contact extends React.Component {
  render() {
    return (       <div>          <h1>Contact...</h1>        </div>
    )  }
}
export default Contact;
```

React router

Step 3: Add a Router

Now, we will add routes to the app. Instead of rendering App element like in the previous example, this time the Router will be rendered. We will also set components for each route.

```
ReactDOM.render((
  <Router history = {browserHistory}>
    <Route path = "/" component = {App}>
      <IndexRoute component = {Home} />
      <Route path = "home" component = {Home} />
      <Route path = "about" component = {About} />
      <Route path = "contact" component = {Contact} />
    </Route>
  </Router>
), document.getElementById('app’))
```

When the app is started, we will see three clickable links that can be used to change the route

<Link> component

- Sometimes, we want to need multiple links on a single page. When we click on any of that particular Link, it should load that page which is associated with that path without reloading the web page. To do this, we need to import **<Link> component** in the index js file.
- This component is used to create links which allow to navigate on different URLs and render its content without reloading the webpage.
- to add some styles to the Link, react router provides a new trick **NavLink** and style **activeStyle**. The activeStyle properties mean when we click on the Link, it should have a specific style so that we can differentiate which one is currently active.
- Use NavLink instead of Link.

React Router – Example 2

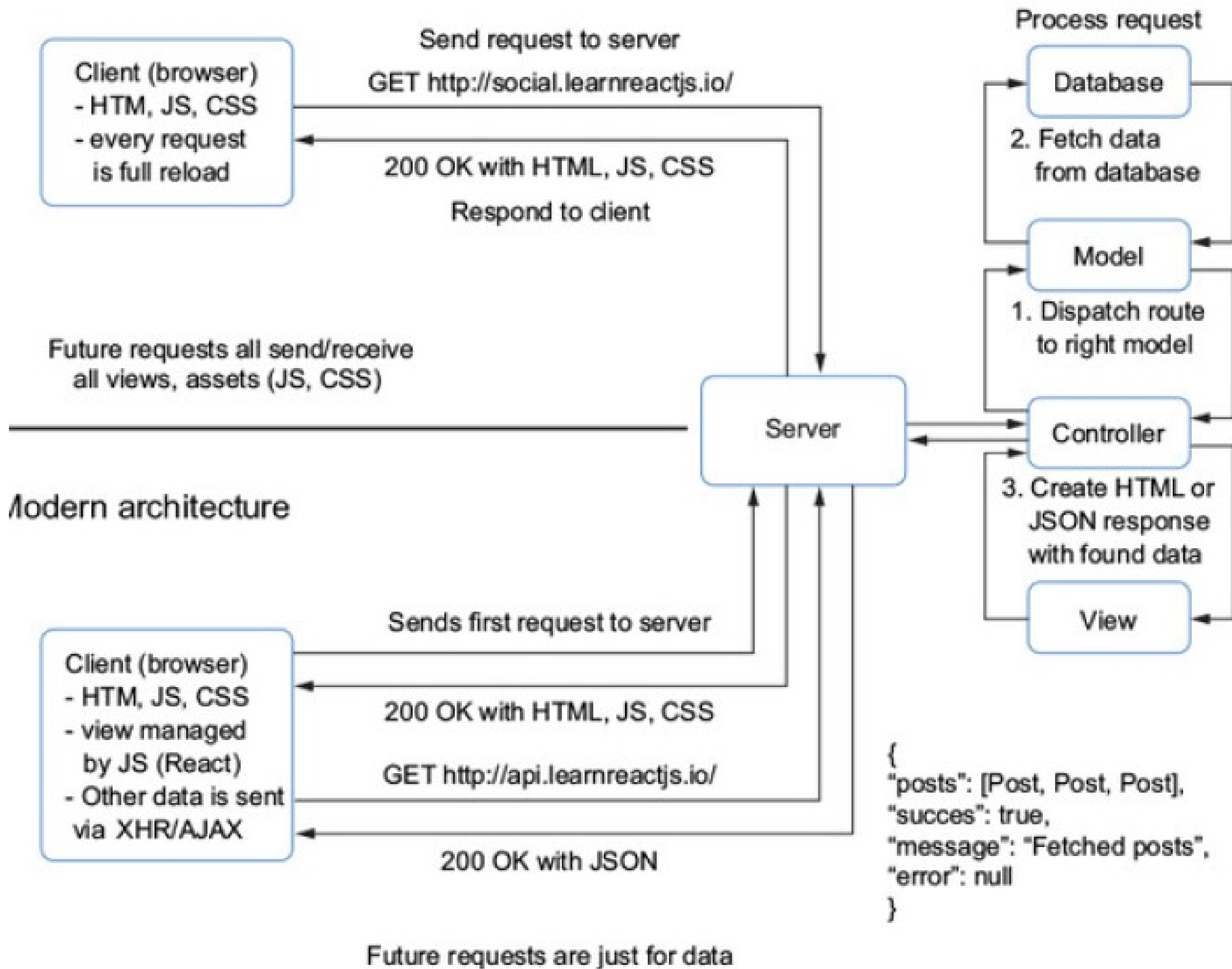
{this.props.children} → means that render my children here.

<IndexRoute> → If you want a child route to be used as the default when no other child matches, this special route is used.

Old & New Web Appl. Architecture

- In the old way, dynamic content would be generated on the server. The server would usually fetch data from a database and use it to populate an HTML view that would be sent down to the client.
- Now there is more application logic on the client that gets managed by JavaScript (in this case, React). The server initially sends down the HTML, JavaScript, and CSS assets, but after that, the client React app takes over. From there, unless a user manually refreshes the page, the server will only have to send down raw JSON data.

React Router



Router - Route

- The Router has Route components as its children. Each of these components uses **two props: a `path` string and a component.** The `<Router/>` will use each `<Route/>` to match a URL and render the right component.

Router – Route Components working