

Programming in React JS

MVC Design Pattern

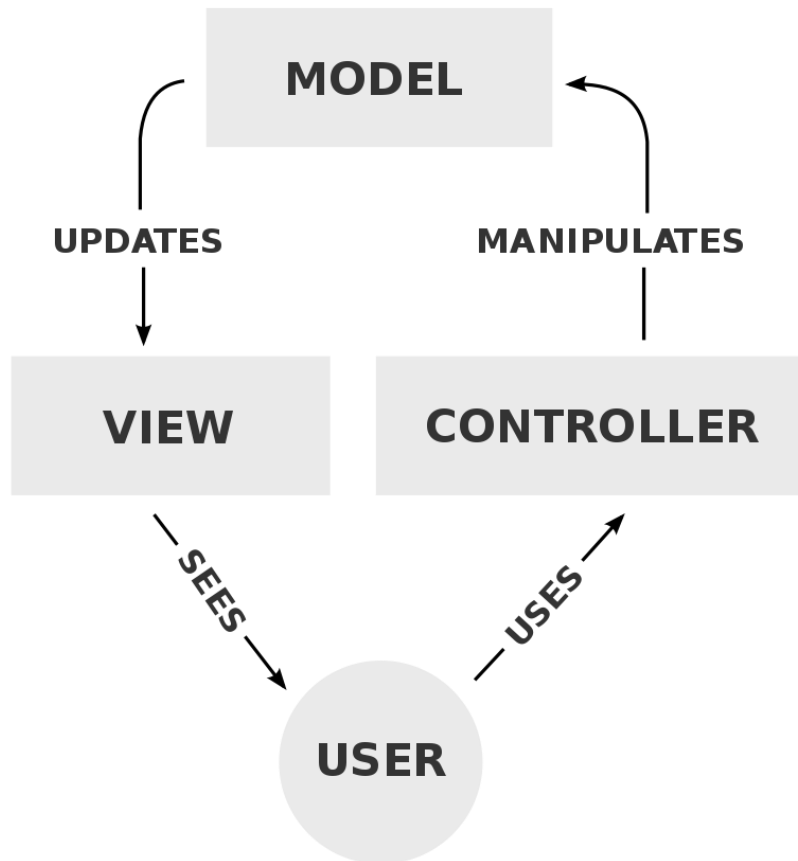


Image Courtesy:

<https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>

Model-View-Controller (MVC) design pattern is used for developing applications in a manner that divides the related program logic into three interconnected elements - a data model, presentation information, and control information.

Model – Responsible for managing the data of the application.
It receives user input from the controller.

View – Responsible for all the UI logic of the application required for representation of information.
Multiple views of the same information are possible.

Controller - Responsible for controlling the application logic
It acts as the coordinator between the View and Model
It accepts user input and converts it to commands for the model or view.

How MVC works

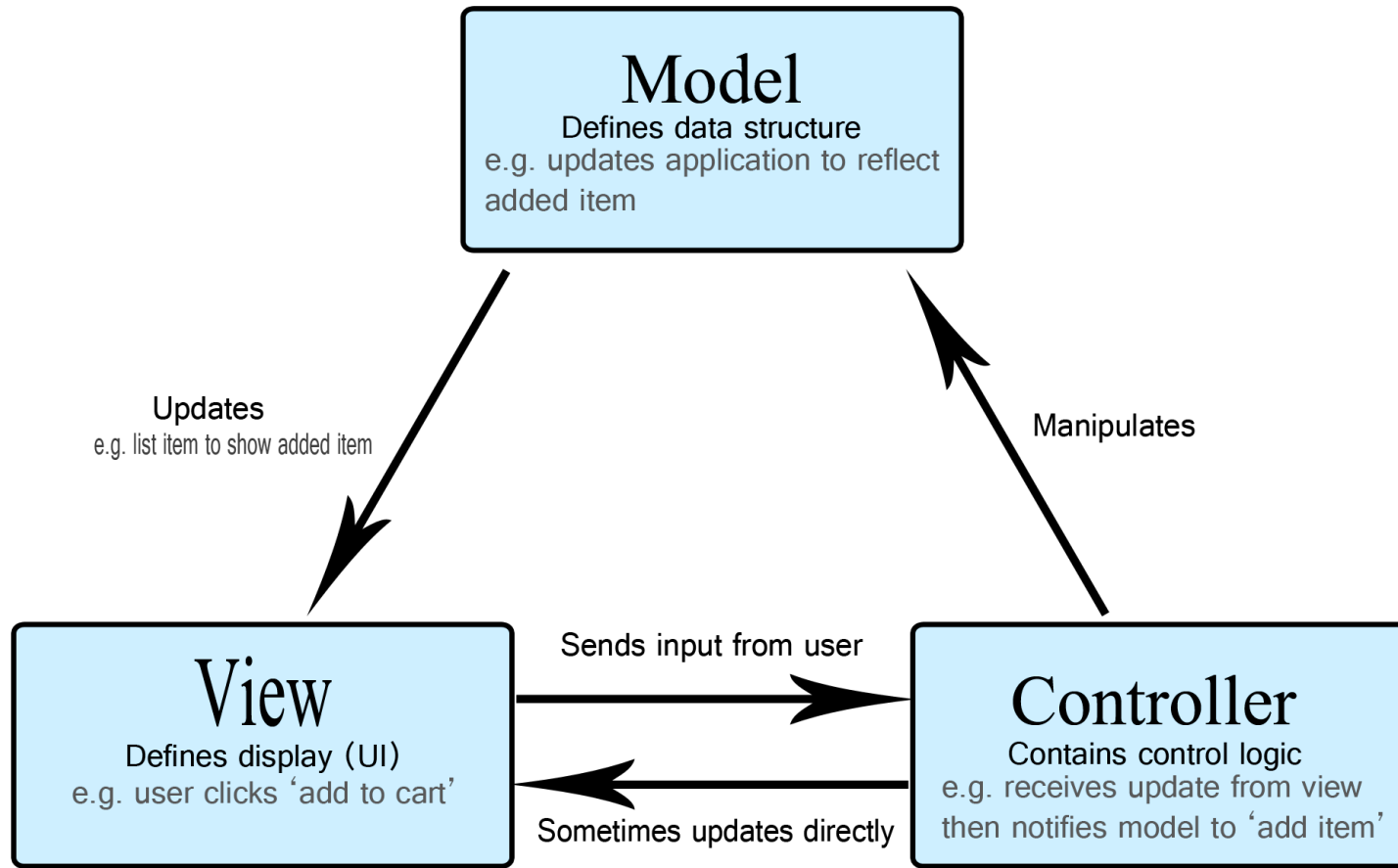


Image Courtesy: <https://blog.cloudboost.io/what-is-model-view-controller-124a9942246>

Ex. – Shopping List App

Model – specifies what data the list items should contain like item name, price, etc.

