

**Department of Computer Science and Engineering**

**6<sup>th</sup> Semester , Program Elective**

# **Advanced Web Programming-II**

**(18CSE648), 3 Credits**

**Anil Kumar**

Assistant Professor

Dept. of CSE, NMIT

**AY 2020-2021**

# Unit-1

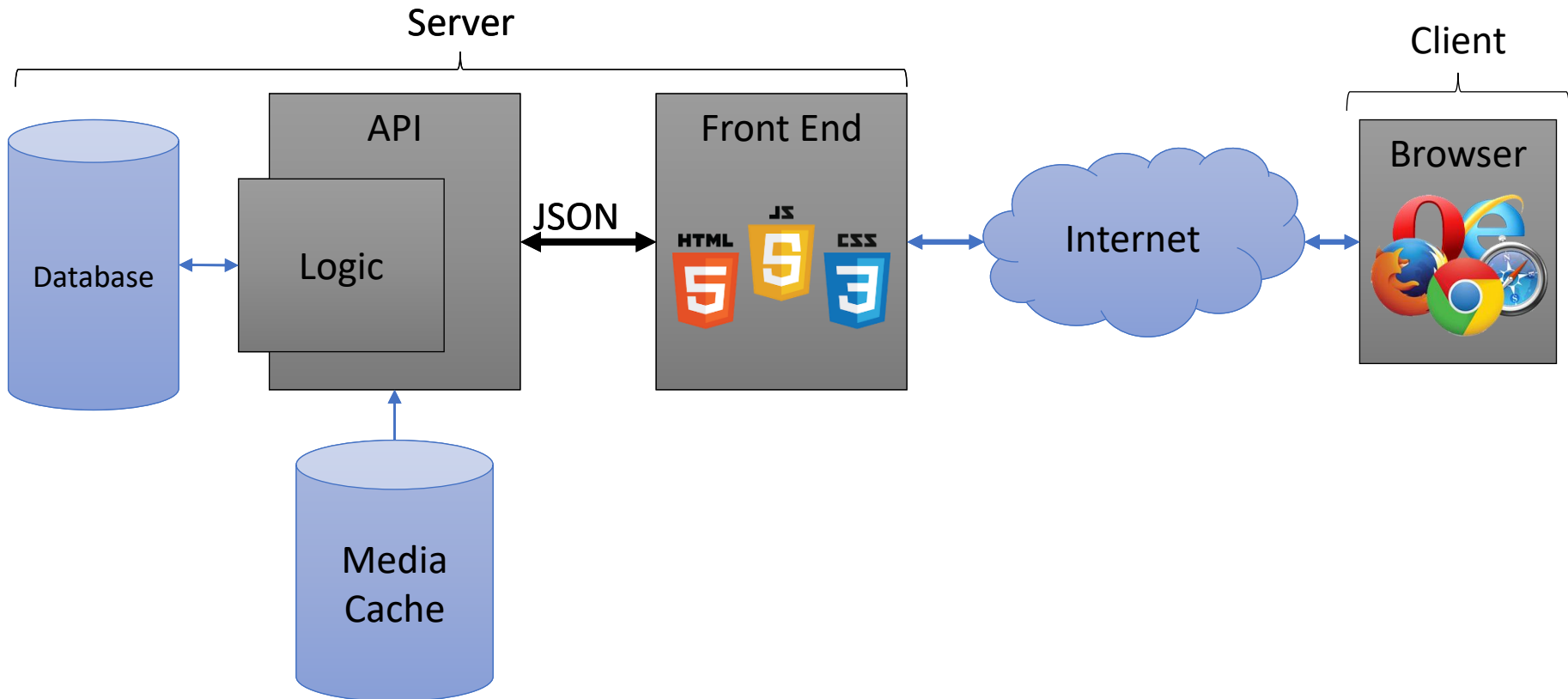
## Introduction to React

# Principles of Web Design/ Applications

- Availability
- Performance
- Reliability
- Scalability
- Manageability
- Cost

# Core Components of Web Applications

- UI (Front End (DOM, Framework))
- Request Layer (Web API)
- Back End (Database, Logic)



# Tools and Frameworks used for Web Development



# Back-End Frameworks





# Front-End Frameworks



# Front End Development



# Front End Languages

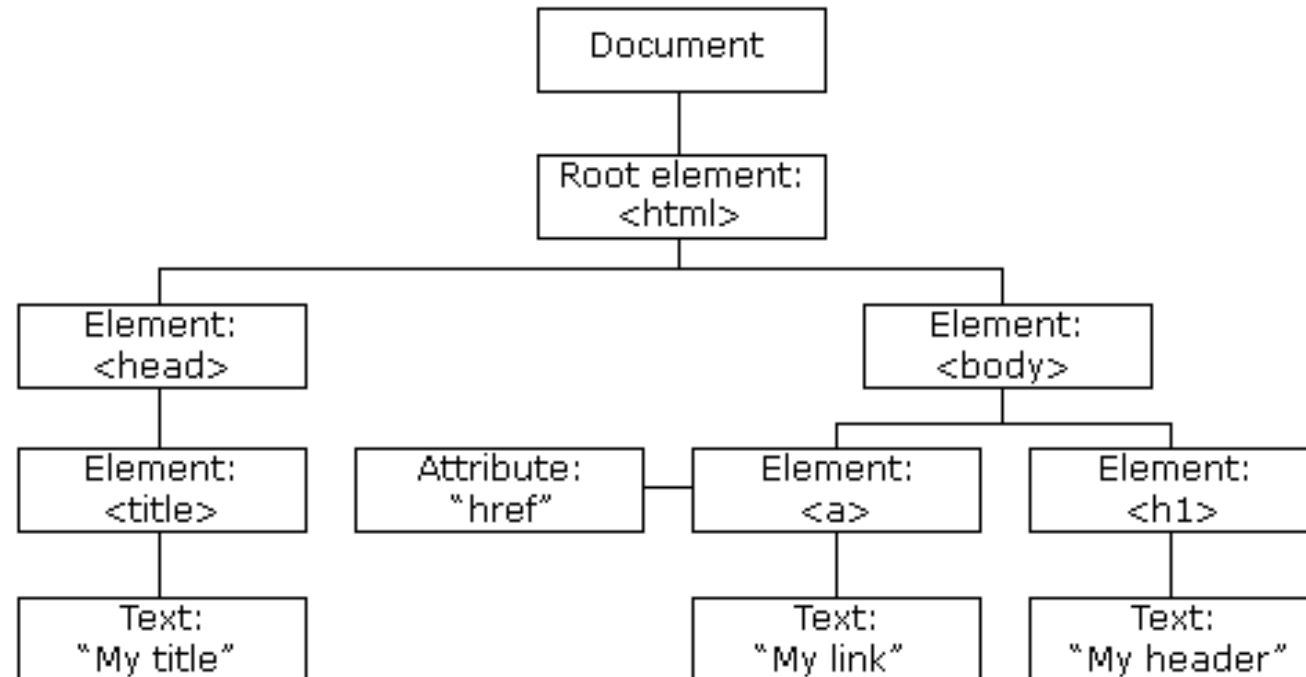
- HTML/CSS
- Javascript
- Java (applets)

Javascript/HTML/CSS is the only real option for front-end native languages and is basically the standard. But there are many variations on JavaScript that are used.

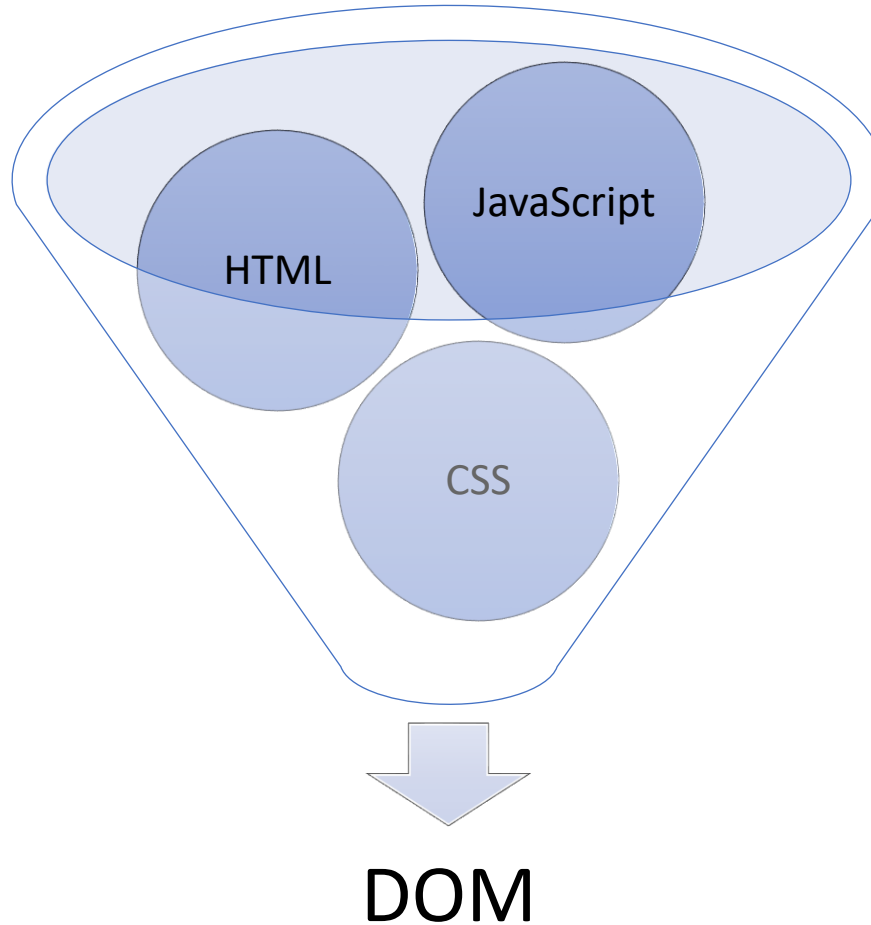


# DOM (Document Object Model)

- Document Object Model makes every addressable item in a web application an Object that can be manipulated for color, transparency, position, sound and behaviors.
- Every HTML Tag is a DOM object

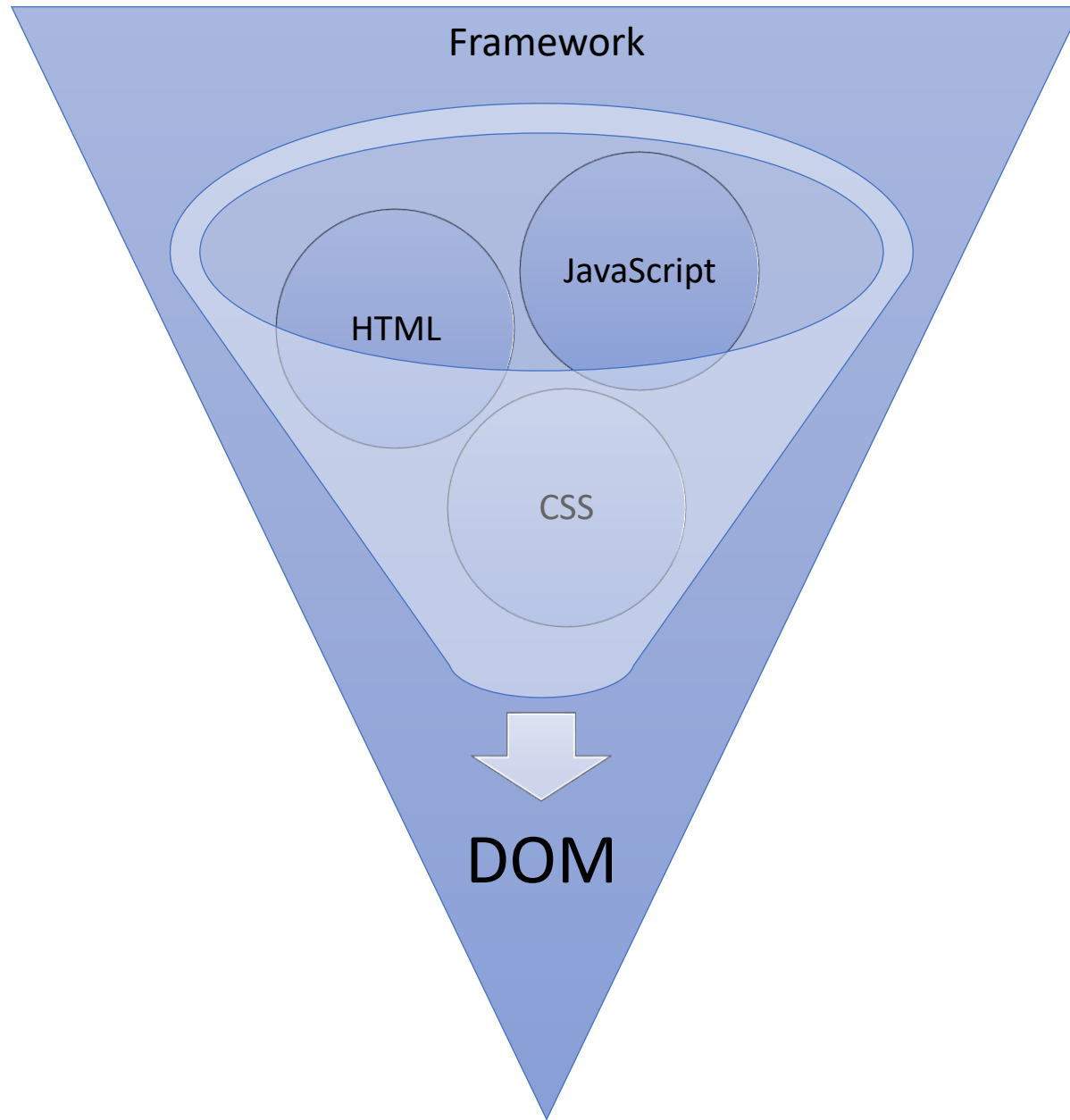


# DOM (Document Object Model)



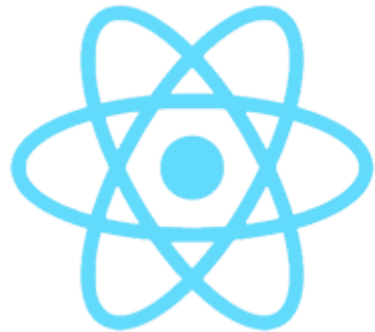
# What is a Framework?

