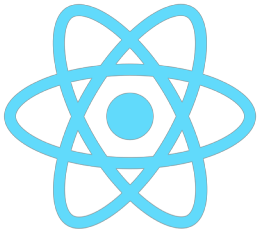


ReactJS

Compiled by
Lakshman M N
Technology Mentor / Evangelist
mnl@lakshman.biz

React!..?

- It is a JavaScript library for creating reactive and fast JavaScript driven web applications.
  - JavaScript running in the browser allows the creation of applications that run fast and feel like mobile applications.
- React is one of the most popular JavaScript library presently as indicated by indeed.com with a listing of more than 45k jobs.
- React was created by Jordan Walke, a software engineer at Facebook.
  - It was open sourced in 2013
- What is react all about?
- Why should I care?

What is React?

- React is a JavaScript library for building user interfaces, that is maintained by Facebook as an Open Source project under the MIT license
- "React is a JavaScript library for building User Interfaces"
  - React applications run in the browser.
  - This facilitates speedier responses to user actions.
  - It is maintained by Facebook and community
- React is a library for building composable user interfaces.
- It has to be emphasised that React is **NOT** a framework; it is a library.
- User Interfaces as in web pages can be split into components.
- Many developers tend to use react as the V in MVC.
- React abstracts the DOM providing a simpler programming model and better performance.

Why React?

- React is a JavaScript library used for building reusable UI components.
- UI state is cumbersome to handle using plain JavaScript.
  - DOM elements need to be manually targeted in order to change the structure of the HTML.
  - Complex web applications that need elements to be added/removed tend to become tedious using just JavaScript/jQuery.
- React helps developers focus on business logic
  - Creators have developed react using patterns and best practices thus ensuring optimal code base for us to start with.
- React enjoys a huge ecosystem and vibrant community support.

React – ES6

- React applications typically are built with the latest version of JavaScript.
- The newer features allow for the creation of cleaner and robust React applications.
- React itself employs a lot of these newer JavaScript features.
- JavaScript as a language is evolving fairly rapidly.

Classes

- Classes are a new feature in ES6, used to describe the blueprint of an object
  - They perceive the transformation of ECMAScript's prototypal inheritance model to a more traditional class-based language.
- ES6 classes offer a much nicer, cleaner and clearer syntax to create objects and deal with inheritance.
- The class syntax is not introducing a new object-oriented inheritance model to JavaScript.

ES 2015

- ES5 has been around since 2009.
  - It is supported by most of the popular browsers.
- In June 2015 a new specification of the JavaScript standard was approved that contains a lot of new features.
- It is called ECMAScript 2015 or also called ES6 as it is the 6th edition of the standard.
  - ECMAScript is the official name of the JavaScript language.
- Existing browsers don't support most of the features of ES6 yet.

ES 2015...

- Feature support across browsers varies widely.
- Are we expected to wait a few years and commence using ES6 after browsers start offering support?
  - http://kangax.github.io/compat-table/es6/
- Fortunately not!
- There are tools that can convert ES6 code into ES5 code.
- We write code using the new useful features of ES6 and generate ES5 code that will work in most of the current browsers.

ES6 / ES 2015

ES6 brings a lot of new features, some of which include:

- Classes
- Arrow Functions
- Template Strings
- Inheritance
- Constants and Block Scoped Variables
- Modules

Classes...

```
class Shape {
  constructor(type){
     this.type = type;
  }
  getType(){
      return this.type;
  }
}
```

- Use the **`class`** keyword to declare a class.
- **`constructor`** is a special method for creating and initializing an object.
  - There can only be one special method with the name **`constructor`**

ES6-Demo01.htm

Classes...

- The **`static`** keyword defines a static method for a class.
- Static methods are called without instantiating their class and are not callable when the class is instantiated.
- Static methods are often used to create utility functions for an application.

```
class Shape {
  constructor(type){
     this.type = type;
  }
  static getClassName(){
    const name = 'Shape';
        return name;
  }
}

console.log(Shape.getClassName());
```

ES6-Demo02.htm

Subclassing

- The **`extends`** keyword is used in class declarations to create a class as a child of another class.
- The **`super`** keyword is used to call functions on an object's parent.

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

  speak() {
    console.log(this.name + ' makes a noise.');
  }
}

class Dog extends Animal {
  speak() {
    super.speak();
    console.log(this.name + ' barks.');
  }
  }

var d = new Dog('Mitzie');
d.speak();
```

ES6-Demo03.htm

this revisited

- In JavaScript '**this**' keyword is used to refer to the instance of the class.