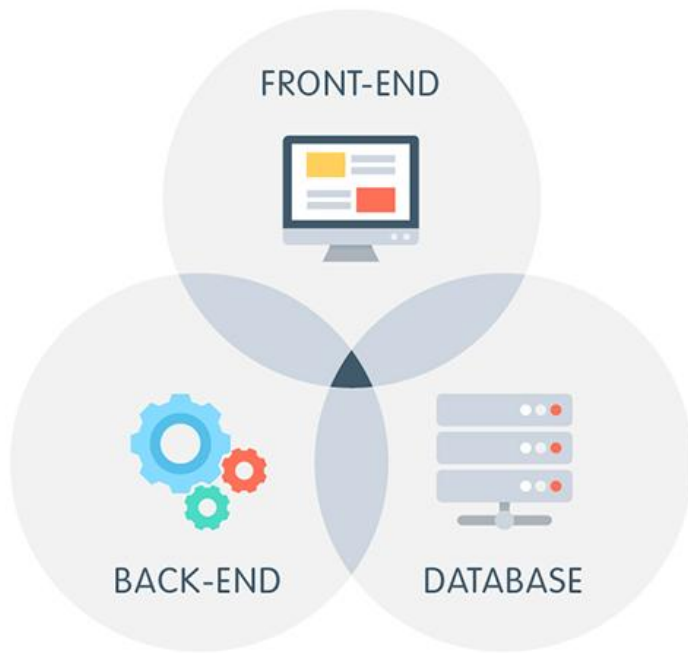
View – defines how the list is presented to the user, and receives data from the model to display.

Controller –

Add/ delete item - Requires the model to be updated, so the input is sent to the controller, which manipulates the model and the model then sends updated data to the view.

Sort item on name/ price – controller directly updates the view without updating the model

FULL-STACK DEVELOPMENT

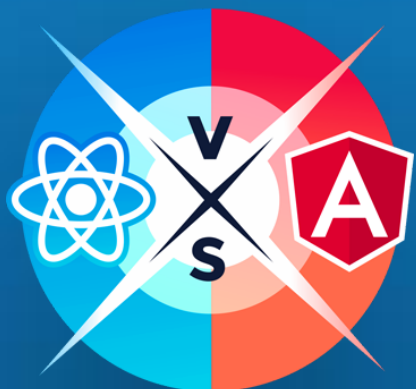


The combination of multiple technologies used to develop any web application is called a **STACK**

- **LAMP** – Linux, Apache, MySQL, PHP
- **MEAN** – MongoDB, Express, AngularJS, Node.js
- **MERN** – MongoDB, Express, React, Node.js

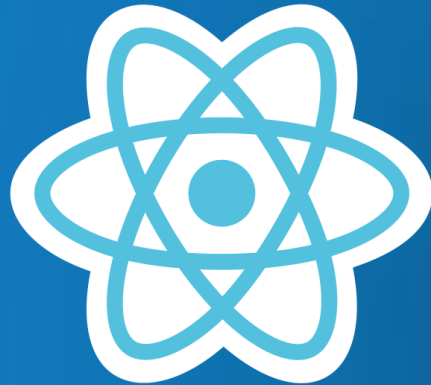
Image Courtesy:

<https://medium.com/@sepandassadi/become-a-full-stack-web-developer-free-resources-8a1c2c0ebd41>



React vs Angular

	REACT	ANGULAR
Initial Release	May 29, 2013	September 14, 2016
Latest Stable Release	17.0.0 / 20 October 2020	11.0.5 / 16 December 2020
Powered By	Facebook	Google
Type	Open Source JS Library	Fully-featured MVC Framework
Written In	JavaScript	TypeScript
Learning Curve	Moderate	Steep
DOM	Virtual DOM	Regular DOM
Language	JSX + JS (ES5 and beyond)	HTML + TypeScript
Data Binding	Unidirectional	Bidirectional
Component Based	Yes	Yes
Self Sufficiency	UI development only Extra libraries must be used	Holistic software development
Architecture Flexibility	Yes	No
Coding Speed	Normal	Slow
App Structure	Flexible Component-based View only	Fixed and complex Component-based Model-View-Controller
Directives	Easily understood with knowledge of JS	Incomprehensible without knowledge of Angular
App Testing	Requires a set of tools to perform different types of testing	Testing of complete app is possible with a single tool



React

- React was created by Jordan Walke, a software engineer at Facebook.
- It is a JavaScript library that aims to simplify development of visual interfaces.
- It is used for building fast and interactive user interfaces for web and mobile applications.
- It is an open-source, component-based, front-end library responsible only for the application's view layer.

Why React

- **Easy creation of dynamic applications** - A component-based approach, well-defined lifecycle, and use of just plain JavaScript make React very simple to learn, and build dynamic web and mobile applications.
- **Fast render with Virtual DOM** - Virtual DOM updates only the items in the Real DOM that were changed, instead of updating all of the components again, as conventional web applications do.
- **Reusable components** - Components are the building blocks of any React application. A single app usually consists of multiple components that can be reused throughout the application, which in turn reduces the application's development time.
- **Unidirectional data flow** - React follows a unidirectional data flow, wherein data always flows from parent to child component. This unidirectional data flow assists in debugging errors and identification of problematic points in an application.
- **Small learning curve** - React is easy to learn, as it mostly combines basic HTML and JavaScript concepts with some beneficial additions.
- **Dedicated tools for easy debugging** - Facebook has released a Chrome extension that can be used to debug React applications. This makes the process of debugging React web applications faster and easier.

Getting Started

Write React code directly in HTML

- Include three CDN's in your HTML file:

[React](#) - the React top level API

[React-DOM](#) - adds DOM-specific methods

[Babel](#) - a JavaScript compiler

```
<!DOCTYPE html>
<html>
  <head>
    <script
src="https://unpkg.com/react@17/umd/react.development.js"></script>
    <script src="https://unpkg.com/react-dom@17/umd/react-
dom.development.js"></script>
    <script
src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>
  <body>

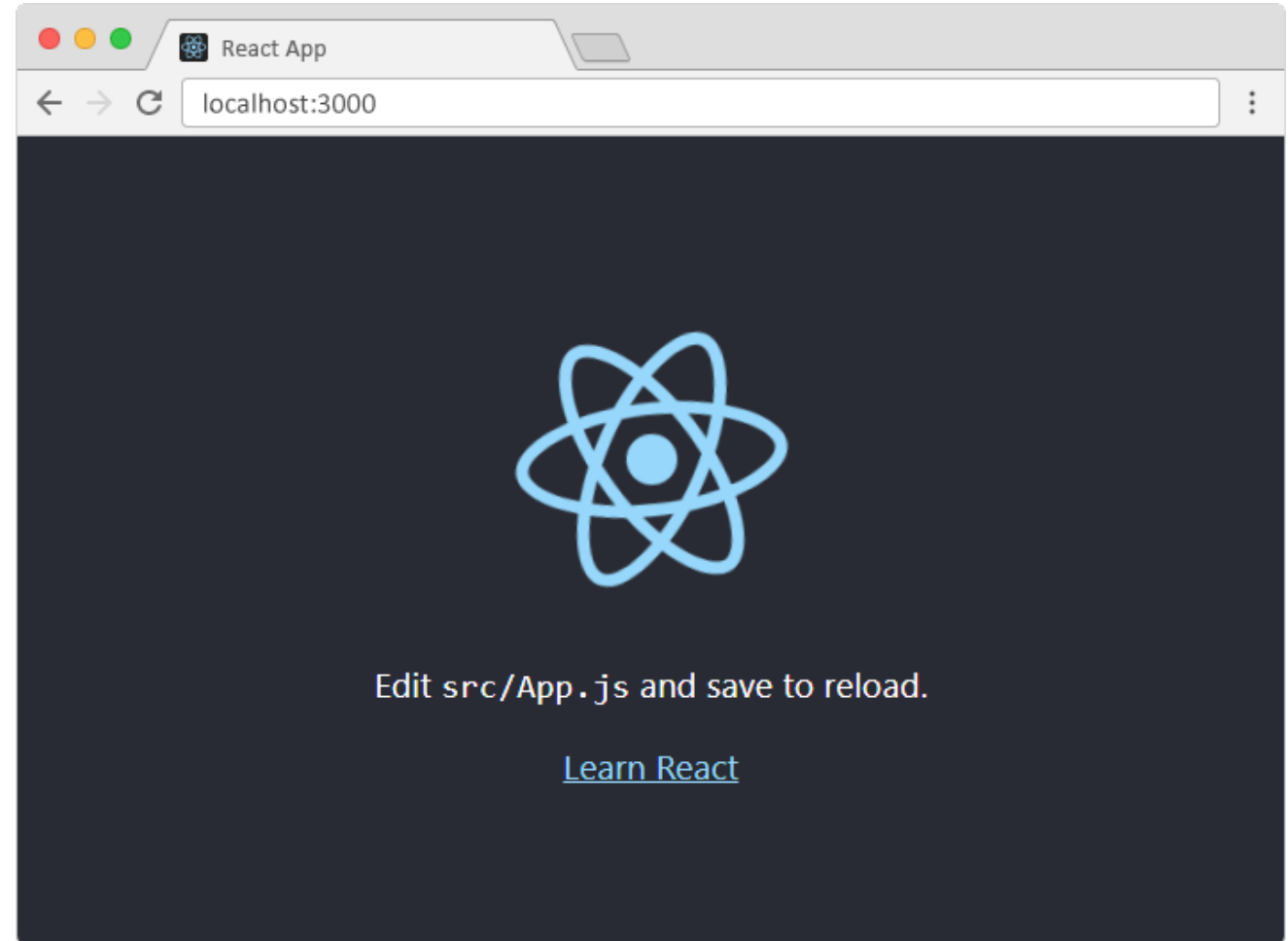
    <div id="mydiv"></div>

    <script type="text/babel">
      class App extends React.Component {
        render() {
          return <h1>Hello world!</h1>
        }
      }

      ReactDOM.render(<App />, document.getElementById('root'))
    </script>
  </body>
</html>
```