- Software Framework designed to reduce overhead in web development
- Types of Framework Architectures
  - Model-View-Controller (MVC)
  - Push vs Pull Based
    - Most MVC Frameworks use push-based architecture “action based” (Django, Ruby on Rails, Symfony, Stripes)
    - Pull-based or “component based” (Lift, Angular2, React)
  - Three Tier Organization
    - Client (Usually the browser running HTML/Javascript/CSS)
    - Application (Running the Business Logic)
    - Database (Data Storage)
- Types of Frameworks
  - Server Side: Django, Ruby on Rails
  - Client Side: Angular, React, Vue



# Javascript Frameworks

- AngularJS/Angular 2
- ASP.net
- React
- Polymer 1.0
- Ember.js
- Vue.js



React



ember

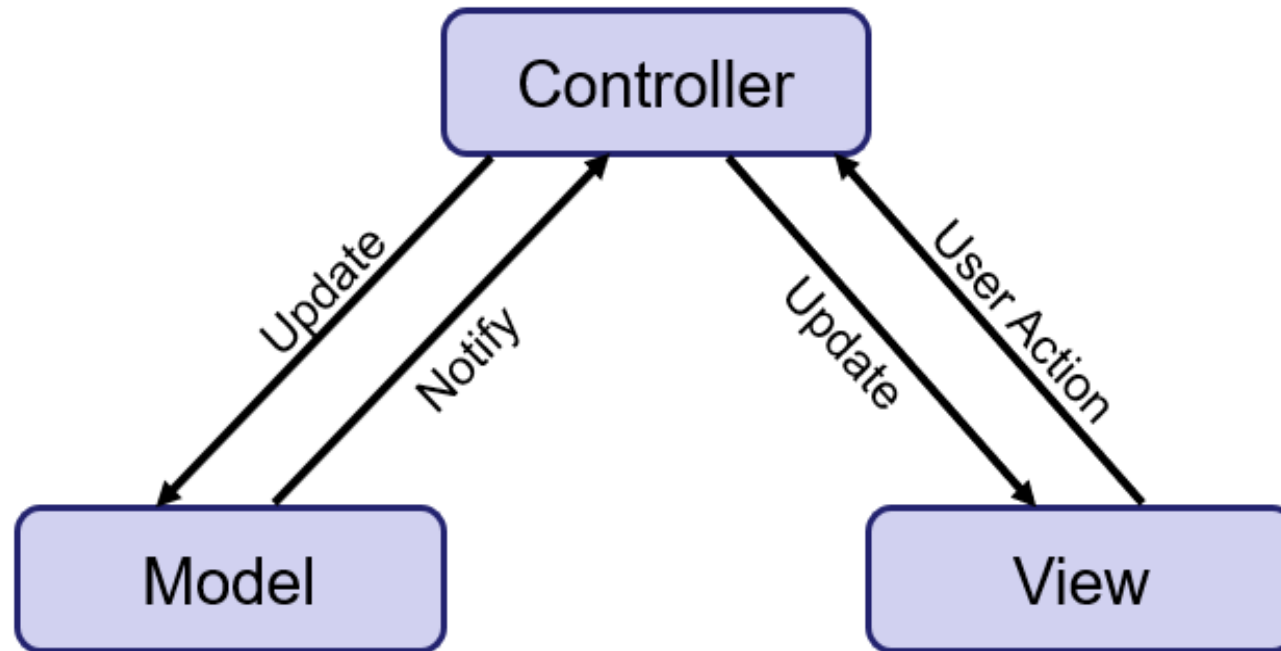


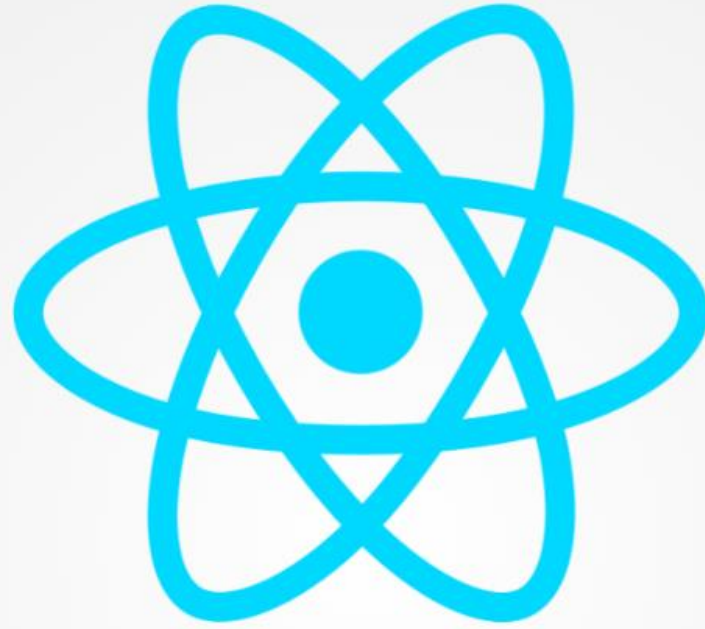


# MVC (Model View Controller)

- A Web Application Development Framework
- Model (M):
  - Where the data for the DOM is stored and handled)
  - This is where the backend connects
- View (V):
  - Think of this like a Page which is a single DOM
  - Where changes to the page are rendered and displayed
- Control (C):
  - This handles user input and interactions
    - Buttons
    - Forms
    - General Interface

# MVC Model



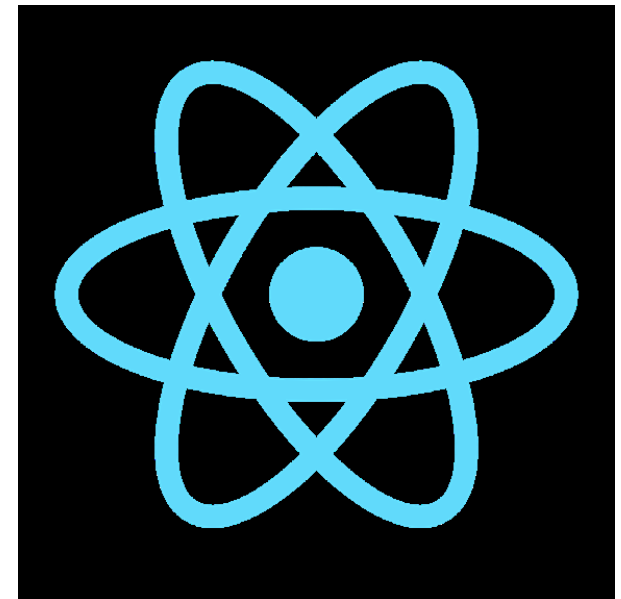


# **React.js**

## **fundamentals**

# What is React?

- React is a JavaScript framework
- Used for front end web development
- React is a view layer library, not a framework like Backbone, Angular etc.
- React was created by Jordan Walke, who was a software engineer at Facebook
- Famous for implementing a virtual dom



- React is a JavaScript library for building fast and interactive user interfaces for the web as well as mobile applications.
- It is an open-source, reusable component-based front-end library.
- In a model view controller architecture, React is the ‘View’ which is responsible for how the app looks and feels View Controller Model.

# Why was React developed?

- Complexity of two-way data binding
- A lot of data on a page changing over time
- Complexity of Facebook's UI architecture



# Why to use React?

- Easy creation of dynamic web applications
- Performance enhancements
- Reusable components
- Unidirectional data flow
- Small learning curve
- Can be used for mobile apps
- Dedicated tools for easy debugging

# Features of React

- **JSX** – JavaScript Syntax Extension

- JSX is a syntax extension to JavaScript. It is used with React to describe what the User Interface should look like
- By using JSX, you can write HTML structures in the same file that contains JavaScript code
- JSX helps in making the code easier to understand and debug as it avoids usage of JS DOM structures which are rather complex



- **Virtual DOM**

- Virtual DOM and Real DOM

- Notice that Virtual DOM is the exact copy of the real DOM
  - React keeps a lightweight representation of the Real DOM in the memory, and that is known as the Virtual DOM.
  - Manipulating the DOM is much slower than manipulating virtual DOM, because nothing gets drawn onscreen.
  - When the state of an object changes, Virtual DOM changes only that object in the real DOM instead of updating all the objects.

- **Performance**

- React uses Virtual DOM that makes the web apps fast.
  - Complex User Interface is broken down into individual components allowing multiple users to work on each component simultaneously.

- **One-way data binding**

- React's one-way data binding keeps everything modular and fast.
- A unidirectional data flow means that when designing a React app you often nest child components within parent components.



## • Extensions

- React goes beyond simple UI and has many extensions for complete application architecture support.
- It provides server-side rendering.
- Supports mobile app development.
- Extended with Flux and Redux, among others.

### Companies which uses React native



Thousands of apps are using React Native, from established Fortune 500 companies to new startups!

- **Debugging**

- React applications are extremely easy to test due to a large developer community.
- Facebook even provides a small browser extension that makes React debugging faster and easier



# Who Uses React?

React in the wild:

facebook®



Instagram



asana:



# Advantages of React

- Easy to understand what a component will render
  - Declarative code → predictable code
  - You don't really need to study JS in the view file in order to understand what the file does.
- Easy to mix HTML and JS
  - You do it already with template libraries (e.g. Handlebars, Mustache, Underscore etc.)
- Uses full power of JS
  - Decoupling templates from logic does not rely on the templates' primitive abstractions, but uses full power of JavaScript in displaying views
- No complex two-way data flow
  - Uses simple one-way reactive data flow
  - Easier to understand than two-way binding
  - Uses less code

- React is fast!
  - Real DOM is slow
  - JavaScript is fast
  - Using virtual DOM objects enables fast batch updates to real DOM, with great productivity gains over frequent cascading updates of DOM tree
- React dev tools
  - React Chrome extension makes debugging so much easier
- Server-side rendering
  - `React.renderToString()` returns pure HTML

# Disadvantages of React

- React is nothing but the view
  - No events
  - No XHR
  - No data / models
  - No promises / deferreds
  - No idea how to add all of the above
- Very little info in the docs
  - But it's not hard to learn
- Building JSX requires some extra work
  - But most of the work is already done for you by **react-tools**
- No support for older browsers
  - React won't work with IE8
  - There some polyfills / shims that help

# Why should I use React?

- Easy to read and understand views
- Concept of components is the *future* of web development
- If your page uses a lot of fast updating data or real time data - React is the way to go
- Once you and your team is over the React's learning curve, developing your app will become a lot faster

# React VS Angular

- **Data Binding:** Angular allows two-way data binding while React allows one-way data binding.
- **DOM Usage:** **Angular** uses the browser's DOM, while **React** uses a virtual DOM.
- **Language:** Angular is a JS framework by nature, but is built to use TypeScript. **React**, on the other hand, is a JavaScript library as well, but recommends using JSX.
- **Learning Curve:** This will differ from individual to individual based on skill and experience. On average, TypeScript is considered harder to learn than JSX, in turn increasing the learning curve with Angular as compared to React.
- **App Structure:** Angular is a fully-featured MVC framework. React is just more of a 'V' in the MVC.



- **Performance:** React.js holds JSX hence the usage of HTML codes and syntax is enabled. Angular on the other is a mere subset of html.
- **Written:** React.js written in JavaScript. Angular Written in Microsoft's Typescript language, which is a superset of ECMAScript 6 (ES6).
- **Preference:** React.js is preferred when the dynamic content needed is intensive. Angular is platform-independent and hence is compatible to work in any platform.
- **Flexibility:** **React** provides the developers with a lot of tools, libraries, and varying architecture to choose from. A skilled React team can select the tools they need at the beginning and deliver a highly customized app. **Angular** does not provide the flexibility that React does. Angular components can be used only within other frameworks, and the codes have to be embedded within an HTML application.

- **Developer's Perspective:**

- Angular is easy to step up but takes time to deliver projects since it has a steeper learning curve and uses a lot of unnecessary syntax for the simplest things, thus increasing coding time and delaying project deliveries.
- React takes longer to set up than Angular but lets you create projects and build apps relatively quickly.
- Plus, you get to add new features to React through different libraries, unlike Angular. React also lacks model and controller components, unlike Angular.

- **Community Support:**

- React has greater popularity among developers on GitHub and NPM. React received more than 135k stars on GitHub, which speaks for the popularity of this framework among the developers' community.
- But Stack OverFlow's study for 2018 suggested the greater popularity of Angular.
- Another survey pointed out that almost 75% of the React users would use it again.

## Popular Brands That Use Angular and React

### Angular



Google



Upwork



HBO



Forbes



Sony

### React



Facebook



Uber



Instagram



Dropbox



Netflix

# React VS Vue

- **Core differences:** The main difference between Vue.js and React is that Vue.js uses templates with declarative rendering while React uses JSX, which is a pretty much a JS extension that allows using HTML within it.
- **Popularity:** React has been a leader in popularity among JavaScript frameworks. It takes the first place with the number of 48,718 dependents, whereas, Vue.js is the second most popular JavaScript framework with half as many dependents 21,575, as reported by Node Package Manager.
- **Framework size:** The size of the framework is a crucial criterion that directly influences the productivity of projects, the smaller the framework is, the better it is for a project. If to round off the size of frameworks, React is around 100 Kb whereas Vue.js is around 80 Kb. Both of them have a relatively small size which makes them convenient for the development of small applications.