```
var shape = {
  name: 'square',
  say: function(){
       console.log('This is say(): ' + this.name);

       setTimeout(function(){
              console.log('Inside setTimeout(): '
              + this.name);
       }, 2000);
  }
};

shape.say();
```

- The **this.name** is empty when accessed within **setTimeout**.

ES6-Demo04.htm
ES6-Demo05.htm

Arrow functions

- ES6 offers new feature for dealing with **this**, "**arrow functions**" **=>**
  - Also known as *fat arrow*
- Some of the motivators for a *fat arrow* are:
  - One does not need to specify **function**
  - It lexically captures the meaning of **this**
- The fat arrow notation can be used to define anonymous functions in a simpler way.
- Helps provide the context to **this**

Arrow functions

```
var shape = {
      name: 'square',
      say: function(){
            console.log('This is say(): ' + this.name);

            setTimeout(() => {
                  console.log('Inside setTimeout(): ' + this.name);
            }, 2000);
      }
};

shape.say();
```

ES6-Demo06.htm

Arrow functions...

- Arrow functions do not set a local copy of **`this`**, **`arguments`** etc.
- When **`this`** is used in an arrow function, JavaScript uses the **`this`** from the outer scope.
- If **`this`** *should* be the calling context, do not use the arrow function.

`let`

- **`var`** variables in JavaScript are *function scoped*.
- This is different from many other languages(Java, C#) where variables are *block scoped*.
- In ES5 JavaScript and earlier, **`var`** variables are scoped to the function and they can "*see*" outside their functions into the outer context.

```
var foo = 123;
if(true){
   var foo = 456;
}
console.log(foo);  //456
```

ES6-Demo07.htm

`let`…

- ES6 introduces the **`let`** keyword to allow defining variables with true *block scope*.
- Use of the **`let`** instead of **`var`** gives a true unique element disconnected from what is defined outside the scope.

```
let foo = 123;
if(true){
 let foo = 456;
}
console.log(foo);  //123
```

ES6-Demo08.htm

`let`…

- Functions create a new variable scope in JavaScript as expected.

```
var num = 123;
function numbers(){
   var num = 456;
}

numbers();
console.log(num);   //123
```

ES6-Demo09.htm

`let`…

- Usage of **`let`** helps reduce errors in loops.
- **`let`** is extremely useful to have for the vast majority of the code.
- It helps decrease the chance of a programming oversight.

```
var index = 0;
var myArray = [1,2,3];
for(let index = 0; index < myArray.length;
        index++){
   console.log(myArray[index]);
}
console.log(index);   //0
```

ES6-Demo10.htm

const

- **const** is a welcome addition in ES6.
- It allows immutable variables.
- To use **const**, replace **var** with **const**

```
const num = 123;
```

- **const** is a good practice for both readability and maintainability.
- **const** declarations must be initialized

```
const foo;  //ERROR
```

const

- A **const** is block scoped like the **let**
- A **const** works with object literals as well.

```
const foo = { bar : 123 };
foo = { bar : 456 };  //ERROR
```

- **const** allows sub properties of objects to be mutated

```
const foo = { bar : 123 };
foo.bar = 456;  //allowed
console.log(foo);  // { bar : 456 }
```

ES6-Demo11.htm

Template Strings

- In traditional JavaScript, text that is enclosed within matching “ or ‘ marks is considered a string.
- Text within double or single quotes can only be on one line.
- There was no way to insert data into these strings.
- If there was a need it would have required concatenation that looked complex and not so elegant.
- ES6 introduces a new type of string literal that is marked with back ticks (`)

Template Strings

- The motivators for Template strings include
  - Multiline Strings
  - String Interpolation
- Multiline Strings

```
var desc = 'Do not give up \
\n Do not bow down';
```

  - with Template Strings

```
var desc = `Do not give up
Do not bow down`;
```

Template Strings...

- String Interpolation

```
var lines = 'Do not give up';
var html = '<div>' + lines + '</div>';
```

  - with Template Strings

```
var lines = 'Do not give up';
var html = `<div>${lines}</div>`;
```

- Any placeholder inside the interpolation **${ }** is treated as a JavaScript expression and evaluated.

ES6-Demo12.htm

Spread and Rest Operators

- These new operators are represented by **...**
- ***Spread*** operator is used to split up array elements OR object properties

```
let myData = ['A', 'B', 'C'];
let myNewData = [...myData, 'D']
console.log(myNewData);  //   ['A', 'B', 'C', 'D']

let user = { name : 'Lakshman'}
let userData = {
  ...user,
    email : 'laks@acme.org'
}
console.log(userData);  //    [object Object] {email:"laks@acme.org",
                                    name: "Lakshman"}
```

ES6-Demo14.htm

Spread and Rest Operators…

- ***Rest*** operator is used to merge a list of function arguments into an array.
  - Array methods can be applied to the arguments list.