Setting up a React environment – using Create React App

- *Install Node.js*
- *Set up create-react-app → run command `npm install -g create-react-app` in your terminal*
- *Create a react app → run command `npx create-react-app your-app-name` in your terminal*
- *Move to the newly created directory*
- *Start the project → run command `npm start` in your terminal*
- *A new window will open in the web browser at `localhost:3000` with your new React app*



Getting Started

When we create a React project, our application's structure looks like this

```
├─ node_modules
├─ public
│  └─ ★ favicon.ico
│     <> index.html
│     {} manifest.json
├─ src
│  └─ # App.css
│     JS App.js
│     JS App.test.js
│     # index.css
│     JS index.js
│     logo.svg
│     JS registerServiceWorker.js
├─ .gitignore
├─ {} package-lock.json
├─ {} package.json
└─ ⓘ README.md
```

node_modules - holds all the dependencies and sub-dependencies of our project

Public folder - this folder contains 3 files

- favicon.ico - contains an icon which displays on the browser.
- index.html - due to this file, the public folder, known as “root folder”, gets served by the web server.
- manifest.json – contains metadata about our application.

src folder - this folder contains actual source code for developers. Pre-existing files are:

- App.css - gives some CSS classes/ styling which is used by App.js file.
- App.js - a sample React component called “App” which we get for free when creating a new app.
- App.test.js - a test file
- index.css - stores the base styling for the application.
- index.js - stores the main Render call from ReactDOM. It imports App.js component and tells React where to render it.
- logo.svg - contains the logo of Reactjs, visible on the browser.
- registerServiceWorker.js - used for registering a service

Package.json - contains the information like the name of project, versions, dependencies etc. Whenever we install a third party library, it automatically gets registered into this file