## Choose Vue.js if

You need to get a working solution as soon as possible (e.g. for startups)

Your app is pretty simple or has to be lightning-fast

You want to migrate an existing project to a new technology over a period of time but have limited resources

Your team consists mostly of HTML/junior developers to save money

Your developers prefer clean code and HTML templates



## Choose React if

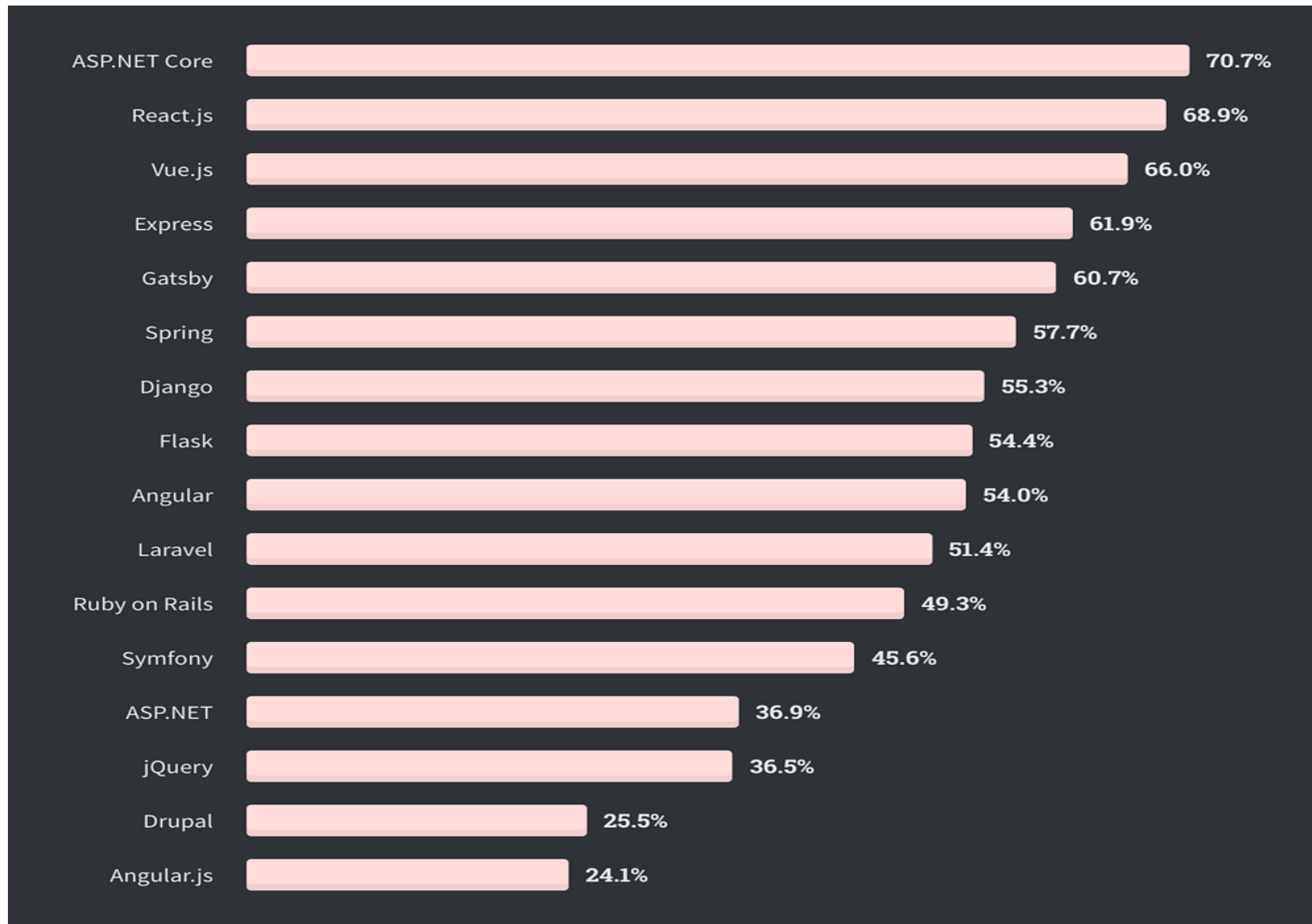
You want to build a complex enterprise-grade solution/SPA

You plan to greatly expand your application's functionality in the future and would need continuous support




You want to build a mobile app

You have a team of experienced React developers

Your developers prefer JavaScript over HTML



*Most-loved frameworks, libraries, and tools. Source: [Stack Overflow Survey](#)*

	 Angular	 React	 Vue
Framework size	143k	97.5k	58.8k
Programming Lang	Typescript	Javascript	Javascript
Ui component	In-built material techstack	React UI tools	Component libraries
Architecture	component-based	component-based	component-based
Learning curve	steep	moderate	moderate
Syntax	Real DOM	Virtual DOM	Virtual DOM
Scalability	modular development structure	component-based approach	template-based syntax
Migrations	API upgrade	React codemod script	Migration helper tool

# Installing React

- Before installing React we need to install Node.js first.
  - React gives you a language to **describe a user interface (UI)**. That interface could be the controls in your car, a modern fridge screen, or your microwave buttons. Basically, it could be any **interface** where a **user** needs to read and interact with an intelligent device.
  - React was initially made for the web user interface and it is most popular there. This is why we have the **ReactDOM** library that works with a browser's DOM. However, React today is also popular for non-Web interfaces like iOS, Android, virtual reality, and even command line interfaces.



- Node is the most popular **platform for tools** to make working with React easier. I think that's the case for 2 main reasons:
  - Node ships with a reliable package manager (NPM) and it works with the NPM registry (hosted at [npmjs.com](https://www.npmjs.com)). Using the NPM CLI, it is very easy install a package from the many available in the registry.
  - Node has reliable module dependency managers ("CommonJS" modules and ECMAScript modules). For CommonJS modules, this is basically the "require" function in Node combined with the "module" object. For ECMAScript modules, dependencies are managed with the import/export JavaScript syntax.
- Node is also a popular platform to run a web server that can host React applications.
- The **Webpack** Node package makes it very easy to bundle your multi-file React application into a single file for production and compile JSX (with **Babel**) during that process.

There are many other reasons to pick Node over other options as the web server to host your React applications. Let me highlight the most important ones:

- Most of the React examples and projects you will find on the web are based on Node. Getting help is easier when your project is using Node.
- Node is JavaScript, which you are already using (with React). You do not need to invest in any other languages. Using the same language allows for true code sharing between server-side and client-side code. You will even stop saying server-side and client-side eventually. It is all just JavaScript that is used on both sides. Hiring for your project also becomes easier because, well, JavaScript is popular.
- *Most importantly:* you can execute your React front-end code in a Node environment. This is the easiest option to do server-side rendering and have Universal/Isomorphic Applications.

# Commands to Install React

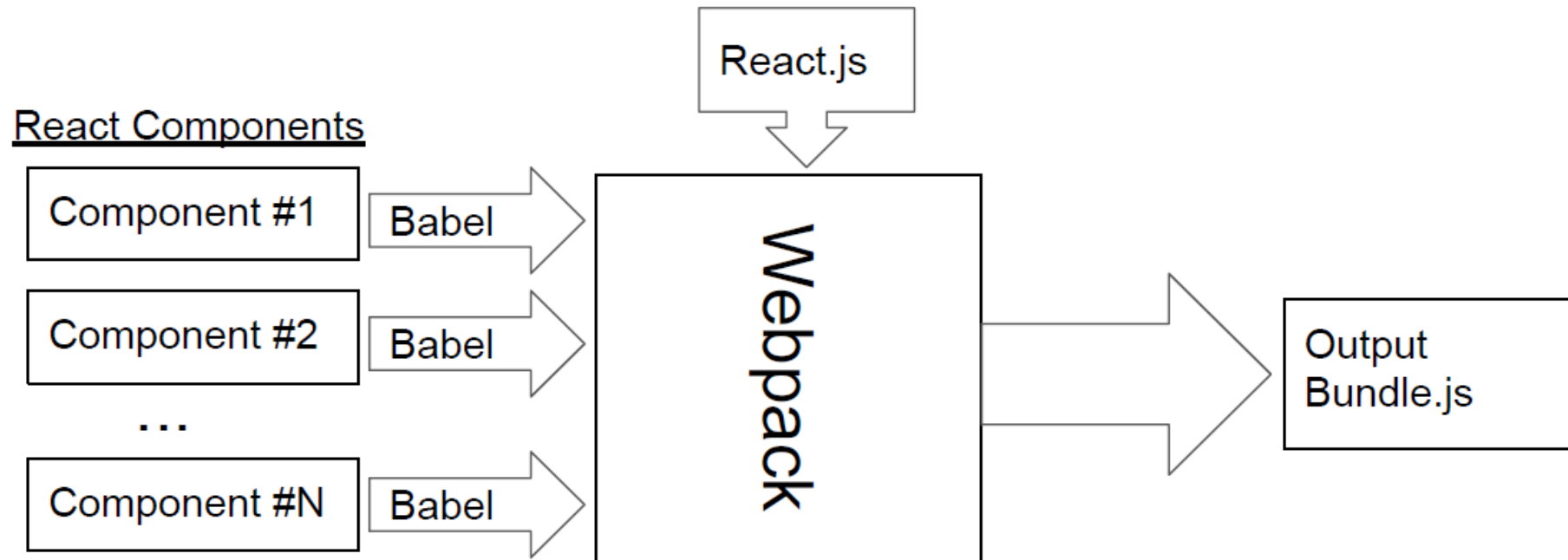
- `npm install -g create-react-app`
  - [create-react-app](#) allows you to bootstrap React applications with zero-configuration.
- `npx create-react-app my-app`
  - It's the simplest approach to learn React without worrying about all the tooling around it. You can bootstrap your first React.js application with npx (which comes via npm) on Windows with create-react-app by passing the name of your application to it on the command line.
- `Cd my-app`
- `Npm start`
  - The command line should give you an output where you can find the application in the browser. The default should be localhost:8080.

# Writing React Directly in HTML

- This method of interacting with React will be simplest way and it's very easy if you have ever worked with HTML, CSS and JavaScript.
- You'll have an HTML file where you load three scripts in the head element pointing to their respective CDN
  - React
  - ReactDOM
  - Babel.
- Then, you can create an empty div element and give it an id of root. This is where your application will live. Lastly, you'll create a script element where you write your React code.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Hello World</title>
<script src="https://unpkg.com/react@16/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
  <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
</head>
<body>
<div id="app"></div>
<script type="text/babel">
  ReactDOM.render(
    <h1>Welcome back to AWP Class</h1>,
    document.getElementById('app')
  );
</script>
</body>
</html>
```

# React tool chain



**Babel** - Transpile language features (e.g. ECMAScript, JSX) to basic JavaScript

**Webpack** - Bundle modules and resources (CSS, images)

Output loadable with single `script` tag in any browser

# Core Concept of Reactjs

JSX

Components

Unidirectional  
Data Flow

Virtual Dom

# Unidirectional Data Flow

- As compare to other MVC frameworks/Library Reactjs use the concept of unidirectional data flow.
- In Reactjs application the data flow from the parent to the children component by the state and the props.
- Only one parent is responsible to update the states and passing the value to the children components via props.
- setState is used to update/refresh the UI when the state change and the value can be pass to the children component by the this.props



# Virtual DOM

- Reactjs uses the concept of the virtual DOM.
- It selectively render the subtree of DOM elements into the rendering of the DOM on state change
- Use different algorithm with the browser DOM tree to identify the changes
- Instead of creating new object, Reactjs just identify what change is took place and once identify update that state.
- This way it is creating a virtual DOM and improving the performance too
- Can be render on server and sync on Local

# Without Components

- Earlier the developers had to write 1000 lines of code for developing a simple single page application.
- Most of those applications followed the traditional DOM structure and making changes to them was very challenging and a tedious task for the developers.
- They manually had to search for the element which needed the change and update it accordingly.
- Even a small mistake would lead to application failure. Moreover, updating DOM was very expensive.
- Thus, the component-based approach was introduced.

# Components


- Components are the building blocks of a React application that represent a part of the user interface.
- Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.
- Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.
- **Re-usability:** A component used in one area of the application can be reused in another area. This helps speed up the development process.
- **Nested Components:** A component can contain several other components.
- **Render method:** In its minimal form, a component must define a render method that specifies how the component renders to the DOM.
- **Passing properties:** A component can also receive props. These are properties passed by its parent to specify values

CartListComponent

CartComponent

CartItemComponent

Items in your cart

Product		Qty	Total	
	Product title \$50.00	<input type="text" value="1"/>	\$50.00	<button>Remove</button>
	Long Product title \$150.00	<input type="text" value="1"/>	\$150.00	<button>Remove</button>
	Product title \$100.00	<input type="text" value="2"/>	\$200.00	<button>Remove</button>

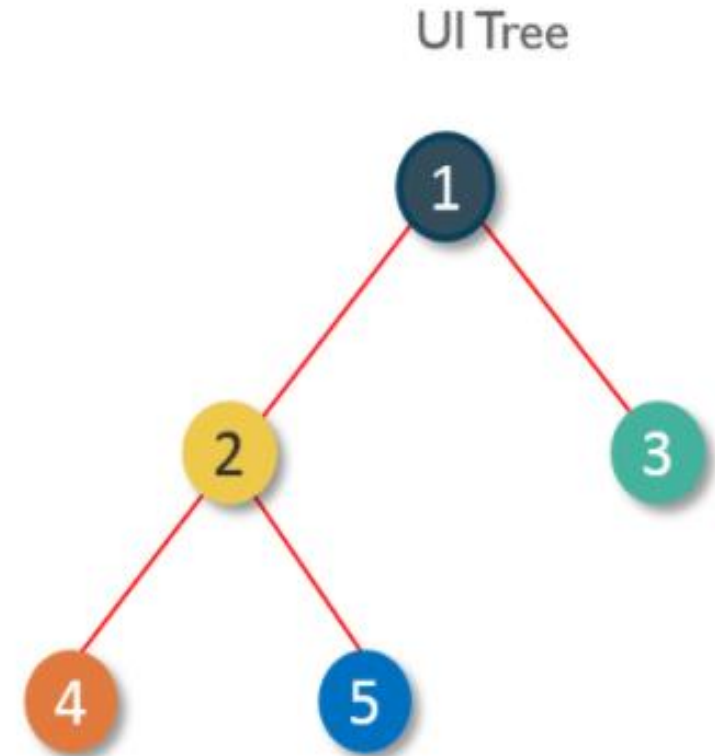
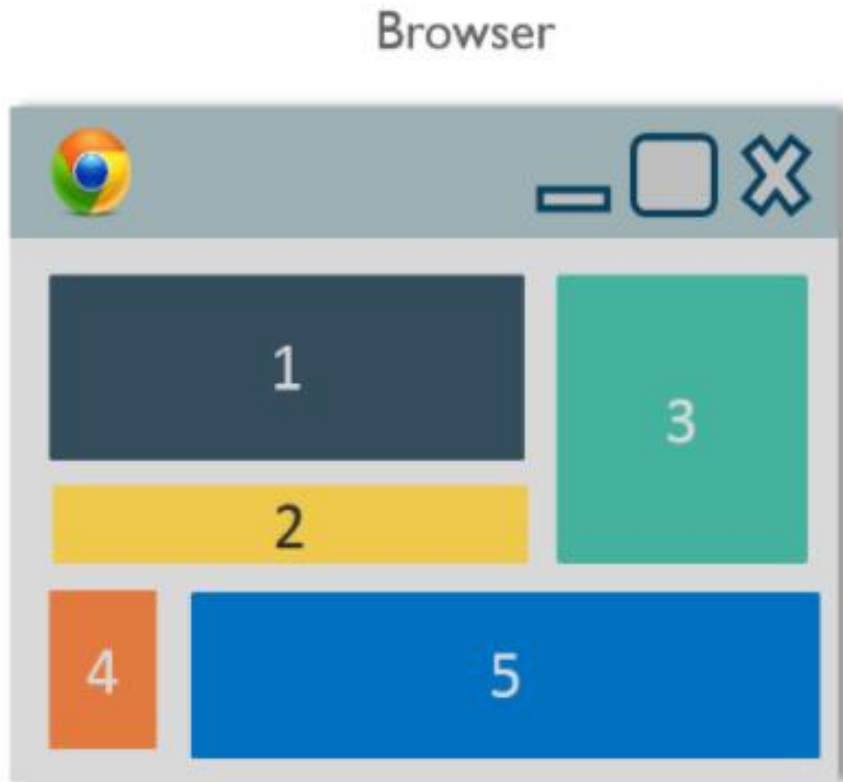
Update cart

Continue shopping

\$250.00

Checkout

ButtonComponent



- The first starting component becomes the root and each of the independent pieces becomes branches, which are further divided into sub-branches.

- This keeps the UI organized and allows the data and state changes to logically flow from the root to branches and then to sub-branches.
- Components make calls to the server directly from the client-side which allows the DOM to update dynamically without refreshing the page.
- Each component has its own interface that can make calls to the server and update them.
- As these components are independent of one another, each can refresh without affecting others or the UI as a whole.

# Creating Components

- **React.createClass()** method to create a component. This method must be passed an object argument which will define the React component.
- Each component must contain exactly one **render()** method.
- It is the most important property of a component which is responsible for parsing the HTML in JavaScript, JSX.
- This **render()** will return the HTML representation of the component as a DOM node. Therefore, all the HTML tags must be enclosed in an enclosing tag inside the **render()**.

# Example of creating Components

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

class MyComponent extends React.Component{
  render(){
    return(
      <div>
        <h1>Hello</h1>
        <h1>This is a Component</h1>
      </div>
    );
  }
}

ReactDOM.render(
  <MyComponent/>, document.getElementById('content')
);
```