```
function getData(...args){
  return(args.filter(el => el > 3).sort())
}
console.log(getData(3,1,4,8,6))    //   [4, 6, 8]
```

ES6-Demo15.htm

Destructuring

- *Destructuring* allows the extraction of array elements or object properties and store them in variables.
  - *Spread* operator takes out all array elements or all properties and distributes them in a new array or object.
- *Destructuring* allows pulling out single array elements or properties and store them in variables.

```
let myChars = ['A', 'B', 'C'];
[char1, char2] = myChars;
console.log(char1, char2);   //  'A' 'B'
[char1, , char3] = myChars;
console.log(char1, char3);   //  'A' 'C'
```

ES6-Demo16.htm

Modules

- JavaScript has always had problem with namespaces.
  - Variables and functions can end up in the global namespace if not cautious.
- JavaScript lacks built-in features specific to code organization.
- Modules help resolve these issues.

Modules in ES6

- In ES6 each module is defined in its own file.
- The functions and variables defined in a module are not visible outside unless explicitly exported.
- A module can be coded in a way such that only those values that need to be accessed by other parts of the application need be exported.
- Modules in ES6 are declarative in nature.
- To export variables from a module use **`export`**.
- To consume the exported variables in a different module use **`import`**.

Working with Modules

```
function getRandom(){
        return Math.random();
}

function add(n1, n2){
        return(n1+n2);
}

export {getRandom, add}
```

Utils.js

- The **export** keyword is used to export the two functions.
- The exported functions can be renamed while exporting

```
export {getRandom as random, add as sum}
```

Utils.js

Working with Modules...

```
import { getRandom, add } from 'Utils';

 console.log(getRandom());
 console.log(add(1,2));
```

useUtils.js

•This imports the exported values from the module 'Utils'.

•A single value can be imported as well

```
import { add } from 'utils';
```

useUtils.js

React - Characteristics

- React apps are built with the latest version of ES.
- React makes use of reusable components.
  - A component is a function/class that returns a section of the interface.
- React is declarative
  - React allows to describe what the application interface should look like
- Unidirectional data flow
  - React applications are built as a combination of parent and child components.
    - Data always flows from parent to child and never in the other direction.
- Powerful type-checking using PropTypes

React – The Virtual DOM

- DOM manipulation is the heart of most modern and interactive web applications.
- DOM manipulation is slow, moreover many JavaScript libraries and frameworks update the DOM more than needed.
- In React for every DOM object there is a corresponding Virtual DOM object.
- A Virtual DOM object is an in-memory representation of the real DOM object.
  - It has the same properties as the real DOM object.
- It lacks the power to directly change what is on the screen.

Virtual DOM…

- Manipulating the DOM is slow whereas manipulating the virtual DOM is fast as nothing gets drawn on the screen.
- At any given time React maintains two virtual DOM, one that is updated and another with the previous state.
- Once the virtual DOM is updated React compares it with the previous version.
- "**Diffing**" is the process by which React figures the virtual DOM objects that have changed.
- React updates only the changed objects in the real DOM.
- Changes on the real DOM cause the screen to change.

Setting up the Environment - Workflow

- Why do we need to setup a workflow?
  - Code optimization
    - This increases the performance of the application.
  - Use ES6 features
    - This is the de-facto standard for react that makes the code easier to read, leaner and faster.
    - Though code is written in ES6 the shipped code needs to run in multiple browsers.
    - The workflow needs to compile ES6 features into browser comprehend able ES.
  - Increase Productivity
    - CSS auto-prefixing to achieve broad browser support.
- The build workflow needs to allow developers to write modern code that facilitates all of the above.

Setting up the Environment - Workflow

- A dependency management tool - ***npm***
  - npm (Node Package Manager) is a command line utility that ships with NodeJS (downloadable from http://www.nodejs.org)
  - Third party libraries and packages are dependencies
  - Compiler to convert ES6 to ES5 is a dependency
  - Build tools are dependencies
- A bundler - ***webpack***
  - Code is authored in a modular manner and split over multiple files.
  - These files need to be bundled when shipped as browsers do not support segmented files.
  - Webpack allows setup of build steps before bundling
    - Babel + presets can be configured in webpack to be a part of the bundling process
- A development Web server

Setting up a React project

- There is a tool called “Create React App” that helps create a project.

  `https://github.com/facebook/create-react-app`
- This tool is maintained by Facebook and has a community around it.
- It is the officially recommended tool for creating *React* projects.
- Considering that we have node installed from **`http://nodejs.org`**

  `npx create-react-app <projectname>`
  - **`create-react-app`** is a node package.
  - **`npx`** is a **`npm`** package runner

Setting up a React project…

- A folder with the name specified will be created in the current path and the project files downloaded into it.