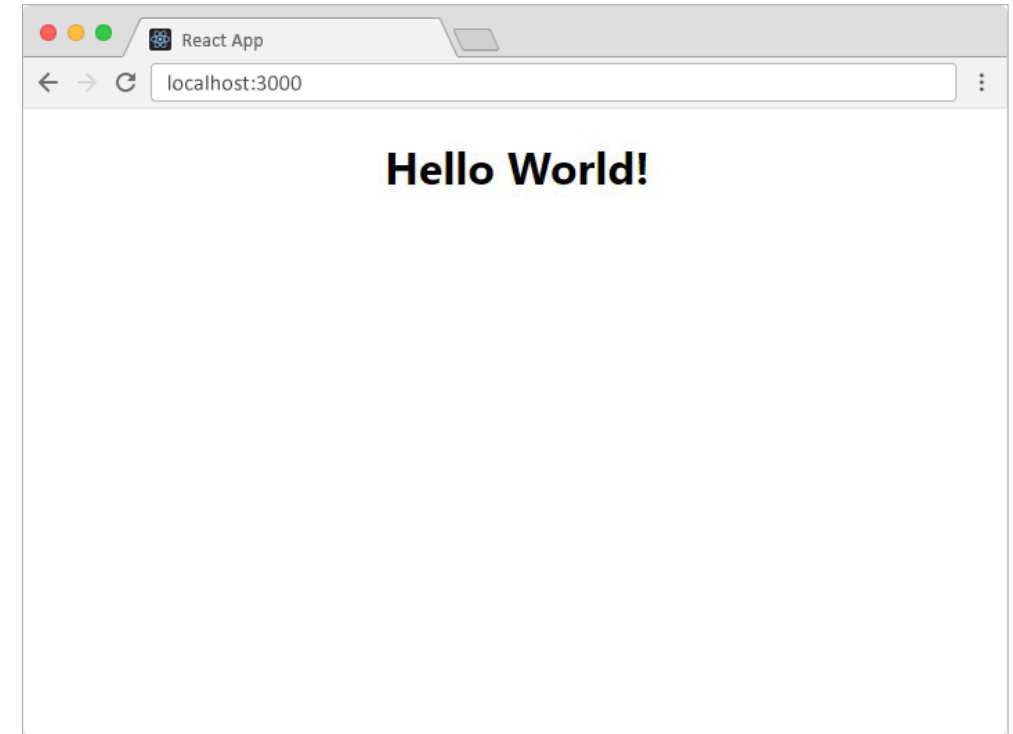
Getting Started

src/index.js

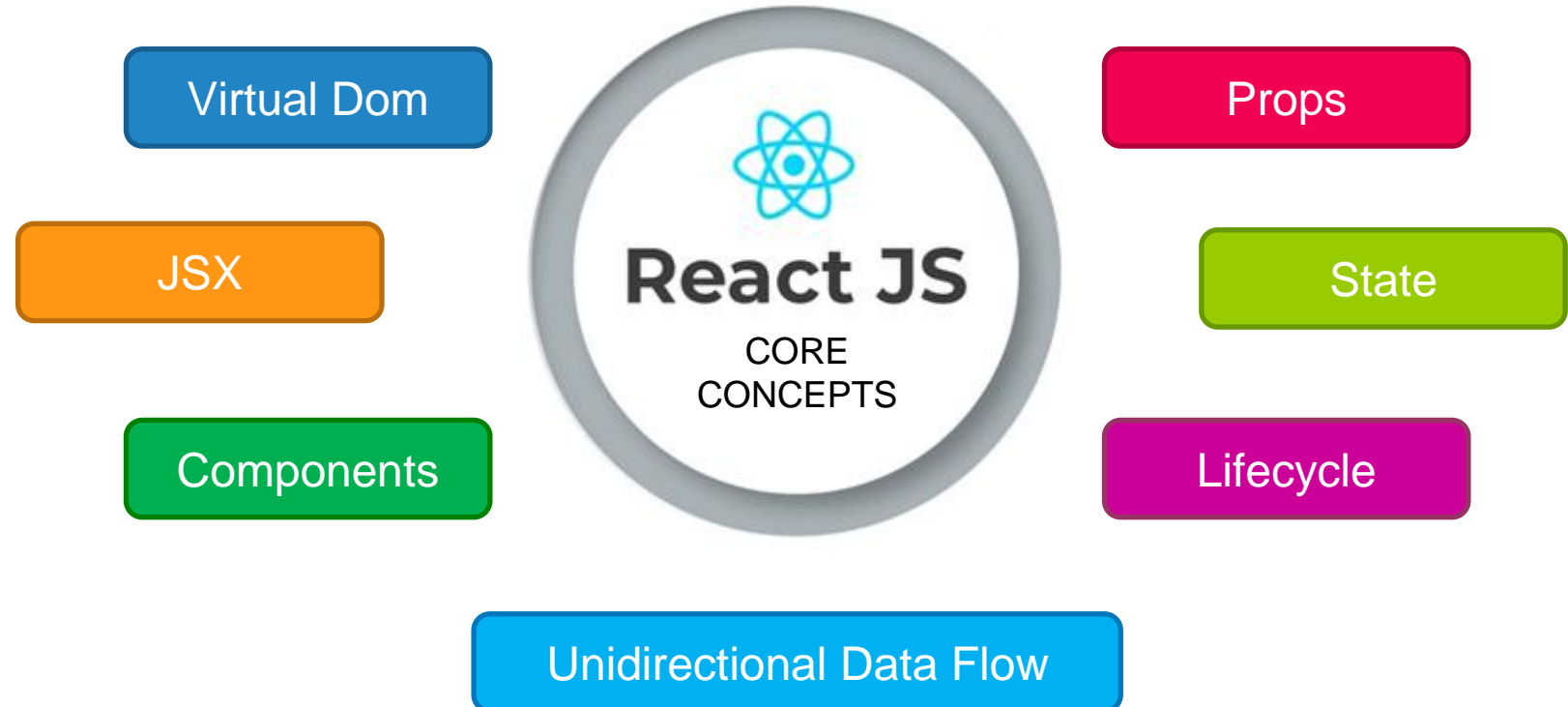
```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'

class App extends React.Component {
  render() {
    return (
      <div className="App">
        <h1>Hello World</h1>
      </div>
    )
  }
}

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



Core Concepts



Document Object Model (DOM)

- The **Document Object Model (DOM)** is a cross-platform and language-independent interface that treats an XML or HTML document as a tree structure wherein each node is an object representing a part of the document.¹
- When a web page gets loaded, the browser creates a DOM of the page, which is constructed as a **Tree of Objects** from all elements on that web page.
- Using DOM interface gives JS all the power it needs to create **Dynamic HTML**.

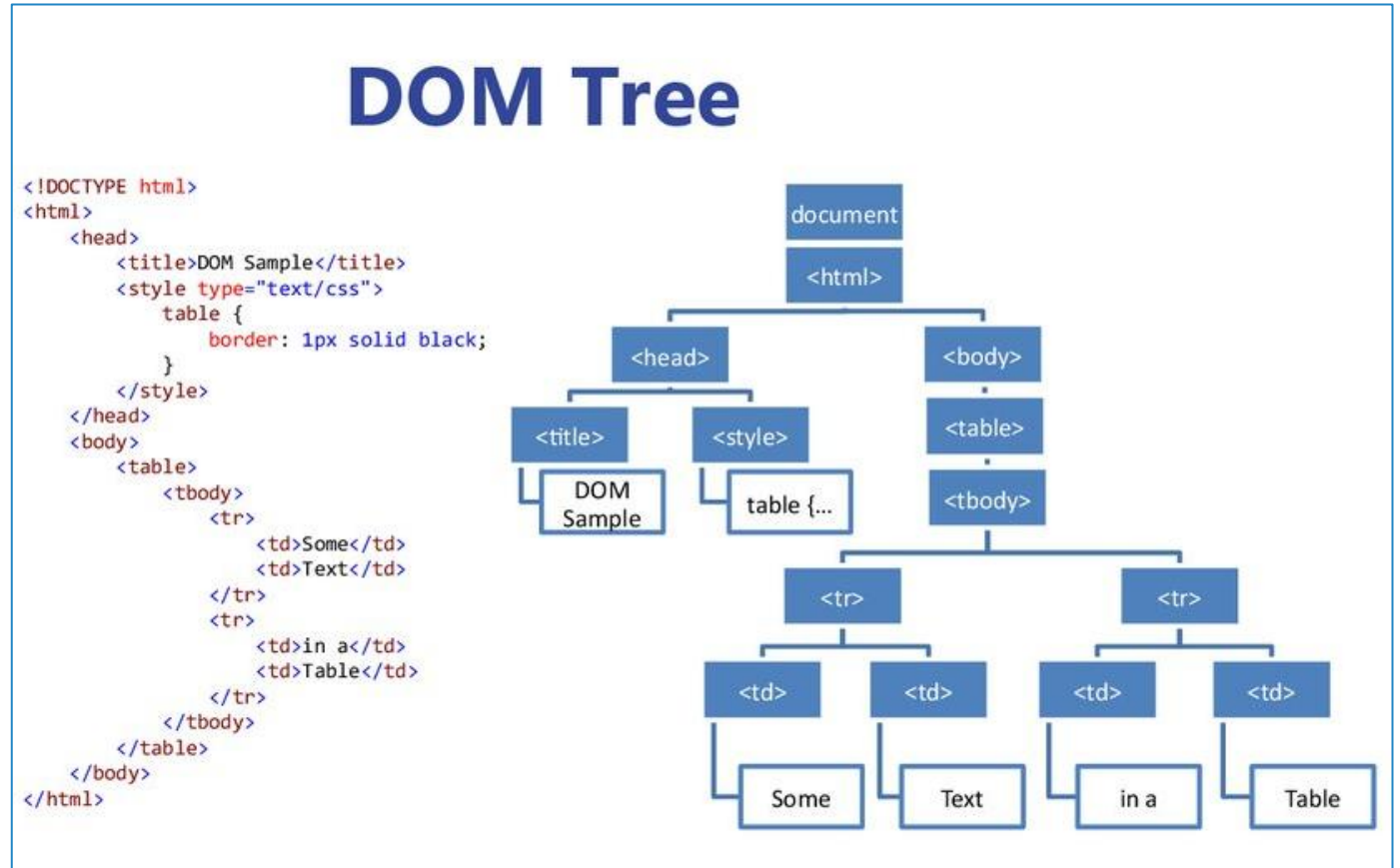


Image Courtesy: <https://en.ppt-online.org/83335>

Text Courtesy: [1] https://en.wikipedia.org/wiki/Document_Object_Model

What is a virtual DOM?

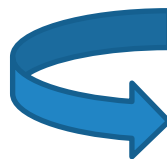
- A virtual DOM object is a *representation* of an actual DOM object, like a lightweight copy.
- Virtual DOM object has the same properties as a real DOM object, but unlike the real object it cannot directly change what's being displayed on the screen.
- Manipulating the regular DOM is slow. Manipulating the virtual DOM is much faster.

How it helps?

Whenever something changes on the screen

1. A snapshot of the virtual DOM is taken and stored.
2. The entire virtual DOM then gets updated.
3. The updated virtual DOM is compared with the pre-updated snapshot. It is figured out as to which objects have changed.
4. Thereafter, only the changed objects get updated in the real DOM.
5. Changes on the real DOM cause the web page to change.

- JSX allows us to write HTML in React
- Under the hood, it runs *createElement()*, which takes the tag, object containing the properties, and children of the component and renders the same information.