# Advantages of React Components

1. **Code Re-usability** – A component-based approach makes your application development easier and faster. If you want to use a pre-existing functionality in your code, you can just put that code in yours instead of building it from scratch. It also allows your application architecture to stay up to date over time as you can update the specific areas which need up-gradations.
2. **Fast Development** – A component-based UI approach leads to an iterative and agile application development. These components are hosted in a library from which different software development teams can access, integrate and modify them throughout the development process.
3. **Consistency** – Implementing these reusable components helps to keep the design consistent and can provide clarity in organizing code throughout the application.
4. **Maintainability** – Applications with a set of well-organized components can be quickly updated and you can be confident about the areas which will be affected and which won't.
5. **Scalability** – The development becomes easier with a properly organized library of ready to implement components. Ensuring the components are properly namespaced helps to avoid style and functionality leaking or overlapping into the wrong place as the project scales up.
6. **Easy Integration** – The component codes are stored in repositories like GitHub, which is open for public use. Application development teams are well-versed in using source code repositories, and so they are able to extract the code as needed and inject it into the application.

# Types of React Components

- Components in React basically return a piece of JSX code that tells what should be rendered on the screen. In React, there are mainly two types of components:
  - Functional Components
  - Class Components

# Functional Components

- The first and recommended component type in React is functional components.
- A functional component is basically a JavaScript/ES6 function that returns a React element (JSX).



- Example of a valid functional component:

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

- Alternatively, you can also create a functional component with the arrow function definition:

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

- This function is a valid React component because it accepts a single “props” (which stands for properties) object argument with data and returns a React element.

- To be able to use a component later, you need to first export it so you can import it somewhere else:

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

- After importing it, you can call the component like in this example:

```
import Welcome from './Welcome';  
  
function App() {  
  return (  
    <div className="App">  
      <Welcome />  
    </div>  
  );  
}
```

# Functional Component in React:

- Is a JavaScript/ES6 function.
- Must return a React element (JSX).
- Always starts with a capital letter (naming convention).
- Takes props as a parameter if necessary.
- Solution without using state.
- We use Func components as much as possible.
- Mainly responsible for the UI
- Stateless/ Dumb/ Presentational

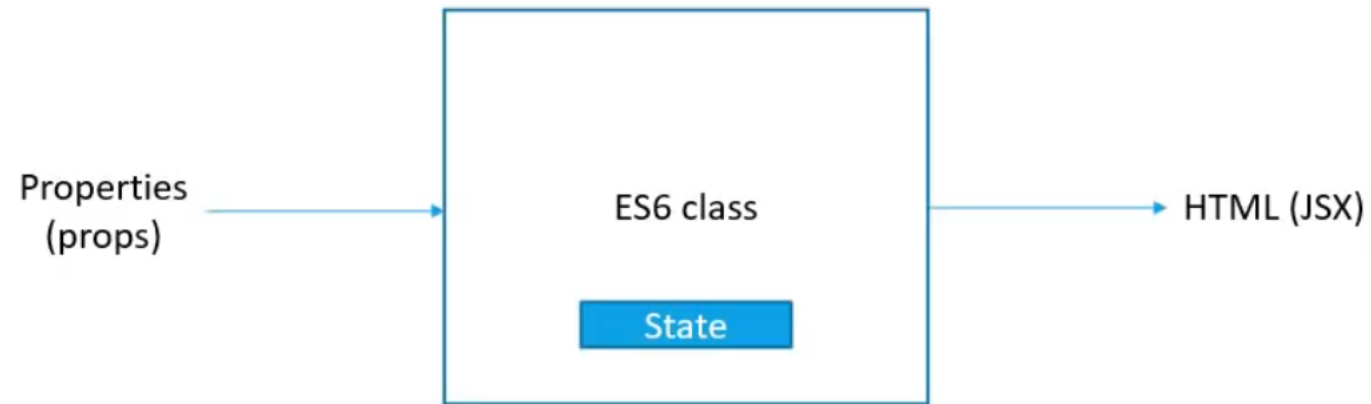
# Class Components

- The second type of component is the class component. Class components are ES6 classes that return JSX.

```
class Welcome extends React.Component {  
  
  render() {  
  
    return <h1>Hello, {this.props.name}</h1>;  
  
  }  
  
}
```

- Different from functional components, class components must have an additional render( ) method for returning JSX.

# Flow of Class Component





# Class Component in React:

- Is an ES6 class, will be a component once it ‘extends’ a React component.
- Takes Props (in the constructor) if needed
- Must have a render( ) method for returning JSX
- More feature rich
- Maintain their own private data – state
- Complex UI
- Provide lifecycle hooks
- Stateful/ Smart/ Container

# React Props

- React is a component-based library which divides the UI into little reusable pieces. In some cases, those components need to communicate (send data to each other) and the way to pass data between components is by using props.
- Props is short for properties and they are used to pass data between React components. React's data flow between components is uni-directional (from parent to child only).
- Props are arguments passed into React components.
- Props are passed to components via HTML attributes.
- React Props are like function arguments in JavaScript *and* attributes in HTML.

# Using Props in React

- We will see how to use Props step by step.
  - 1.Firstly, define an attribute and its value(data)
  - 2.Then pass it to child component(s) by using Props
  - 3.Finally, render the Props Data

# Example of Props

```
class ParentComponent extends Component {  
  render() {  
    return (  
      <ChildComponent name="First Child" />  
    );  
  }  
}
```

```
const ChildComponent = (props) => {  
  return <p>{props.name}</p>;  
};
```

- Firstly, we need to define/get some data from the parent component and assign it to a child component's "prop" attribute.

```
<ChildComponent name="First Child" />
```

- "Name" is a defined prop here and contains text data. Then we can pass data with props like we're giving an argument to a function:

```
const ChildComponent = (props) => {  
  // statements  
};
```

**Component:**  
**Greeting**

**Property:**  
**fullName**

```
class Greeting extends React.Component {  
  render() {  
    return (  
      <div className="greeting">  
        <h3>Hello {this.props.fullName}</h3>  
      </div>  
    );  
  }  
}
```

props-example1.jsx

```
<Greeting fullName='Tran' />
```

# State

- React has another special built-in object called state, which allows components to create and manage their own data. So unlike props, components cannot pass data with state, but they can create and manage it internally.

```
class Test extends React.Component {  
  constructor() {  
    this.state = {  
      id: 1,  
      name: "test"  
    };  
  }  
  render() {  
    return (  
      <div>  
        <p>{this.state.id}</p>  
        <p>{this.state.name}</p>  
      </div>  
    );  
  }  
}
```

# How do you update a component's state?

- State should not be modified directly, but it can be modified with a special method called `setState( )`.

```
this.state.id = "2020"; // wrong
```

```
this.setState({      // correct  
  id: "2020"  
});
```



# What happens when state changes?

- A change in the state happens based on user-input, triggering an event, and so on. Also, React components (with state) are rendered based on the data in the state. State holds the initial information.
- So when state changes, React gets informed and immediately re-renders the DOM – **not the whole DOM, but only the component with the updated state.** This is one of the reasons why React is fast.
- The `setState( )` method triggers the re-rendering process for the updated parts. React gets informed, knows which part(s) to change, and does it quickly without re-rendering the whole DOM.

There are 2 important points we need to pay attention to when using state:

- State shouldn't be modified directly – the `setState( )` should be used
- State affects the performance of your app, and therefore it shouldn't be used unnecessarily.

# Differences between props and state?

- Components receive data from outside with props, whereas they can create and manage their own data with state
- Props are used to pass data, whereas state is for managing data
- Data from props is read-only, and cannot be modified by a component that is receiving it from outside
- State data can be modified by its own component, but is private (cannot be accessed from outside)
- Props can only be passed from parent component to child (unidirectional flow)
- Modifying state should happen with the `setState ( )` method

# ES6

- ES6 stands for ECMAScript 6.
- ECMAScript was created to standardize JavaScript, and ES6 is the 6th version of ECMAScript, it was published in 2015, and is also known as ECMAScript 2015.
- **ECMAScript** stands for European Computer Manufacturer's Association. ECMAScript is a Standard for scripting languages such as JavaScript, JScript, etc.
- React uses ES6, and you should be familiar with some of the new features like:
  - Classes
  - Arrow Functions
  - Let
  - Const

# Variables(let, const)

- Till now, **var** keyword was used to declare variables in JavaScript.
- ES6 introduced two new ways to declare variables with **const** and **let**.
- When used to declare variables, they are scoped to the block and not the function; this means they are only available within that block.
- **const**: This variable can not be reassigned. JavaScript will throw an error if we try to reassign a new value.
- **let**: This variable can be reassigned. This is similar to var.

- example of using **let**:

```
let name = 'Ram';  
name = 'Sham';  
console.log(name);  
//output  
Sham
```

- Example of using **const**:

```
const cities = ["bangalore", "mysore", "dharward"];  
console.log(...cities);  
//output  
bangalore mysore dharward
```

-----  
**const** message = 'hello there!'; *// another example*

```
// error if we try to reassign message  
message = 'my new message'; // <- ERROR
```

# The spread operator

- The spread operator denoted by ... is used to expand iterable objects into multiple elements as shown in the previous example.
- The spread operator can also be used to combine multiple arrays into one array containing all array elements as shown below.

```
const east = ["Uganda", "Kenya", "Tanzania"];  
const west = ["Nigeria", "Cameroon", "Ghana"];
```

```
const countries = [...east, ...west];  
console.log(countries);
```

```
//output
```

```
[ 'Uganda', 'Kenya', 'Tanzania', 'Nigeria', 'Cameroon',  
  'Ghana' ]
```

# Arrow Functions (=>)

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

## Before:

```
function sayHello() {  
  return 'i am saying hello';  
}
```

## After:

```
const sayHello = () => {  
  return 'i am saying hello';  
}
```

```
// es6 implicit return 1-liner (same as above)  
const sayHello = () => 'i am saying hello';
```



- Arrow functions may be preferred because of the following:-
  - short syntax
  - they are easy to write and read
  - they automatically return when their body is a single line of code.
- Another example of arrow function:

```
function add(a, b) {  
  return a +_ b;  
}
```

```
// es6  
const add = (a, b) => a +_ b;
```

- **Note:** Functions and Arrow Functions are both valid and can be used throughout our code. It is up to you to determine which is the better one to use for each scenario.

# Classes

- A class is a type of function, but instead of using the keyword **function** to initiate it, we use the keyword **class**, and the properties are assigned inside a **constructor()** method.

```
// class
class App extends React.Component {
  state = { message: 'my message' };
  updateMessage = message => this.setState({ message });
  render() {
    const { message } = this.state;
    return (
      <div>my message is: {message}</div>
    );
  }
}
```

- With the new **class** syntax, less code is required to create a function. The function contains a clearly specified constructor function and all the code needed for the class is contained in its declaration.
- ES6 also introduced two new keywords, **super** and **extends** which are used to extend classes.
- To create a class inheritance, use the **extends** keyword.
- A class created with a class inheritance inherits all the methods from another class.
- The **super()** method refers to the parent class.
- By calling the **super()** method in the constructor method, we call the parent's constructor method and gets access to the parent's properties and methods.
- Note that classes in javascript are still functions and their behavior is not the same as those in object-oriented programming languages such as Java.

- React does not use inheritance except in the initial component class, which extends from the react package.
- Inheritance uses the keyword **extends** to allow any component to use the properties and methods of another component connected with the parent.
- Using the **extends** keyword, you can allow the current component to access all the component's properties, including the function, and trigger it from the child component.

# PraentClass.js

```
import React from "react";

class ParentClass extends React.Component {
  constructor(props) {
    super(props);
    this.callMe = this.callMe.bind(this);
  }

  // ParentClass function
  callMe() {
    console.log("This is a method from parent class");
  }

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

- ParentClass extends the component from React as `React.component`, which means the newly created component itself is using the inheritance. After creating parent class/component, create one child component, `Example.js`.
- The Example class extends ParentClass so the child class will access all the properties and methods created inside the parent component.