```
cd <projectname>
npm start
```

  - This starts the development server, loads the application that can be accessed at **`http://localhost:3000`**
  - The process started with **`npm start`** needs to be kept running since the server can detect changes made to the code and reload the page in the browser.

Folder structure

- Open the folder in VS Code (IDE).
- The general dependencies of the project are defined in **`package.json`**
  - **`react`** is the library to help with logic and components
  - **`react-dom`** helps render the components into the DOM
  - **`react-scripts`** is a package that helps set up the build workflow, development server and required ES6 support.
- The **`node_modules`** folder contains all the dependencies.
- The **`public`** folder is the root folder that gets served by the web server.
  - The **`index.html`** is the single html page residing in this folder.
  - There will be no more html pages in this folder.
- Script files are placed in the **`src`** folder.

File structure

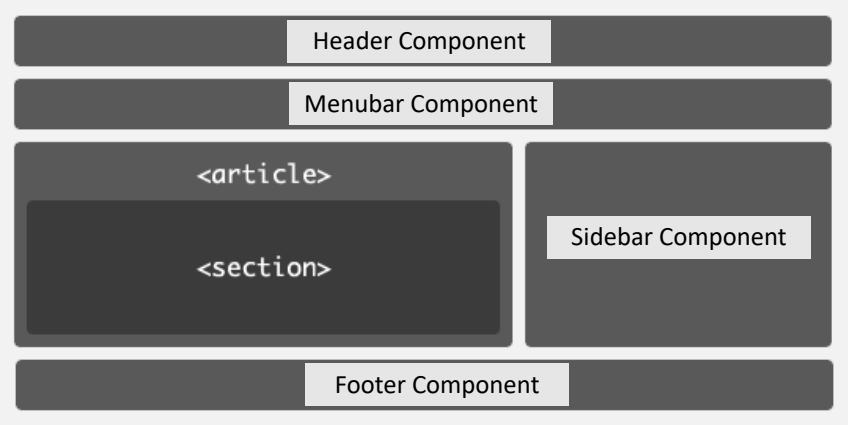
- The script files will get injected into **`index.htm`** by the build workflow.
  - The **`<div id="root"></div>`** is the mount point for the react application.
- **`manifest.json`** is added by the *create-react-app* to provide a progressive web application out of the box.
- **`index.js`** gets access to the **`root`** element in the DOM and renders the react application with the **`render()`**
- **`App.js`** contains the react component created by *create-react-app*
- **`registerServiceWorker.js`** created by *create-react-app* is used to help pre-cache files and is related to progressive web application.

Components

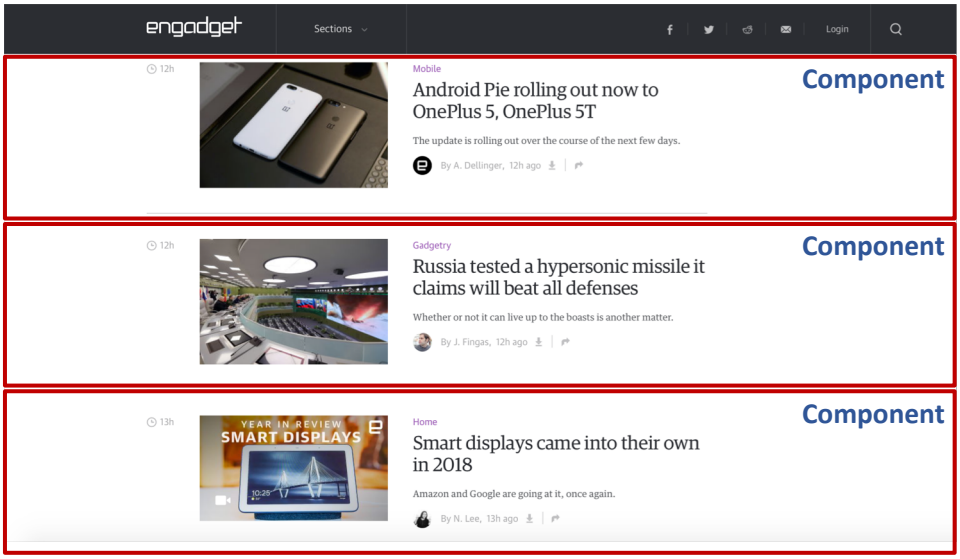
- A Component can be considered as a custom HTML element.
- React therefore addresses the problem of having to build sophisticated user interfaces with HTML and JavaScript.
- *A component is a simple, manageable, maintainable and reusable piece of code*.
- The component can be plugged into web pages desiring a specific functionality.
  - Enhancements to the component lead to enhancement of the entire page.

Components?

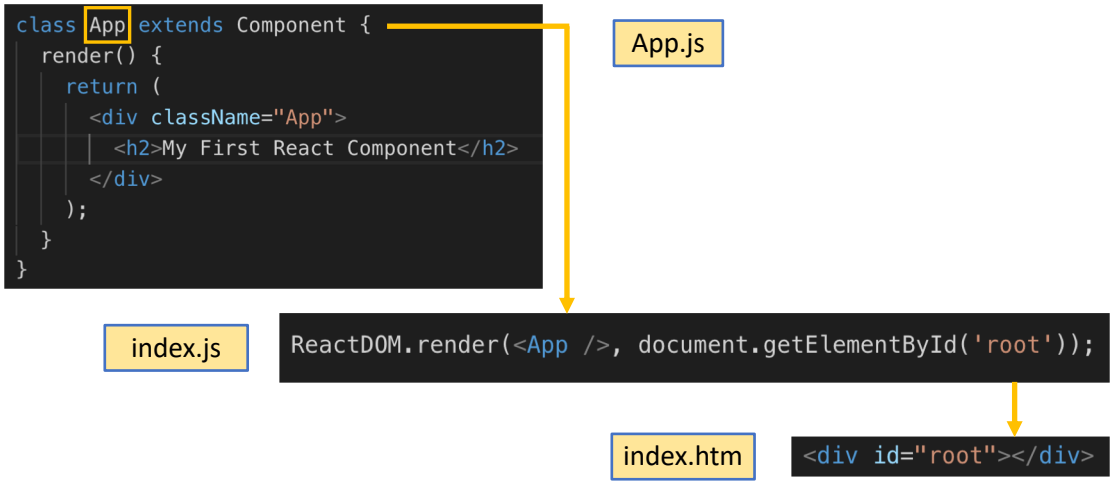
- Every web page can be split into components.
- Components can be reused and hence help simplify the creation of web pages



Components…



React Components



React Components

- In a react application generally, one root component is rendered.
- All other components needed by the application are nested in there.
- A react component is extended from the **`Component`** class that is imported from the **`react`** library.
- The component has a method called **`render()`**
  - The **`render()`** method returns "HTML like" content.
  - This method is called by ***react*** to emit content into the DOM and thereby render content to the view.

JSX?

- JavaScript eXtension (JSX) is used in React to describe the UI.
- JSX is a preprocessor step that adds XML syntax to JavaScript.
  - Like in XML, JSX tags have a tag name, attributes and children.
  - If a value is enclosed in quotes, it is regarded as a string.