```
const heading = <h1 className="site-heading">Hello, React</h1>  
const heading = React.createElement('h1', {className: 'site-heading'}, 'Hello, React!')
```

A few key points to remember when using JSX

- *className* is used instead of *class* for adding CSS classes, as *class* is a reserved keyword in JS
- Properties and methods in JSX are camelCase (*onclick* will become *onClick*)
- Self-closing tags must end in a slash - e.g. ``

- To write HTML on multiple lines, put the HTML inside parentheses
- Multi-line HTML code must be wrapped in ONE top level parent element

```
const myDiv = (  
  <div>  
    <h1>REACT IN THE REAL WORLD</h1>  
    <p>React has become arguably the most popular JavaScript framework  
      currently in use </p>  
  </div>  
)
```


Expressions in JSX

- JavaScript expressions can be embedded inside JSX using curly braces
- The expression can be a variable, a function, a property, or any valid JS expression

```
const txt1 = <h1>React is {5 * 2} times better with JSX</h1>;
```

```
const name = 'Anu'  
const txt1 = <h1>Hello, {name}</h1>
```

```
const myId = 'test'  
const element = <h1 id={myId}>Hello, world!</h1>
```

Declarative code in JSX

Declarative programming **focuses on the WHAT rather than the HOW**.

It is driven by describing the end result, rather than specifying the step by step process of getting to the end result.

```
function addCourseNameToBody() {  
  const bodyTag = document.querySelector('body')  
  const divTag = document.createElement('div')  
  let h1Tag = document.createElement('h1')  
  h1Tag.innerText = "Learning React"  
  divTag.append(h1Tag)  
  bodyTag.append(divTag)  
}
```

```
class Course extends Component {  
  render() {  
    return(  
      <div>  
        <h1>{this.props.name}</h1>  
      </div>  
    );  
  }  
}
```

Applying style in JSX

```
class Styles extends React.Component {  
  h2Style = {  
    fontSize: 40,  
    textDecoration: 'underline'  
  };  
  
  render() {  
    return (  
      <div>  
        <h2 style={this.h2Style}>React JS</h2>  
  
        <p style={{ fontStyle: 'italic', color: 'blue' }}>  
          React is a JavaScript library that aims to  
          simplify development of visual interfaces.  
        </p>  
  
        <button className="btn btn-success">Get Started</button>  
      </div>  
    );  
  }  
}
```

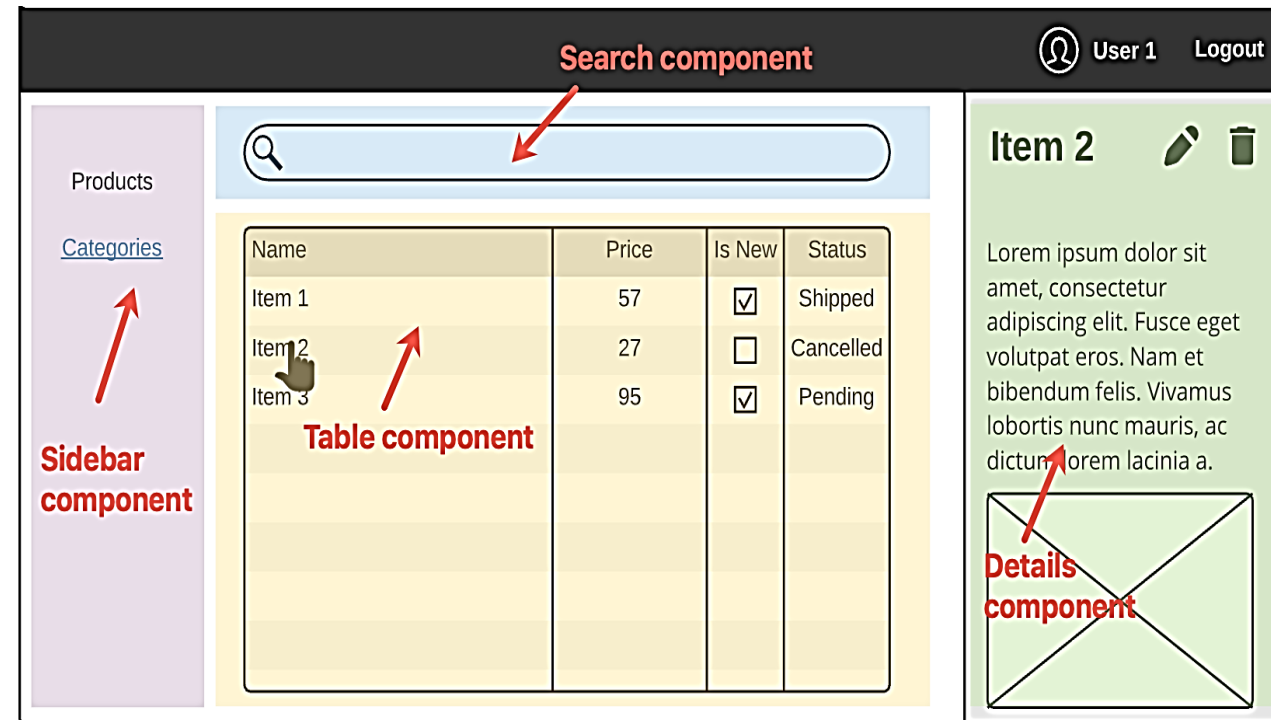
As a property

Inline Style

Using Classes

Components

- A component is one isolated piece of user interface
- Components are independent and reusable bits of codes that collectively form the UI
- Components are like JS functions that return HTML elements
- Component's name must start with an upper case letter
- Components are of two types,
 - Stateful components
 - Stateless components
- React apps have multiple components, and each component can be placed in a separate file



The mockup illustrates a web application layout with four distinct components highlighted by red arrows:

- Search component:** A search bar at the top center.
- Sidebar component:** A vertical navigation menu on the left containing 'Products' and 'Categories' (with a red arrow pointing to it).
- Table component:** A table displaying product data with columns: Name, Price, Is New, and Status.
- Details component:** A panel on the right showing 'Item 2' details, including a description and a placeholder image (indicated by a red 'X').

Name	Price	Is New	Status
Item 1	57	<input checked="" type="checkbox"/>	Shipped
Item 2	27	<input type="checkbox"/>	Cancelled
Item 3	95	<input checked="" type="checkbox"/>	Pending

Stateful Components

- A stateful component is always a class component
- It is created by extending the `React.Component` class
- It has a separate `render()` method for returning JSX on the screen
- It may hold and manage its own state
 - component properties are kept in an object called state
- Its behavior can be made reactive to its state

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

Stateless Components

- Stateless component can be created using function or class
- They hold no state of their own
- They may receive props from other components

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}
```

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

Nested Components

```
const TableHeader = () => {  
  return (  
    <thead>  
      <tr>  
        <th>Brand</th>  
        <th>Headquarters</th>  
      </tr>  
    </thead>  
  )  
}
```

```
const TableBody = () => {  
  return (  
    <tbody>  
      <tr>  
        <td>Tata</td>  
        <td>India</td>  
      </tr>  
      <tr>  
        <td>Toyota</td>  
        <td>Japan</td>  
      </tr>  
    </tbody>  
  )  
}
```

```
class Table extends Component {  
  render() {  
    return (  
      <table>  
        <TableHeader />  
        <TableBody />  
      </table>  
    )  
  }  
}
```

Components in Files

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

class Hello extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    return <h2>Hi, {this.props.name}</h2>;
  }
}

export default Hello;
```

If your component has a constructor function, the props should always be passed via the `super()` method.