```
export default class Example extends ParentClass {  
  constructor() {  
    super();  
  }  
  render() {  
    this.callMe();  
    return false;  
  }  
}
```

- Here in the child class, the **this.callMe()** function is called the part of parent class implementation, so the parent component's properties and its methods can be accessed by implementing inheritance in the child component.

# React Component Lifecycle

- What is lifecycle methods and why it is important?
  - Around us everything goes through a cycle of taking birth, growing and at some point of time it will die.
  - The lifecycle methods are various methods which are invoked at different phases of the lifecycle of a component.
- We need more control over the stages that a component goes through.
- The process where all these stages are involved is called the **component's lifecycle** and every React component goes through it.
- React provides several methods that notify us when certain stage of this process occurs.
- These methods are called the **component's lifecycle methods** and they are invoked in a predictable order.
- React Lifecycle methods are the set of actions, which happen between the start and end of a **React Component**.

# Four phases of a React component

- The React component goes through the following phases
  - Initialization
  - Mount (Birth of your component)
  - Update (Growth of your component)
  - Unmount (Death of your component)

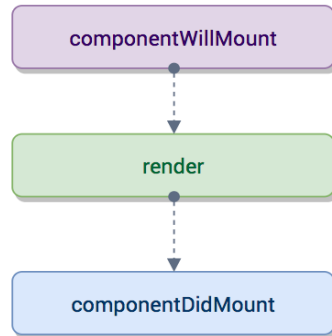


# Visual representation of the phases and the methods of ReactJs lifecycle.

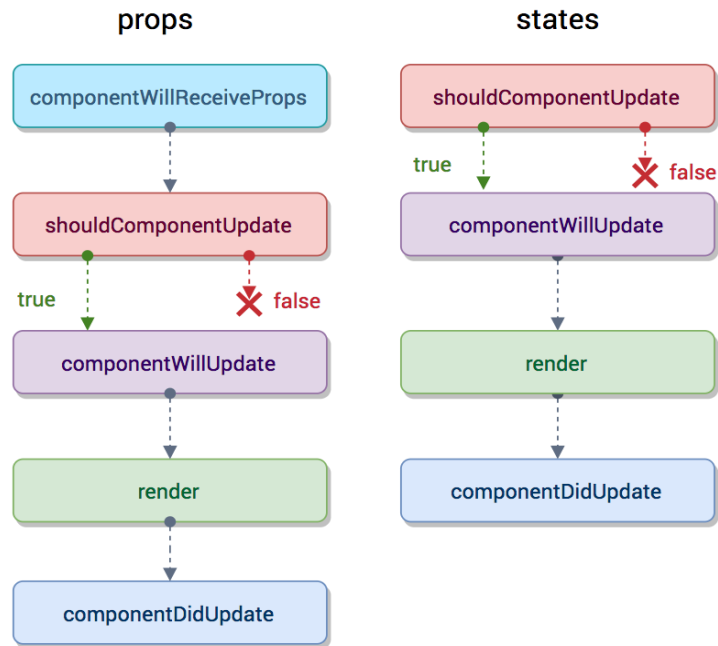
## Initialization

setup props and state

## Mounting



## Updation



## Unmounting



# Initialization

- It is the birth phase of the lifecycle of a ReactJS component. Here, the component starts its journey on a way to the DOM. In this phase, a component contains the default Props and initial State.
- The component is setting up the initial state in the constructor, which can be changed later by using the `setState` method.
- The initialization phase is where user define defaults and initial values for **this.props** and **this.state** by implementing **getDefaultProps()** and **getInitialState()** respectively.

- **getDefaultProps():** It is used to specify the default value of this.props. It is invoked before the creation of the component or any props from the parent is passed into it.
- **getInitialState():** It is used to specify the default value of this.state. It is invoked before the creation of the component.

# Mount

- After preparing with basic needs, state and props, our React Component is ready to mount in the browser DOM.
- This phase gives hook methods for before and after mounting of components.
- The methods which gets called in this phase are
  - `componentWillMount()`
  - `render()`
  - `componentDidMount()`

- **componentWillMount():**

- This method is executed just before the React Component is about to mount on the DOM.
- This method is executed once in a lifecycle of a component and before first render.
- **Usage:** This method is used for initializing the states or props, there is a huge debate going on to merge it with the constructor.
- In the case, when user call `setState()` inside this method, the component will not re-render.

- **render():**

- This method mounts the component onto the browser.
- This is a pure method, which means it gives the same output every time the same input is provided.
- This method is defined in each and every component. It is responsible for returning a single root HTML node element. If you don't want to render anything, you can return a null or false value.

- **componentDidMount():**

- This is the hook method which is executed after the component did mount on the dom.
- This method is executed once in a lifecycle of a component and after the first render.
- As, in this method, user can access the DOM.
- **Usage:** this is the right method to integrate API.
- This is invoked immediately after a component gets rendered and placed on the DOM. Now, user can do any DOM querying operations.

# Update

- This phase starts when the react component has taken birth on the browser and grows by receiving new updates.
- This phase also allows to handle user interaction and provide communication with the components hierarchy. The main aim of this phase is to ensure that the component is displaying the latest version of itself.
- Unlike the Birth or Death phase, this phase repeats again and again.
- The component can be updated by two ways:
  - Sending new props
  - Updating the state



- sending new props:
  - The methods which gets called in this phase are :
    - `componentWillReceiveProps()`
    - `shouldComponentUpdate()`
    - `componentWillUpdate()`
    - `render()`
    - `componentDidUpdate()`

- **componentWillReceiveProps():**

- This method gets executed when the props have changed and is not first render.
- Sometimes state depends on the props, hence whenever props changes the state should also be synced, This is the method where it should be done.
- **Usage:** This is how the state can be kept synced with the new props.
- It is invoked when a component receives new props. If you want to update the state in response to prop changes, you should compare `this.props` and `nextProps` to perform state transition by using **`this.setState()`** method.

- **shouldComponentUpdate():**

- This method tells the React that when the component receives new props or state is being updated, should React re-render or it can skip rendering?
- It is invoked when a component decides any changes/updation to the DOM. It allows you to control the component's behavior of updating itself. If this method returns true, the component will update. Otherwise, the component will skip the updating.

- **componentWillUpdate():**

- This method is executed only after the `shouldComponentUpdate()` returns true.
- This method is only used to do the preparation for the upcoming render, similar to **`componentWillMount()`** or constructor.
- It is invoked just before the component updating occurs. Here, you can't change the component state by invoking **`this.setState()`** method. It will not be called, if **`shouldComponentUpdate()`** returns false.

- **render():**

- This method is mounts the component onto the browser.
- It is invoked to examine this.props and this.state and return one of the following types: React elements, Arrays and fragments, Booleans or null, String and Number. If shouldComponentUpdate() returns false, the code inside render() will be invoked again to ensure that the component displays itself properly.

- **componentDidUpdate():**

- This method is executed when the new updated component has been updated in the DOM.
- This method is used to re trigger the third party libraries used to make sure these libraries also update and reload themselves.
- It is invoked immediately after the component updating occurs. In this method, you can put any code inside this which you want to execute once the updating occurs. This method is not invoked for the initial render.

- updating the state
  - The methods which gets called in this phase are :
    - `shouldComponentUpdate()`
    - `componentWillUpdate()`
    - `render()`
    - `componentDidUpdate()`

# Unmount

- In this phase, the component is not needed and the component will get unmounted from the DOM.
- It is the final phase of the react component lifecycle. It is called when a component instance is destroyed and unmounted from the DOM.
- The method which is called in this phase :
  - `componentWillUnmount()`

- **componentWillUnmount():**

- This method is the last method in the lifecycle.
- This is executed just before the component gets removed from the DOM.
- **Usage:** In this method, we do all the cleanups related to the component.
- This method is invoked immediately before a component is destroyed and unmounted permanently. It performs any necessary cleanup related task such as invalidating timers, event listener, canceling network requests, or cleaning up DOM elements. If a component instance is unmounted, you cannot mount it again.
  - For example, on logout, the user details and all the auth tokens can be cleared before unmounting the main component.

# Destructuring in Reactjs

- Destructuring is a convenient way of accessing multiple properties stored in objects and arrays. It was introduced to JavaScript by ES6 and has provided developers with an increased amount of utility when accessing data properties in Objects or Arrays.
- Destructuring when used it does not modify an object or array but rather copies the desired items from those data structures into variables. These new variables can be accessed later on in a React component.
- Example: Imagine you have a person object with the following properties:

```
const person = {  
  firstName: "React",  
  lastName: "JS",  
  city: "Bangalore"  
}
```



- Before ES6, you had to access each property individually:

```
console.log(person.firstName) // React  
console.log(person.lastName) // JS  
console.log(person.city) // Bangalore
```

- Destructuring lets us streamline this code:

```
const { firstName, lastName, city } = person;
```

- Is equivalent to:

```
const firstName = person.firstName  
const lastName = person.lastName  
const city = person.city
```

- So now we can access these properties without the person. prefix:

```
console.log(firstName) // React  
console.log(lastName) // JS  
console.log(city) // Bangalore
```

# What are the benefits?

- Destructuring enables faster access to nested data. A developer no longer has to iterate over an Array or object multiple times to retrieve properties of its nested children.
- Some React components have multiple ternary operators within its render function. Having single word destructured variables vastly improves a components readability.
- Improving readability is more useful then one may initially think. Essentially, the developer is eliminating an additional helper to loop through nested data in React's props.
- Not only cutting down the amount of code used but also providing a component with the exact data properties it needs in readily accessible variables.
- As with most ES6 improvements, destructuring aims to make the developers life that bit easier. It is certainly a quality of life improvement when considering the improved readability and reduction of written code.

# Event Handling

- Event handlers determine what action is to be taken whenever an event is fired. This could be a button click or a change in a text input.
- Essentially, event handlers are what make it possible for users to interact with your React app.
- Handling events with React elements is similar to handling events on DOM elements, with a few minor exceptions. If you're familiar with how events work in standard HTML and JavaScript, it should be easy for you to learn how to handle events in React.

# Binding Event Handlers

- There are four different ways to bind event handlers in React components:
  - Binding Event Handler in Render Method.
  - Binding Event Handler using Arrow Function.
  - Binding Event Handler in the Constructor.
  - Binding Event Handler using Arrow Function as a Class Property.

- **Binding Event Handler in Render Method:** User can bind the handler when it is called in the render method using *bind()* method.
- **Binding Event Handler using Arrow Function:** This is pretty much the same approach as above, however, in this approach user will be binding the event handler implicitly. This approach is the best if users want to pass parameters to your event.
- **Binding Event Handler in the Constructor:** In this approach, user will be going to bind the event handler inside the constructor. This is also the approach that is mentioned in **ReactJS documentation**. This has performance benefits as the events aren't binding every time the method is called, as opposed to the previous two approaches.
- **Binding Event Handler using Arrow Function as a Class Property:** This is probably the best way to bind the events and still pass in parameters to the event handlers easily.

# Methods as Props

- Earlier we saw how parent components can pass props to its child components.
- What if a child component wants to communicate with the parent component can be done?. It can be done by using props.
- But this time we pass a reference to a method as props to the child component.

# Conditional Rendering

- Conditional rendering is the ability to render different UI markup based on certain conditions.
- It is a way to render different elements or components based on a condition
- Conditional rendering in React works the same way conditions work in JavaScript.

- The Conditional Rendering concept can be applied often in the following scenarios:
  - Rendering external data from an API.
  - Showing or hiding elements.
  - Toggling application functionality.
  - Implementing permission levels.
  - Handling authentication and authorization.



# Types of Conditional rendering

1.if/else

2.Element Variables

3.Ternary Conditional Operator

4.Short Circuit Operator

## **1. if/else:**

- if/else cannot be used inside return statement and JSX
- You can use if/else statement in render()

## **2. Element Variables:**

- Element variables is a simple variable which can hold the JSX element and it can be rendered when required.

## **3. Ternary Conditional Operator:**

- You can use JavaScript in JSX, but it becomes difficult when using statements like if, else within JSX.
- Hence, if you want to use if/else statement in JSX then you can use ternary conditional operator

## **4. Short Circuit Operator:**

- Short Circuit Operator includes the JavaScript logical && operator.
- Therefore, if the condition is true, the element right after && will appear in the output. If it is false, React will ignore and skip it.

# List Rendering

- Lists are used to display data in an ordered format and mainly used to display menus on websites.
- In React, Lists can be created in a similar way as we create lists in JavaScript.
- A map is a data collection type where data is stored in the form of pairs. It contains a unique key.
- The value stored in the map must be mapped to the key. We cannot store a duplicate pair in the map().
- It is because of the uniqueness of each stored key. It is mainly used for fast searching and looking up data.

# Example of Map()

```
<script type="text/javascript">  
  var numbers = [1,2,3,4,5];  
  const updatedNums = numbers.map((number)=>{  
    return (number + 2);  
  });  
  console.log(updatedNums);  
</script>
```

# Keys

- A “Key” is a special string attribute you need to include when creating lists of elements.
- Keys give the elements a stable identity.
- Keys helps React identify which items have changed, are added, or are removed.
- It helps in efficient update of the user interface.

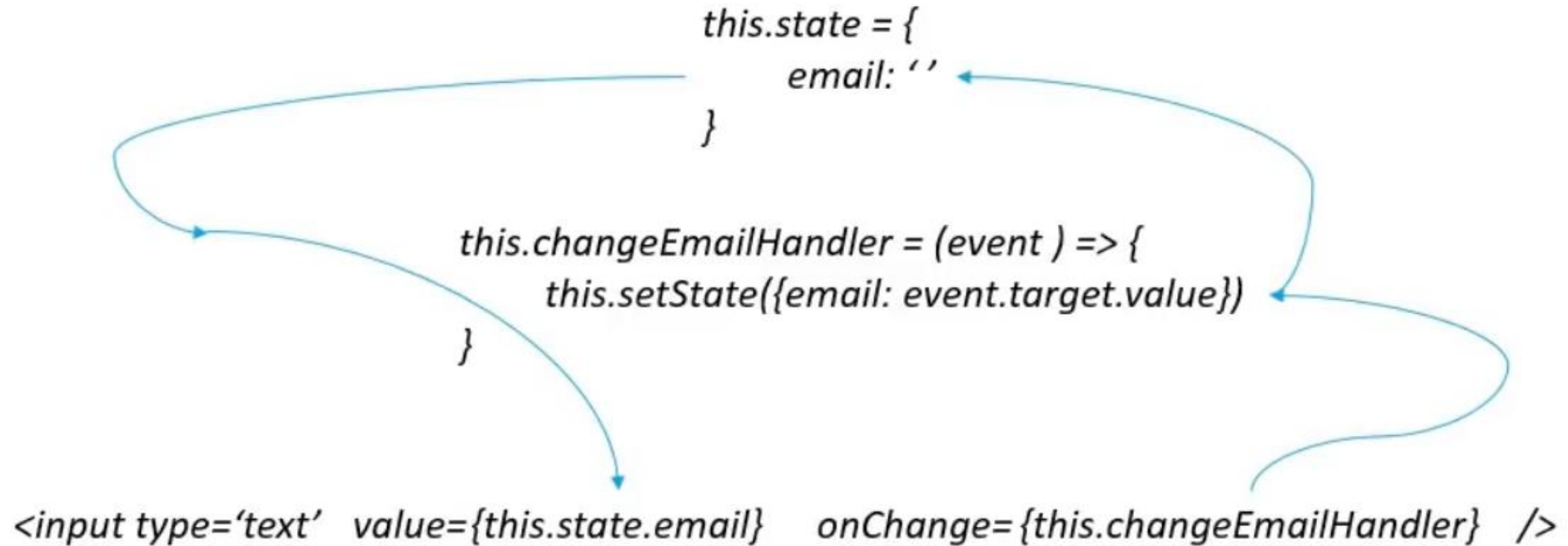
# Styling React Components

1. CSS Stylesheets
2. Inline Styling
3. CSS Modules
4. CSS in JS Libraries

# ReactJS - Forms

- Forms are a crucial component of React web applications.
- They allow users to directly input and submit data in components ranging from a login screen to a checkout page.
- Since most React applications are *single page applications* (SPAs), or web applications that load a single page through which new data is displayed dynamically, you won't submit the information directly from the form to a server.
- React forms present a unique challenge because you can either allow the browser to handle most of the form elements and collect data through React change events, or you can use React to fully control the element by setting and updating the input value directly.
- The first approach is called an uncontrolled component because React is not setting the value. The second approach is called a controlled component because React is actively updating the input.

# Controlled Components





# React Fragments

- React Fragments allow you to wrap or group multiple elements without adding an extra node to the DOM.
- This can be useful when rendering multiple child elements / components in a single parent component.
- React fragments let you group a list of children without adding extra nodes to the DOM because fragments are not rendered to the DOM.
- React components can only render one element, and if you have multiple elements, the common practice is to wrap them in a single root element, usually a `<div>` wrapper.
- This workaround works for most cases, but sometimes adding an extra DOM element is not feasible.

- React Fragments do not produce any extra elements in the DOM, which means that a Fragment's child components will be rendered without any wrapping DOM node. React Fragments enable you to group multiple sibling components without introducing any unnecessary markup in the rendered HTML.
- React Fragments serve as a cleaner alternative to using unnecessary **div**'s in your code.

- For example we could have a component Table that renders an HTML table and inside that table the columns are rendered with another component called Columns. It would probably look something like this:

```
class Table extends React.Component {  
  render() {  
    return (  
      <table>  
        <tr>  
          <Columns />  
        </tr>  
      </table>  
    );  
  }  
}
```

```
class Columns extends React.Component {  
  render() {  
    return (  
      <div>  
        <td>Hello</td>  
        <td>World</td>  
      </div>  
    );  
  }  
}
```

- This would result in an invalid HTML to be rendered because the wrapper div from Columns component is rendered inside the <tr>.

```
<table>
  <tr>
    <div>
      <td>Hello</td>
      <td>World</td>
    </div>
  </tr>
</table>
```

- If we wrap the `<td>` elements in a `<div>`, this will add a `<div>` element in middle of `<tr>` and `<td>` and break the parent-child relationship.
- In order to correctly render the HTML, `<Column />` must return multiple `<td>` elements without any extra element in between the `<tr>` and `<td>` tags.

# Solution

- React fragments let you group a list of children without adding extra nodes to the DOM because fragments are not rendered to the DOM.
- So basically we use `React.Fragment` where we would normally use a wrapper `div`.
- We can make use of fragments with `<React.Fragments>` syntax. So we could write the `Columns` component as follows.

```
class Columns extends React.Component {  
  render() {  
    return (  
      <React.Fragment>  
        <td>Hello</td>  
        <td>World</td>  
      </React.Fragment>  
    );  
  }  
}
```

- Now the Table component would render following HTML.

```
<table>
  <tr>
    <td>Hello</td>
    <td>World</td>
  </tr>
</table>
```

- Fragments can also be declared with a short syntax which looks like an empty tag. Here is an example.

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

# Keyed Fragments

- The shorthand syntax does not accept key attributes. You need a key for mapping a collection to an array of fragments such as to create a description list.
- If you need to provide keys, you have to declare the fragments with the explicit **<React.Fragment>** syntax.

## Why to use Fragments

- 1.It makes the execution of code faster as compared to the div tag.
- 2.It takes less memory.

# React Refs

- Refs is the shorthand used for **references** in React.
- It is an attribute which makes it possible to store a reference to particular DOM nodes or React elements.
- It provides a way to access React DOM nodes or React elements and how to interact with it.
- It is used when we want to change the value of a child component, without making the use of props.



# When to Use Refs

- Refs can be used in the following cases:
  - When we need DOM measurements such as managing focus, text selection, or media playback.
  - It is used in triggering imperative animations.
  - When integrating with third-party DOM libraries.
  - It can also use as in callbacks.

# Callback refs

- In react, there is another way to use refs that is called "callback refs" and it gives more control when the refs are set and unset.
- Instead of creating refs by `createRef()` method, React allows a way to create refs by passing a callback function to the `ref` attribute of a component.
  - `<input type="text" ref={element => this.callRefInput = element} />`
- The callback function is used to store a reference to the DOM node in an instance property and can be accessed elsewhere. It can be accessed as below:
  - `this.callRefInput.value`

# Forwarding Ref from one component to another component

- Ref forwarding is a technique that is used for passing a ref through a component to one of its child components. It can be performed by making use of the **React.forwardRef()** method.
- This technique is particularly useful with higher-order components and specially used in reusable component libraries.

# High Order Components

- A higher-order component (HOC) is an advanced technique in React for reusing component logic.
- HOCs are commonly used to design components with certain shared behavior in a way that makes them connected differently than normal state-to-props pattern.
- Concretely, a higher-order component is a function that takes a component and returns a new component.
  - `const EnhancedComponent = higherOrderComponent(WrappedComponent);`
- Whereas a component transforms props into UI, a higher-order component transforms a component into another component.

# Structure Of A Higher-Order Component

A HOC is structured like a higher-order function:

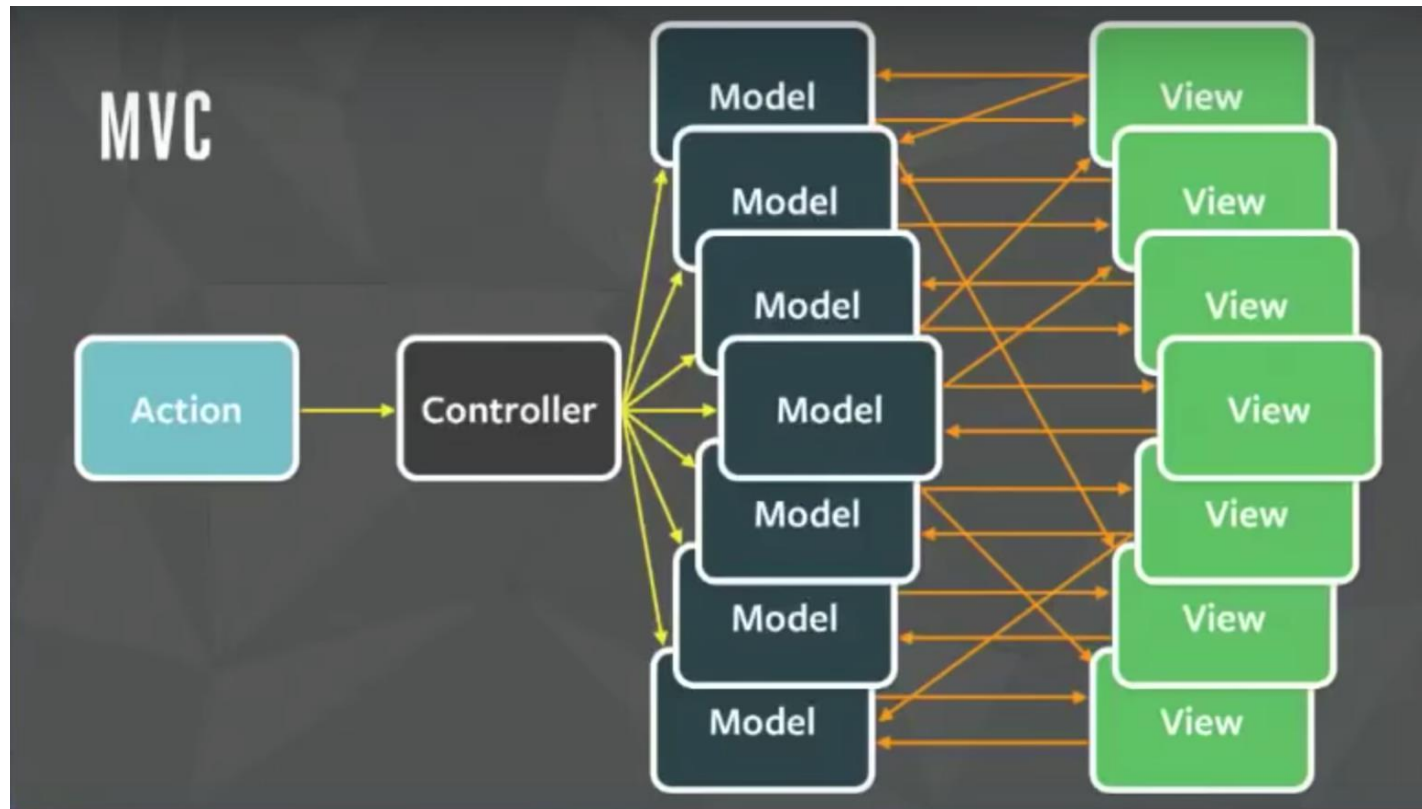
- It is a component.
- It takes another component as an argument.
- Then, it returns a new component.
- The component it returns can render the original component that was passed to it.

# Flux

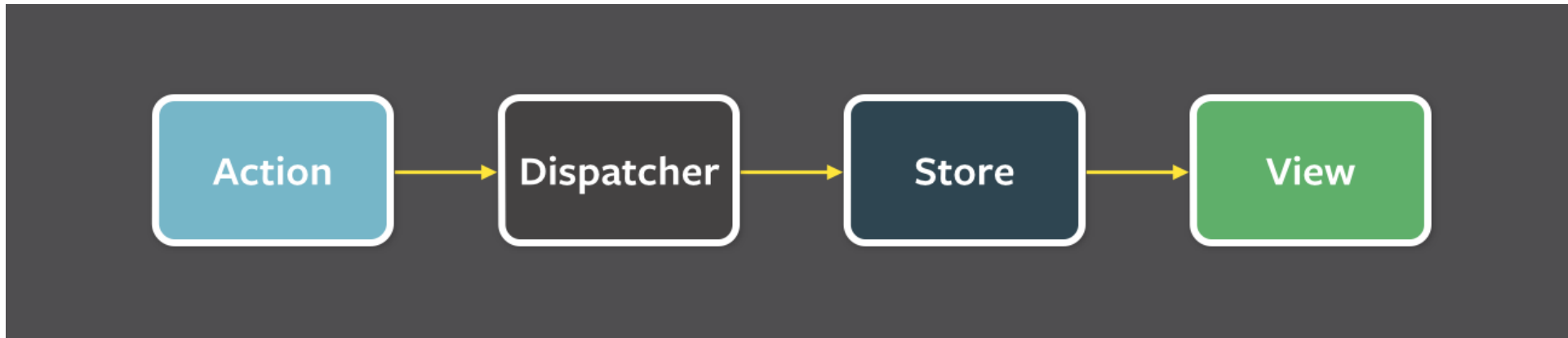
- Flux uses a unidirectional data flow pattern to solve state management complexity. Remember it is not a framework – rather it's more of a pattern that targets to solve the state management issue.
- Flux is a pattern for managing how data flows through a React application. As we've seen, the preferred method of working with React components is through passing data from one parent component to its children components. The Flux pattern makes this model the default method for handling data.

# Why Flux? Already React MVC Framework?

- Imagine your client's application scales up. You have interaction between many models and views. How would it look?



- The relationship between components gets complicated. It becomes hard to scale the application. Facebook faced the same issue.
- To solve this issue they architected a Single directional data flow.



- There are four distinct roles for dealing with data in the flux methodology:
  - Dispatcher
  - Stores
  - Views (our components)
  - Action

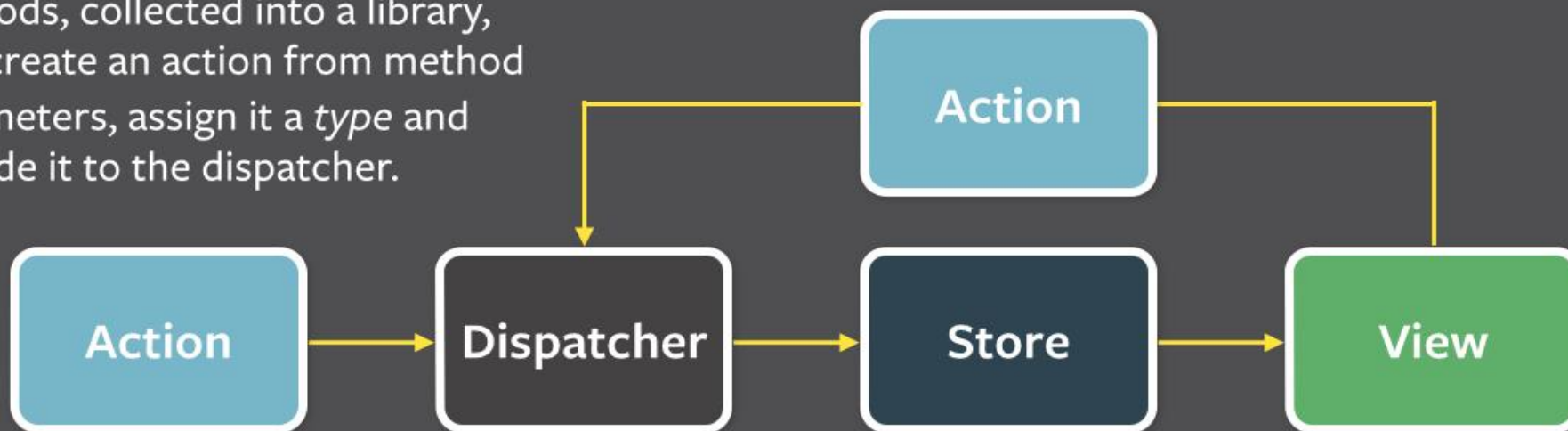


- **View:** this component renders the UI. Whenever any user interaction occurs on it (like an event) then it fires off the action. Also when the Store informs the View that some change has occurred, it re-renders itself. For example, if a user clicks the Add button.
- **Action:** this handles all the events. These events are passed by the view component. This layer is generally used to make API calls. Once the action is done it is dispatched using the Dispatcher. The action can be something like add a post, delete a post, or any other user interaction.
- **Dispatcher:** is basically the manager of this entire process. It is the central hub for your application. The dispatcher receives actions and dispatches the actions and data to registered callbacks.
- **Store:** this holds the app state and is a data layer of this pattern. Do not consider it as a model from MVC. An application can have one or many app stores. Stores get updated because they have a callback that is registered using the dispatcher.

- The major idea behind Flux is that there is a single-source of truth (the stores) and they can only be updated by triggering actions. The actions are responsible for calling the dispatcher, which the stores can subscribe for changes and update their own data accordingly.
- When a dispatch has been triggered, and the store updates, it will emit a change event which the views can rerender accordingly.
- This may seem unnecessarily complex, but the structure makes it incredibly easy to reason about where our data is coming from, what causes its changes, how it changes, and lets us track specific user flows, etc.
- The key idea behind *Flux* is: “Data flows in one direction and kept entirely in the stores.”

# Structure and Data Flow

*Action creators* are helper methods, collected into a library, that create an action from method parameters, assign it a *type* and provide it to the dispatcher.



Every action is sent to all stores via the *callbacks* the stores register with the dispatcher.

After stores update themselves in response to an action, they emit a *change* event.

Special views called *controller-views*, listen for *change* events, retrieve the new data from the stores and provide the new data to the entire tree of their child views.

# Flux brings various key benefits

- 1.The code becomes quite clear and easy to understand.
- 2.Easily testable using Unit Test.
- 3.Scalable apps can be built.
- 4.Predictable data flow.

**Note:** The only drawback with the Flux is that there is some boilerplate that we need to write. Besides the boilerplate, there is little code we need to write when adding components to the existing application.

In programming, the term boilerplate code refers to blocks of code used over and over again.

# Redux



# Getting Started with Redux

- “Redux is a predictable state container for JavaScript apps.”
- Let’s divide it into three parts
  - It is for JavaScript apps
  - It is a state container
  - It is predictable

# Redux is for JavaScript apps

- Redux is not tied to React.
- It can be used with React, Angular, and Vue etc.
- Redux is a library for JavaScript applications

# Redux is a state container

- Redux stores the state of your application
- Consider a React app – state of a component
- State of an app is the state represented by all the individual components of that app
- Redux will store and manage the application state

## LoginFormComponent