```
const appElement = <h2 className='App'>React Component</h2>;
```

- JSX makes React a lot more elegant though React can be used without JSX.
- JSX allows HTML to be put into JavaScript.

JSX-Demo01.js

JSX

- React embraces "*separation of concerns*" with loosely coupled units called components that contain both markup and logic.
- React uses JSX for templating instead of regular JavaScript.
- JSX syntax is intended to be used by preprocessors (like Babel) to transform HTML-like text into standard JavaScript objects that can be parsed by a JavaScript engine.
- Familiarity of HTML helps create templates quicker.
- JSX is faster since optimization is performed when compiling code into JavaScript.
- JSX produces React "Nodes"

React Nodes

- A React Node is defined as a light, stateless, immutable and virtual representation node.
- React nodes are not real DOM nodes themselves, but a representation of a potential DOM node.
- The representation is considered the virtual DOM.
- React is used to define a virtual DOM using React nodes.
- React nodes can be created using JSX or JavaScript.

Creating React Nodes

- Creating React nodes using JavaScript is accomplished by **`React.createElement()`**
- This method is used to create a virtual DOM representation of an element node.
- To create the virtual DOM the React element node should be rendered to a real DOM.
- This is achieved using the **`ReactDOM.render()`** method.

JSX...

- JSX is simply converting XML-like markup into JavaScript.