```
// index.js
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import Hello from './app.js';
```

```
ReactDOM.render(<Hello name="Jane"/>, document.getElementById('root'))
```


Props

- Props are arguments passed to React components, by using HTML attributes
- They are like function arguments in JavaScript
- They store the value of a tag's attributes and work similar to the HTML attributes
- Props are read-only
- All React components must act like pure functions with respect to their props

```
// Props
class Hello extends React.Component {
  render() {
    return (
      <div>
        <h2>Hi! I am {this.props.name}</h2>
        <p>I am a {this.props.age} years old {this.props.occupation}</p>
      </div>
    );
  }
}
```

```
let name_1 = "Jamie";
```

```
ReactDOM.render(<Hello name={name_1} age={27} occupation="Banker"/>, document.getElementById('root'))
```

Props

```
//Props Component
class Age extends React.Component {
  render() {
    let age = 2021-this.props.details.year;
    return (
      <h2>Hey {this.props.details.name}!, you are {age} years old.</h2>
    );
  }
}
```

```
class Person extends React.Component {
  render() {
    const personinfo = {name: "Chris", year: 1976};
    return (
      <div>
        <Age details={personinfo} />
      </div>
    );
  }
}
```

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

State

- The state is a built-in React object that is used to store data/ information about the component
- The component re-renders when the state object changes
- The state object is initialized in the constructor with as many properties as desired
- *this.state.propertyname* - used to access the state object properties anywhere in the component

```
//State
class Person extends React.Component {
  constructor(props) {
    super(props);
    this.state = {name: this.props.name, occupation: "Chef"};
  }

  render() {
    return (
      <div>
        <h2>Hi! I am {this.state.name}
        <br/>I am a {this.state.occupation}</h2>
      </div>
    );
  }
}
```

```
ReactDOM.render(<Person name="Jamie"/>, document.getElementById('root'));
```

```
//setState()
class Person extends React.Component {
  constructor(props) {
    super(props);
    this.state = {name: this.props.name, occupation: "Chef"};
  }

  // setState()
  changeDetails = () => {
    this.setState({occupation: "Blogger", name: "Jane"});

    render() {
      return (
        <div>
          <h2>Hi! I am {this.state.name}
          <br/>I am a {this.state.occupation}</h2>
          <button type="button" onClick={this.changeDetails}>Change Details</button>
        </div>
      );
    }
  }
}
```

this.setState()
method is used to
change the value of
a property in the
state object This
method is used