```
state = {  
  username: '',  
  password: '',  
  submitting: false  
}
```

## UserListComponent

```
state = {  
  users: []  
}
```

## Application

```
state = {  
  isUserLoggedIn: true,  
  username: 'Vishwas',  
  profileUrl: '',  
  onlineUsers: [],  
  isModalOpened: false  
}
```



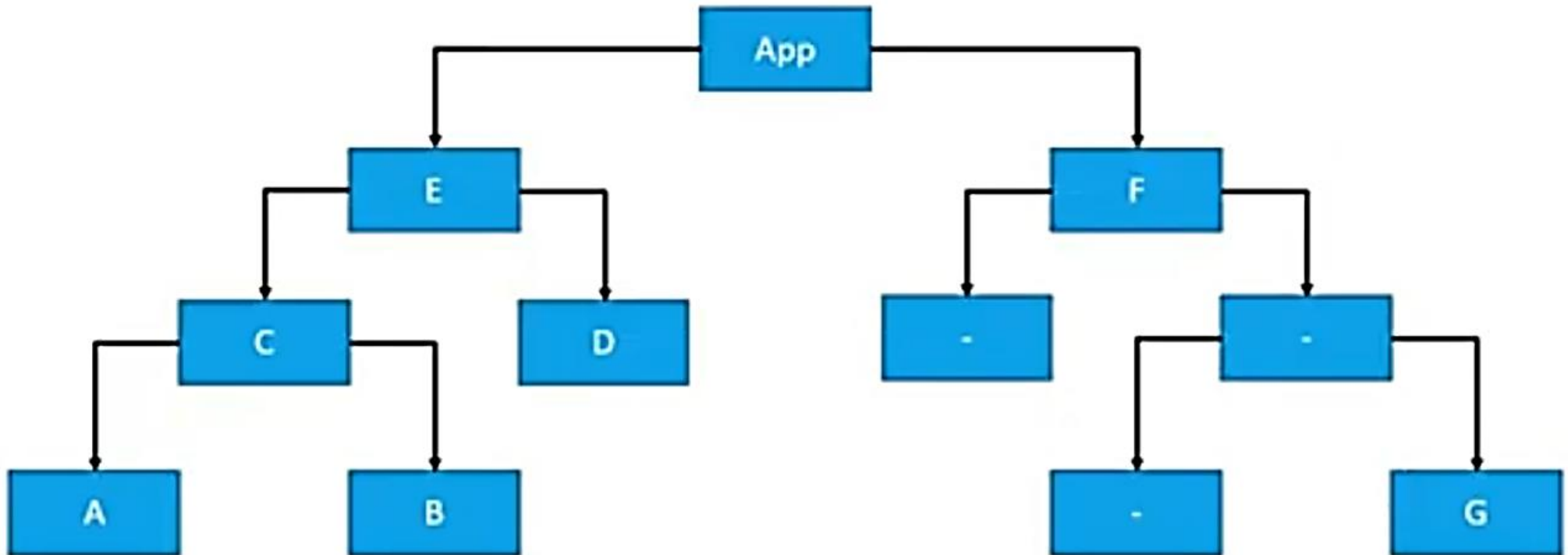
# Redux is Predictable

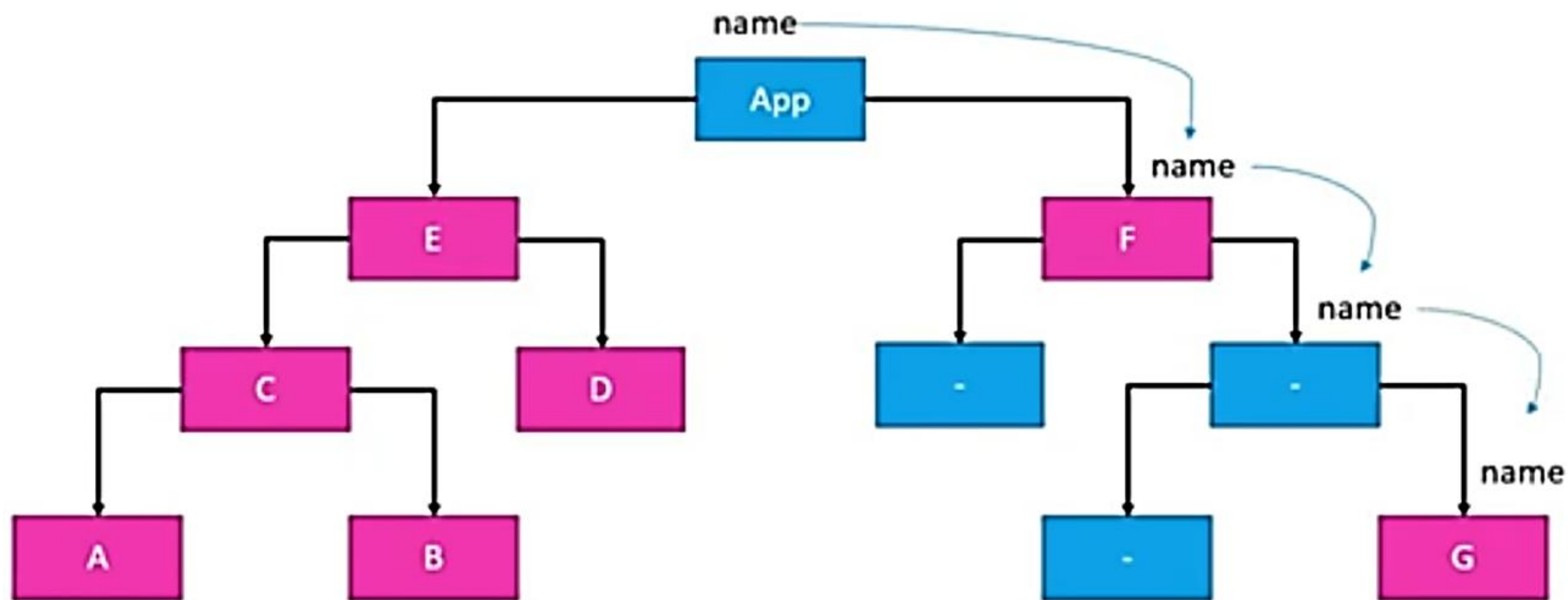
- Predictable in what way?
- Redux is a state container
- The state of the application can change.
- Ex: Counter application – Item (default)-> item(Increment or Decrement)
- In redux , all state transitions are explicit and it is possible to keep track of them
- The changes to your application's state become predictable

# React + Redux

- Why would we want to use redux in react application?
- Components in React have their own state
- Why do we need another tool to help manage that state?

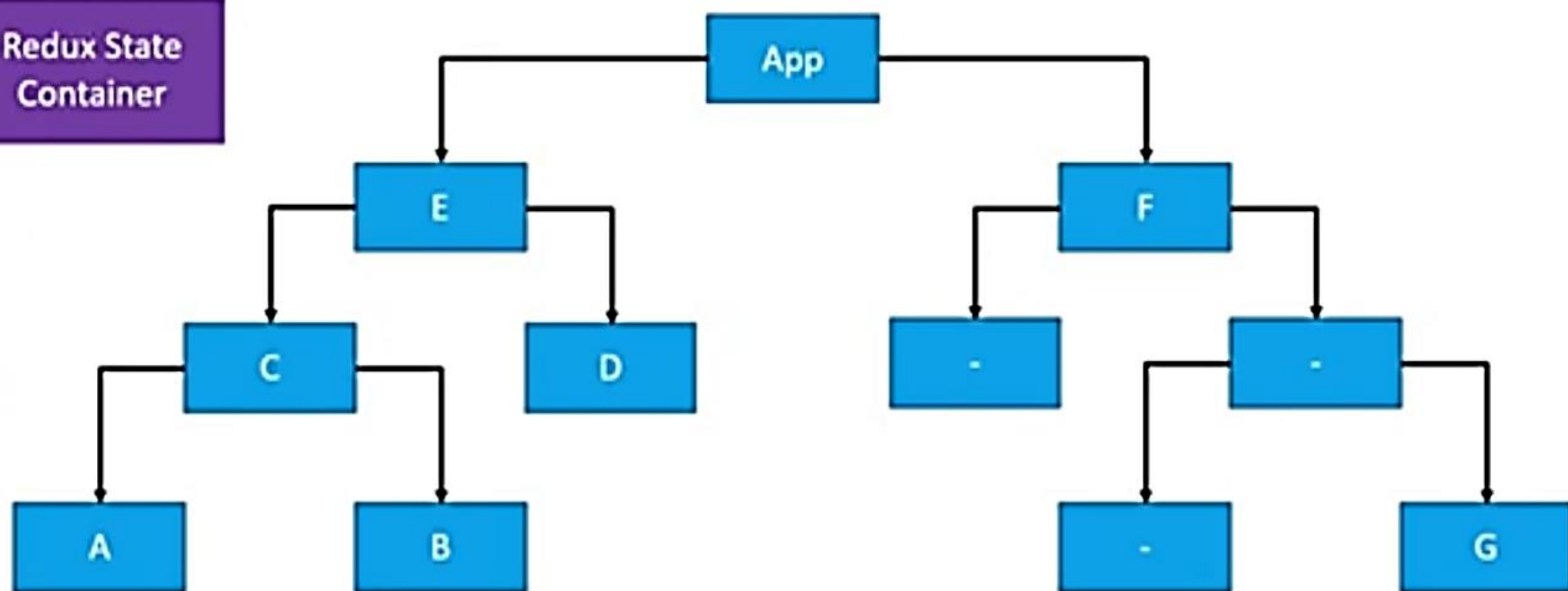
# State in a React App

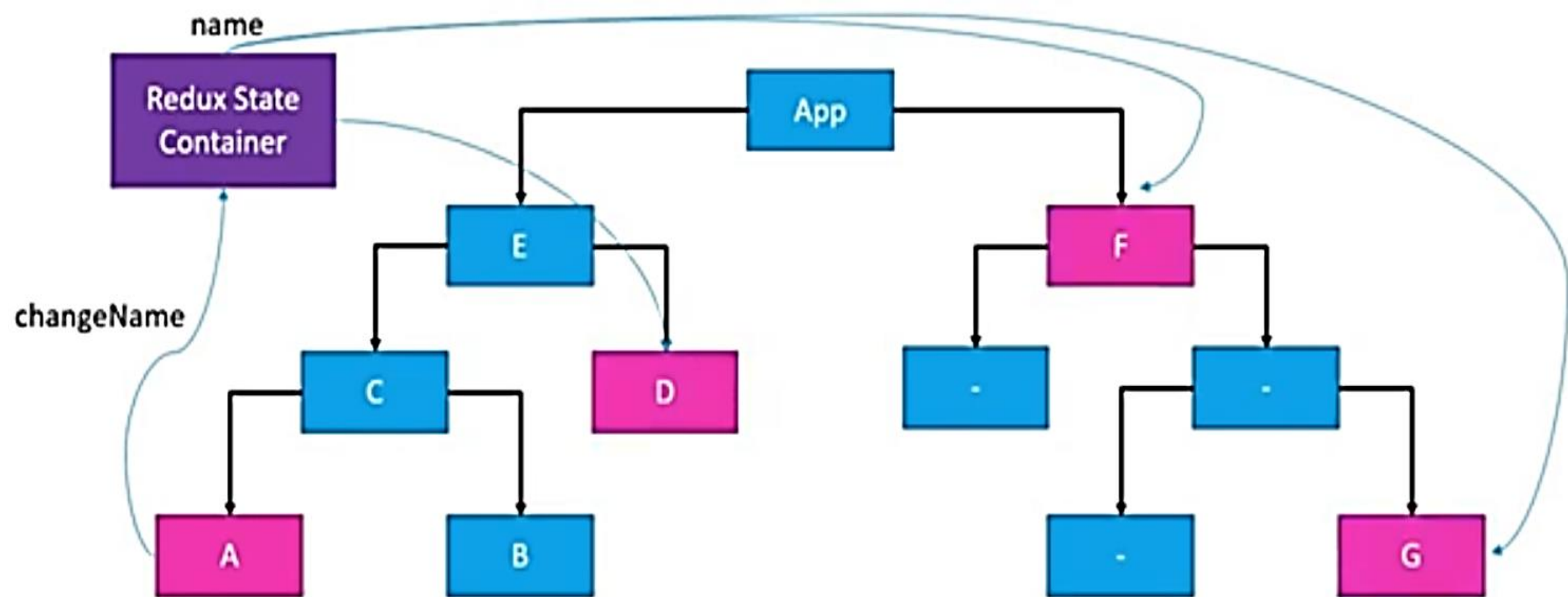




name

Redux State  
Container





# React-Redux



# Flux vs Redux

## Flux

- With Flux it is a convention to have multiple stores per application; each store is a singleton object.
- Flux has a single dispatcher and all actions have to pass through that dispatcher. It's a singleton object. A Flux application cannot have multiple dispatchers.
- Store handles all logic.
- Follows the unidirectional flow

## Redux

- In Redux, the convention is to have a single store per application, usually separated into data domains internally.
- Redux has no dispatcher entity. Instead, the store has the dispatching process baked in.
- Reducer handles all logic.
- Follows the unidirectional flow



# Redux's Three Core Concepts

## Cake Shop

### **Entities**

Shop:- Stores cakes on a shelf

Shopkeeper:- At the front of the store

Customer:- At the store entrance

### **Activities**

Customer:- By a cake

Shopkeeper:- Remove a cake from the shelf

-Receipt to keep track

Customer:- At the store entrance

# Three Core Concepts

Cake Shop Scenario	Redux	Purpose
Shop	Store	Holds the state of your application
Intention to Buy Cake	Action	Describes what happened
Shopkeeper	Reducer	Ties the store and actions together

- A **Store** that holds the state of your application.
- An **Action** that describes the changes in the state of the application.
- A **Reducer** which actually carries out the state transition depending on the action.

# Three Principles

## First Principle

*“ The state of your whole application is stored in an object tree within a single store.”*

Maintain our application state in a single object which would be managed by the Redux store.

## Cake Shop

Let's assume we are tracking the number of cakes on the shelf

```
{  
  numberOfCakes: 10  
}
```

# Three Principles

## Second Principle

*“The only way to change the state is to emit an action, an object describing what happened”*

To update the state of your app, you need to let Redux know about that with an action.

Not allowed to directly update the state object

## Cake Shop

Let the shopkeeper know about your action- BUY\_CAKE

{

type: BUY\_CAKE

}

# Three Principles

## Third Principle

*“To specify how the state tree is transformed by actions, you write pure reducers”*

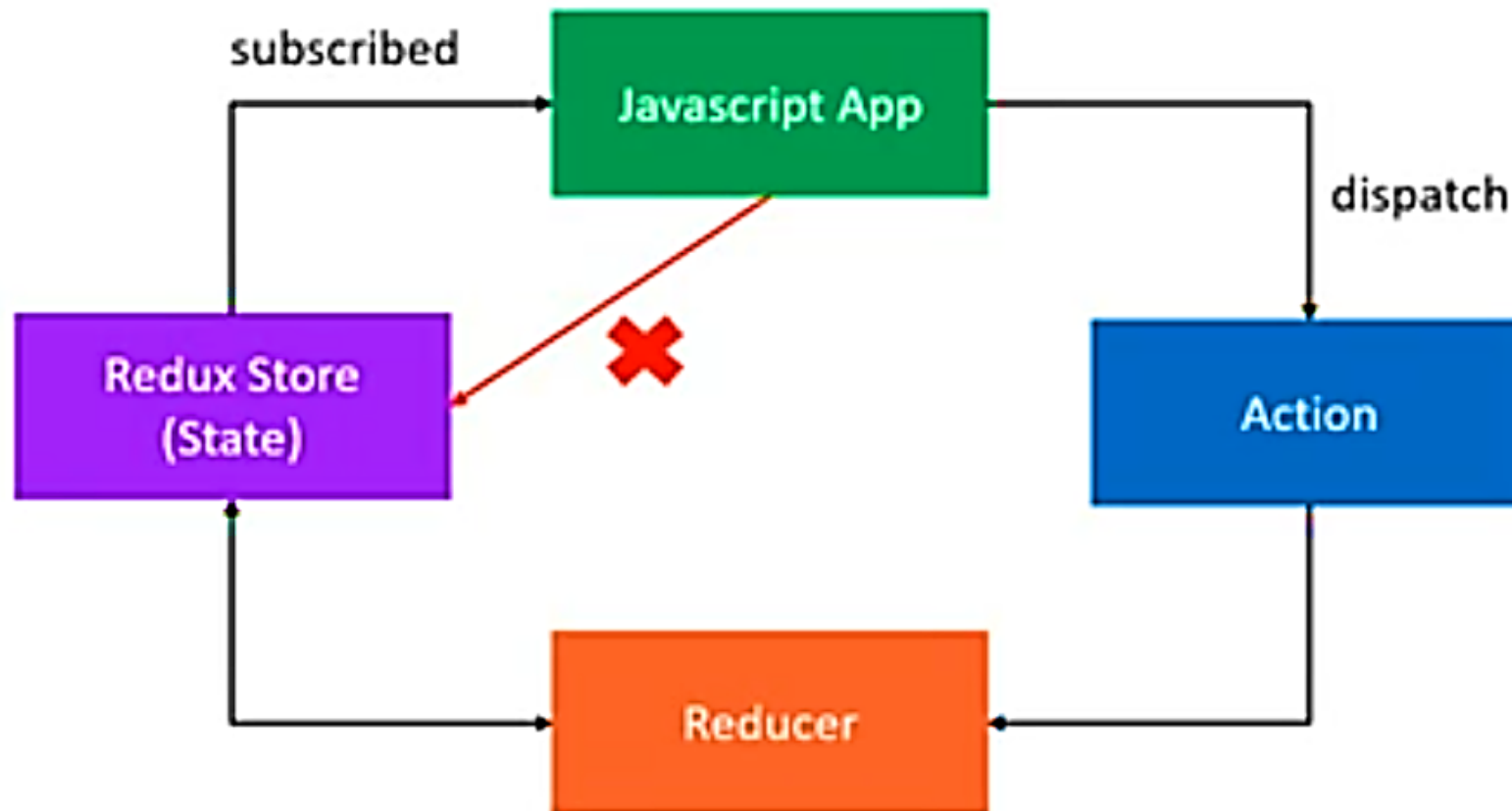
Reducer – (previousState, action) => newState

## Cake Shop

Reducer is the shopkeeper

```
const reducer = (state, action) => {  
  switch (action.type) {  
    case BUY_CAKE: return {  
      numOfCakes: state.numOfCakes - 1  
    }  
  }  
}
```

# Three Principles Overview



# Actions

- The only way your application can interact with the store.
- Carry some information from your app to Redux store.
- Plain JavaScript objects
- Have a 'type' property that indicates the type of action being performed.
- The 'type' property is typically defined as a string constants.

# Reducers

- Specify how the app's state changes in the response to actions sent to the store.
- Function that accepts state and action as arguments, and returns the next state of the application.
- `(previousState, action) => newState`



# Redux Store

- One store for the entire application
- Responsibilities:
  - Holds application state
  - Allows access to state via **getState()**
  - Allows state to be updated via **dispatch(action)**
  - Registers listeners via **subscribe(listener)**
  - Handles unregistering of listeners via the function returned by `subscribe(listener)`

# Cakes and Ice Creams

- Cake shop
  - Cakes stored on the shelf
  - Shopkeeper to handle BUY\_CAKE from the customer
- 
- Sell Ice Creams
  - Ice Creams stored in the freezer
  - New shopkeeper to handle BUY\_ICECREAM from customer.

# Actions

## **Synchronous Actions**

- As soon as an action was dispatched, the state was immediately updated.
- If we dispatch the BUY\_CAKE action, the numOfCakes was right away decremented by 1.
- Same with BUY\_ICECREAM action.

## **Asynchronous Actions**

- Asynchronous API calls to fetch data from an end point and use the data in our application.

# Our Application

- Fetches the list of users from an API end point and stores it in Redux store.
- State?
- Actions?
- Reducer?

# State

```
State{  
    loading: true,  
    data: [ ],  
    error: ''  
}
```

Loading: Displays a loading spinner in our component

Data: List of users

Error: Display error to the user.

# Action

FETCH\_USERS\_REQUEST – Fetch the list of users

FETCH\_USERS\_SUCCESS – Fetched successfully

FETCH\_USERS\_FAILURE – Error fetching the data

# Reducers

Case: FETCH\_USERS\_REQUEST

loading: true

Case: FETCH\_USERS\_SUCCESS

loading: false,

users: data (from API)

case FETCH\_USERS\_FAILURE:

loading: false,

error: data (from API)

# Testing React Components

- Testing may not matter much when you are working on a small app but it becomes very important in large scale apps which are worked on by many developers.
- Even a small change has the potential to break something in your app and if we have tests written for different features of our app, we can easily detect when a change breaks something in our app - **before we ship our code to production.**
- We will be looking into **React Testing Library**. Some functions with **Jest** testing framework.
- React Testing Library is used with a testing framework. Although React Testing Library is not dependent on any specific testing framework and can be used with any framework, the official docs of this library recommend to use the library with **Jest**.



```
npm install --save-dev @testing-library/react
```

- **Basics:**
  - **it** or **test**: describes the test itself. It takes as parameters the name of the test and a function that holds the tests.
  - **expect**: the condition that the test needs to pass. It will compare the received parameter to a matcher.
  - **a matcher**: a function that is applied to the expected condition.
  - **render**: the method used to render a given component.

# What is the React Testing Library?

- The React Testing Library is a very light-weight package created by **Kent C. Dodds**. It's a replacement for Enzyme and provides light utility functions on top of **react-dom** and **react-dom/test-utils**.
- The React Testing Library is a DOM testing library, which means that instead of dealing with instances of rendered React components, it handles DOM elements and how they behave in front of real users.

# React Routers

- React is a library for creating front end views. It has a big ecosystem of libraries that work with it. Also, we can use it to enhance existing apps.
- To build single-page apps, we have to have some way to map URLs to the React component to display.
- React Router is the perfect tool to link together the URL and your React app. React Router is the de-facto React routing library, and it's one of the most popular projects built on top of React.
- React at its core is a very simple library, and it does not dictate anything about routing.

- Routing in a Single Page Application is the way to introduce some features to navigating the app through links, which are expected in normal web applications:
  - 1.The browser should **change the URL** when you navigate to a different screen
  2. **Deep linking** should work: if you point the browser to a URL, the application should reconstruct the same view that was presented when the URL was generated.
  - 3.The **browser back (and forward) button** should work like expected.
- Routing links together your application navigation with the navigation features offered by the browser: the address bar and the navigation buttons.
- For Installation of React-Router we will use the following command:

**npm install react-router-dom**

# Router Components

- The 4 components you will interact the most when working with React Router are:
  - **BrowserRouter**, usually aliased as **Router**
  - **Link**
  - **Route**
  - **Switch**
- **BrowserRouter** wraps all your Route components.
- **Link** components are used to generate links to your routes
- **Route** components are responsible for showing - or hiding - the components they contain.
- The **Switch** component is used to render components only when the path will be matched. Otherwise, it returns to the not found component.

- **BrowserRouter:** BrowserRouter is the router implementation that uses the *HTML5 history API* to keep your UI up to date with the browser URL. It is BrowserRouter's responsibility to store all the components and its routes as an object.
- **Switch:** Switch components are used to render the default components once the app rendered, and it will switch between routes as needed.
- **Route:** The route is a statement that holds the specific path of the app along with the component's name and renders it once it matches the URL.
- **Link:** The link is similar to the HREF link, which allows you to redirect to the specific components based on the specified path.

# What does the `<Route>` render?

- Routes have three props that can be used to define what should be rendered when the route's path matches. Only one should be provided to a `<Route>` element.
- **component**: A React component. When a route with a component prop matches, the route will return a new element whose type is the provided React component (created using `React.createElement`).
- **render**: A function that returns a React element . It will be called when the path matches. This is similar to component, but is useful for inline rendering and passing extra props to the element.
- **children**: A function that returns a React element. Unlike the prior two props, this will always be rendered, regardless of whether the route's path matches the current location.

# Nested Routing & URL Parameters

- If the URL location matches the /courses path, then the Core\_Courses, Elective\_Courses, and MOOC\_Courses links are rendered via the Courses component.
- Going one step further, if the URL location matches /courses/Core\_Courses, /courses/Elective\_Courses, and /courses/MOOC\_Courses path, then This is technology, This are Core\_Courses, and This are Core\_Courses are rendered respectively.
- In the above example, there's a lot of repetition and hardcoding. The more the lines of code, the harder it becomes to change a route.



# Path and Match

- The **path** prop is used to identify the portion of the URL that the router should match. It uses the **Path-to-RegExp** library to turn a path string into a regular expression. It will then be matched against the current location.
- If the router's path and the location are successfully matched, an object is created which is called a **match object**. The match object contains more information about the URL and the path.

- This information is accessible through its properties, listed below:
  - **match.url:** a string that returns the matched portion of the URL. This is particularly useful for building nested **<Link>** components.
  - **match.path:** a string that returns the route's path string — that is, **<Route path="">**. We'll be using this to build nested **<Route>** components.
  - **match.isExact:** a Boolean that returns true if the match was exact (without any trailing characters).
  - **match.params:** an object containing key/value pairs from the URL parsed by the Path-to-RegExp package.
- We used the **match.params** which provides a key/value object of the URL location. **:Courses** is the URL param. Therefore, **match.params.course** will provide the value of the correct URL location.

# React-Spring

- React Spring is a spring-physics based animation library that powers most UI related animation in React.
- It is a bridge on the two existing React animation libraries; React Motion and Animated.
- Given the performance considerations of animation libraries, React Spring is the best of both worlds. It inherits animated powerful interpolations and performance while maintaining react-motion's ease of use.

- There are five major hooks available in React Spring:
  - **useSpring**: a single spring, moves data from a -> b
  - **useSprings**: multiple springs, for lists, where each spring moves data from a -> b
  - **useTrail**: multiple springs with a single dataset, one spring follows or trails behind the other
  - **useTransition**: for mount/unmount transitions (lists where items are added/removed/updated)
  - **useChain**: to queue or chain multiple animations together

# Explanation of the Translate Code

- **useSpring** accepts an object with a **from** and **to** property. There's an optional **config** property which allows you to fine tune the animation settings.
- **from** indicates the values of properties from which the animation starts. In our case, we are saying the value of the **transform** property shall start **from translate(0px)**.
- **to** indicates the values that properties should animate towards. In the example, we are animating towards **translate(120px)**.