```
ReactDOM.render(<Person name="Jamie"/>, document.getElementById('root'));
```

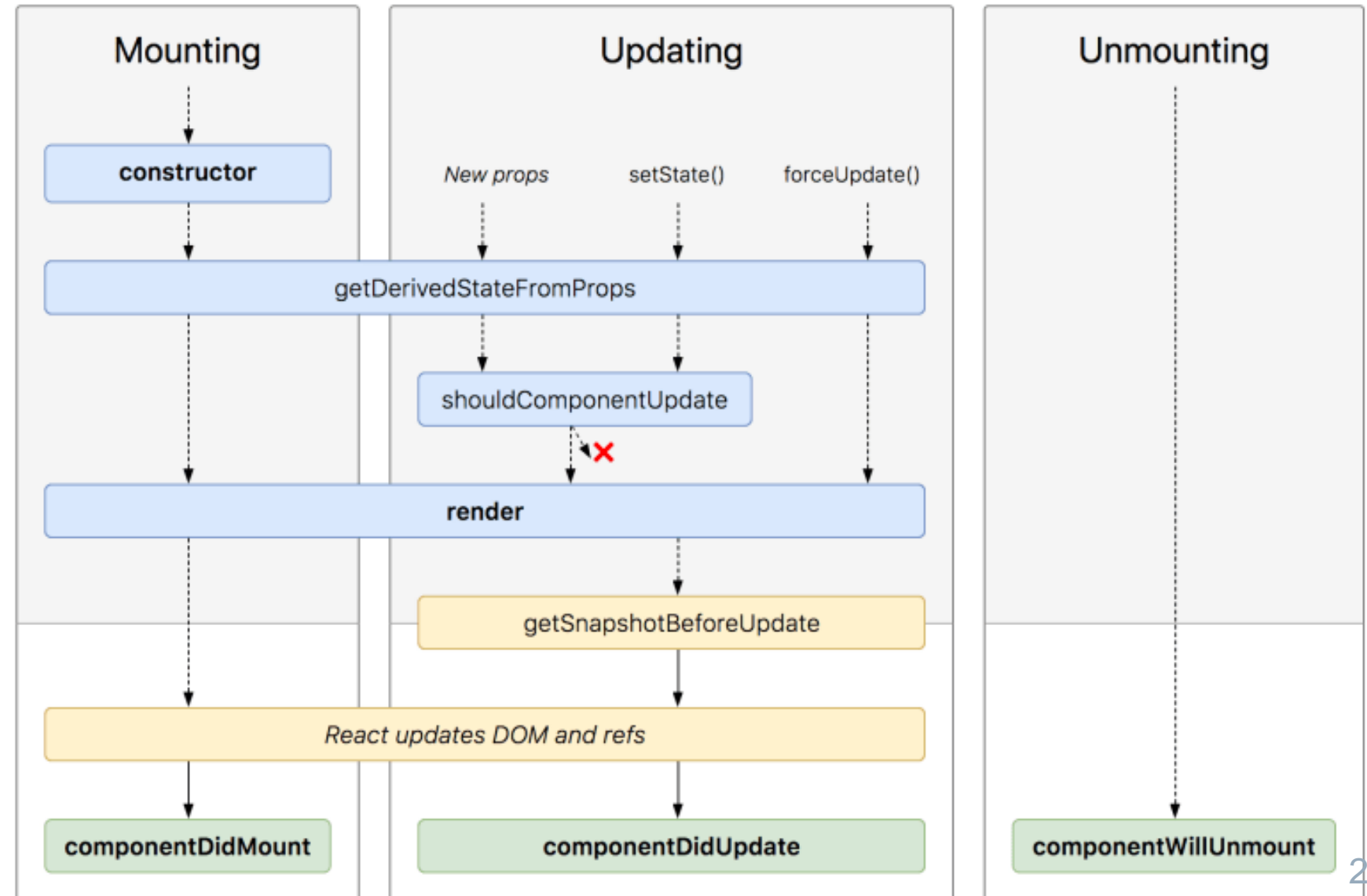
React Lifecycle Methods

React version 16.4

Language en-US

Every React component's lifecycle consists of three main phases –

- **MOUNTING**
(mounted on the DOM)
- **UPDATING**
(state/ props get updated)
- **UNMOUNTING**
(unmounted from the DOM)



React Lifecycle Methods

- **constructor()** – invoked before the component is mounted
 - sets up the initial state and values
 - invoked with the props as arguments
 - first statement should be a call to *super(props)*
 - serves two main purposes → Initializing local state by assigning an object to *this.state*
 - Binding event handler methods to an instance
- **render()** - most used lifecycle method
 - is the only required method within a class component
 - handles the rendering of your component to the UI
 - invoked during mounting as well as updating phases
 - should be a pure function
 - does not directly interact with the browser
 - will not be invoked if *shouldComponentUpdate()* returns false

- **componentDidMount()** – invoked immediately after a component is mounted
a good place to initiate API calls, if you need to load data from a remote endpoint
allows the use of `setState()`
- **componentDidUpdate()** – invoked as soon as an update occurs (*not invoked for the initial render*)
it updates the DOM (*say in response to state/ props change*)
can call `setState()` in this lifecycle - but wrap it in a condition that checks for state/
prop changes from previous state
will not be invoked if `shouldComponentUpdate()` returns false
- **componentWillUnmount()** – invoked just before a component is unmounted and destroyed
cleanup activities should be done in this method
should not call `setState()` in this, as the component will never be re-rendered

- **getDerivedStateFromProps()** – first method to be invoked when a component gets updated (*if defined*)
is invoked prior calling the render method (*both on initial mount & subsequent updates*)
is a static function that does not have access to “this”
returns an object to update state in response to prop changes
returns null if there is no change to state
- **shouldComponentUpdate()** – it is useful when you don’t want to render your state or prop changes
lets React know if a component’s output is not affected by a change in state/ props
the default behavior is to re-render on every state change
is invoked before rendering when new props or state are being received
is not called for the initial rendering or when *forceUpdate()* is used
- **getSnapshotBeforeUpdate()** – invoked before the most recently rendered output is committed (*to the DOM*)
aids the component to capture some information from the DOM before it is changes
value that it returns is passed on to *componentDidUpdate()*

- **Error boundaries** - React components that handle JS errors anywhere in their child component tree, log those errors, and display a fallback UI
- The following methods are invoked when an error is thrown by a descendant component
 - static `getDerivedStateFromError()`** - It receives the error that was thrown as a parameter and returns a value to update the state of the component.
Rendering of fallback UI can be handled here.
 - `componentDidCatch()`** - It takes in two parameters →
 - the error that was thrown, and
 - an object containing info about which component threw the errorIt can be used for activities like logging errors, etc.

Unidirectional Data Flow (UDF)

UDF means that data has one, and only one, way to be transferred to other parts of the application, generally from top (parent) to bottom (child) components.

This means that:

- state is passed to the view and child components
- actions are triggered by the view
- actions update the state
- the state change is passed to the view and child components

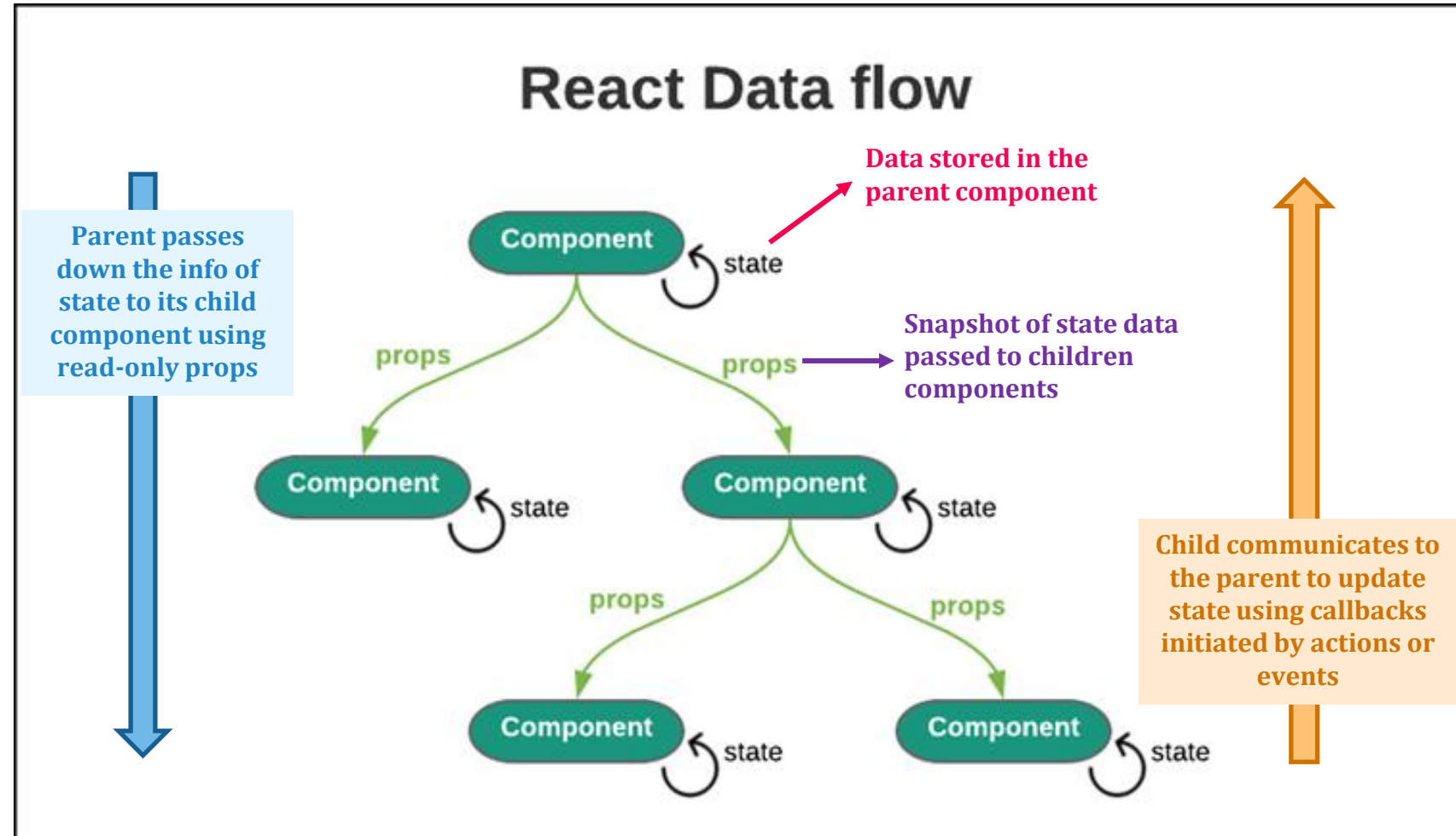


Image Courtesy: <https://medium.com/javascript-in-plain-english/reactjs-training-understanding-react-and-typescript-d01deb2dd127>

Some advantages of UDF

- Easier to debug as there is a single source of truth, i.e., the state is maintained at a single location, it can be easily traced what actions are updating the data as the flow is unidirectional.
- You can control which components to re-render on a common state change. You put all those components under one common parent component.
- Easy to trace and fix bugs and errors caused by any bad code
- Leads to a predictable and clean data flow architecture allowing more control over the data.
- Less error prone, as you have more control over the data.

React Events

React has the same events as HTML: click, change, mouseover, keydown etc.

Event Handling

1. Add event to the element

Events are named using camelCase, rather than lowercase
[onClick instead of onclick]

2. Include event handler as a method in the component class

Event handlers are written inside curly braces

onClick = {calcSum} [as written in React]

onClick = "calcSum()" [as written in HTML]

3. Bind *this* - using arrow functions bind *this* to the component instance

React Events

```
class EventBind extends Component {
  constructor(props) {
    super(props)
    this.state = {
      message: 'Welcome'
    }
  }

  clickHandler() {
    this.setState({
      message: 'Farewell'
    })
  }

  render() {
    return (
      <div>
        <h3>{this.state.message}</h3>
        <button onClick={this.clickHandler}>Click</button>
      </div>
    )
  }
}
```

Output

TypeError: Cannot read property 'setState' of undefined

clickHandler

F:/ReactJS/helloworld/src/components/EventBind.js:15

```
12 | }
13 |
14 | clickHandler() {
> 15 |   this.setState({
    |   ^       message: 'Goodbye'
16 |   })
17 |
18 |   console.log(this)
```

Binding Event Handler

Binding Event Handler in Render Method

```
render() {  
  return (  
    <div>  
      <h3>{this.state.message}</h3>  
      <button onClick={this.clickHandler.bind(this)}>Click</button>  
    </div>  
  )  
}
```

Binding Event Handler using Arrow Function

```
render() {  
  return (  
    <div>  
      <h3>{this.state.message}</h3>  
      <button onClick={() => this.clickHandler()}>Click</button>  
    </div>  
  )  
}
```

Binding Event Handler

Binding Event Handler in the Constructor

```
constructor(props) {  
  super(props)  
  this.state = {  
    message: 'Welcome'  
  }  
  this.clickHandler = this.clickHandler.bind(this)  
}
```

Binding Event Handler using Arrow Function as a Class Property

```
clickHandler = () => {  
  this.setState({message: 'Farewell'})  
}  
  
render() {  
  return (  
    <div>  
      <h3>{this.state.message}</h3>  
      <button onClick={this.clickHandler}>Click</button>  
    </div>  
  )  
}
```

40

Composition vs Inheritance

Two ways of enhancing React components:

- **Inheritance**

property by which the objects of a derived class possess copies of the member data and functions of the base class

allows derived (child) component to *inherit and extend* base (parent) component's properties and duplicate

Eg. - A **current account is a bank account** can be modeled with inheritance

- **Composition**

Instead of inheriting properties from a base class, it describes a class that can reference one or more objects of another class as instances

allows combining one or more components into a newer, enhanced component
composition only aims at *enhancing behavior*

Eg. - A **current account has transaction history** can be modeled with composition

Both Inheritance and Composition, aim towards code reuse and cleaner code structure. But what does React recommend?

React recommends use of Composition over Inheritance, because

- everything in React is a component, and it follows a strong component-based model
- extending components with inheritance might lead to a scenario at some point, where the behaviors cannot be clubbed seamlessly.
- composition avoids too much nesting of components, thus leading to cleaner, more maintainable code.
- composition makes refactoring easier because instead of needing to modify and test every subclass/superclass, you can just swap in and out any particular object that you want to add/ update.
- if a class (component) wants to use behavior of multiple objects providing multiple types of additional functionality to your class, you don't have to decide which component should be made the superclass (base component).
- composition leads to loose coupling between child and parent component, making it easier to maintain code

Rendering classes dynamically

```
class Styles extends React.Component {  
  state = {  
    marks: 80  
  };  
  
  render() {  
    let classes = "badge badge-";  
    classes += (this.state.marks) > 30 ? "primary" : "warning";  
  
    return (  
      <h3>Result: &nbsp;  <span className={classes}>  
        {this.state.marks > 30 ? 'Pass': 'Fail'}  
      </span>  
    </h3>  
    );  
  }  
}
```

Runtime class
assignment

Rendering classes dynamically

```
class Styles extends React.Component {  
  state = {  
    marks: 80  
  };  
  
  render() {  
    return (  
      <h3>Result: &nbsp;<span className={this.getBadgeClasses()}>  
        {this.state.marks > 30 ? 'Pass': 'Fail'}  
      </span>  
    </h3>  
    );  
  }  
  
  getBadgeClasses() {  
    let classes = "badge badge-";  
    classes += (this.state.marks) > 30 ? "primary" : "warning";  
    return classes;  
  }  
}
```

Runtime class
assignment
using function

Conditional Rendering

```
class CRender extends React.Component {  
  state = {  
    fruits: ["apple", "banana", "orange"]  
  };  
};
```

```
renderFruits() {  
  if (this.state.fruits.length == 0) return <p>No Fruits!</p>  
  return <ul>{this.state.fruits.map(fruit => <li key={fruit}>{fruit}</li>)}</ul>;  
}
```

Rendering List

```
render() {  
  return (  
    <div>  
      {this.state.fruits.length === 0 && "Please add fruits to list"}  
      {this.renderFruits()}  
    </div>  
  );  
}
```

Conditional
Rendering

React Router

- **React Router is the de facto standard routing library for React.**
- It conditionally renders components depending on the route being used in the URL (eg. - / for the home page, /about for the about page, etc.).

- **Installation**

React Router has three modules:

react-router – core package that provides the core routing components and functions for React Router applications

react-router-dom – provides the routing components for web development environment [to install → [npm install react-router-dom](#)]

react-router-native - provides the routing components for mobile app development

Types of Routers

React Router provides three types of routers:

- **BrowserRouter** - uses HTML 5 history API
should be used when there is a server handling the dynamic requests
- **HashRouter** - uses client-side hash routing
used for static web apps, where the server can only respond to requests for files that it knows about
useful for situations where you don't have a server logic to handle the client-side
- **MemoryRouter** - keeps the URL changes in memory not in the user browsers
doesn't change the URL in your browser
useful for testing and non-browser environments like React Native

The URLs they create for a page “about”

<code>http://localhost:3000/about</code>	<code>// <BrowserRouter></code>
<code>http://localhost:3000/#/about</code>	<code>// <HashRouter></code>
<code>http://localhost:3000</code>	<code>// <MemoryRouter></code>

Route

- Used to render UI component whose path matches the current URL.
- `path` prop is used for specifying the path to be matched.
- `/` passed as the path to a route component, is called an *Index Route*.
- Route component matches just the beginning of the URL, and not the whole thing.
- `exact` prop is used to render component only if the paths are exactly the same.
- `component` prop is used to specify which component is to be rendered when the path matches the current URL.

Link

- Used for internal linking/ navigation between the pages of the application.
- Link components can be made to point to different routes, with the help of `to` attribute.
- A link component renders an anchor tag in your HTML document.
- `NavLink` is used to add the style attributes to the active routes
- `activeClassName` prop when used with `NavLink`, applies the corresponding style for an active route.

Switch

- Used to render the components only when path matches. In case of no match it fallbacks to the not found component.
- On rendering, Switch component searches through its children route elements to identify which route's path matches the current URL.
- With `<Switch>`, only the first child `<Route>` that matches the location gets rendered. If `<Switch>` is not used and multiple `<Route>`s are present, then all the routes that match are rendered inclusively.

Link, Route, and Switch components

```
import React from 'react'
import { Route, Link, Switch } from 'react-router-dom'
import Home from './home.jsx'
import About from './about.jsx'
import Contact from './contact.jsx'
import Notfound from './notfound.jsx'
```

```
class App extends React.Component {
  render() {
    return (
      <div>
        <h2>React Router Demo</h2>
        <nav className="navbar navbar-light">
          <ul className="nav navbar-nav">
            <li> <Link to="/">Home</Link> </li>
            <li> <Link to="/about">About</Link> </li>
            <li> <Link to="/contact">Contact</Link> </li>
          </ul>
        </nav>
        <hr />
        <Switch>
          <Route exact path="/" component={Home} />
          <Route path="/about" component={About} />
          <Route path="/contact" component={Contact} />
          <Route component={Notfound} />
        </Switch>
      </div>
    );
  }
}
```

Useful Reference Links

- <https://reactjs.org/>
- https://www.w3schools.com/react/react_getstarted.asp
- <https://www.guru99.com/reactjs-tutorial.html#12>
- <https://www.tutorialspoint.com/reactjs/index.htm>
- <https://www.javatpoint.com/reactjs-tutorial>
- <https://www.youtube.com/watch?v=Ke90Tje7VS0>
- [https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side JavaScript frameworks/React getting started](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started)