```
<h2>React Component</h2>
```

- gets compiled to

```
React.createElement('h2',null,'React Component')
```

  - The first argument is the element that is to be rendered
  - The second argument is a JavaScript object and is optional
  - The third argument represents the children (the content nested in the element mentioned as the first argument)

JSX-Demo02.js

JSX-Demo03.js

JSX...

```
<div className="App"><p>React Component</p></div>
```

- transforms to

```
React.createElement('div',{className:'App'},
                React.createElement('p', null, 'React
                                        Component'))
```

- React uses **`className`** instead of the traditional DOM **`class`**.
  - **`class`** is a keyword in ES6

JSX-Demo04.js

Expressions

- The **{ }** brackets in JSX indicate that the content is JavaScript.
- It is eventually parsed by the JavaScript engine.
- **{ }** can be used anywhere among the JSX expressions as long as the result is a valid JavaScript.

```
const contentNode = 'React Component';

<h2>{ contentNode }</h2>
```

JSX-Demo05.js

Conditional statements

- `if-else` statements do not work in JSX.
- `if` statements can be used outside JSX to determine the content.
- Ternary operator can be employed in JSX

```
<select name='selCity'>
  <option>Delhi</option>
  <option>Mumbai</option>
  { currCity==='Bangalore' ? null :
                    <option>Bangalore</option>
  }
</select>
```

JSX-Demo06.js

IIFE

- Immediately invoked function expression can be used in JSX for logic.

```
{(() => {
     if(localStorage.getItem('user'))
       this.userName = localStorage.getItem('user')
     else
       this.userName = 'Guest';
    return <p>{this.userName}</p>
   }
)()}
```

JSX-Demo07.js

Creating Components

- A React application can be depicted as a component tree - having one root component ("App") and a number of nested child components.
- Components can be created in two possible ways:
  - **Functional components**
    - Also known as *presentational* or *stateless* components
  - **Class-based components**
    - Also known as *containers* or *stateful* components

Creating a Functional Component

- A React component is basically any part of a UI that can contain React nodes.

```
import React from 'react';

const user = () => {
    const userName = 'Lakshman'
    return <p>{userName}</p>
}

export default user
```

User.js
Component-Demo01.js

Creating a Class based Component

- A class based component is a JavaScript class.
  - Class in ES6 can have constructor, methods etc.

```
class userContainer extends React.Component
```

- It extends the **`React.Component`** and requires a method **`render()`**
- React expects the class to return a react element from the **`render()`**

```
render(){
    return(
        <div>
            <p>{this.userName}</p>
        </div>
        );
    }
```

UserClass.js
UserClassApp.js

Component Props

- Component ***Props*** (short for properties) function like HTML attributes.
- Props provide configuration values for the component.
  - Props is **readonly**

```
<User userName='Lakshman' />
<User userName='Sam' />
```

App.js

```
const user = (props) => {
    return <p>{props.userName}</p>
}
```

User.js

User1.js
Component-Demo02.js

Props – Children property

- React provides access to a special prop called children.
- Children is a reserved term referring to the all the content between the component tag.

```
<User userName = 'Lakshman'>Lakshman M N</User>
<User userName = 'Sam' />
```

App.js

User.js

```
<p>User Name : {props.userName}</p>
<p>Name : {props.children}</p>
```

User2.js
Component-Demo03.js

Props Validation

- With the developmental growth of an application a number of bugs can be screened using type checking.
- The `prop-types` library can be used to run type checking on the props for a component.

```
npm install --save prop-types
```

- `prop-types` can be used to document the intended types of properties passed to components.
- React will check props passed to components against those definitions and warn if they do not match.

Props Validation

```
import PropTypes from 'prop-types';
```

- `PropTypes` exports a range of validators that can be used to ensure valid data.

```
course.propTypes = {
    name : PropTypes.string,
    duration : PropTypes.number
    }
```

- If default props are set for the React component, the values are first resolved before type-checking against `propTypes`.
  - Default values are also subject to prop type definitions.
- `propTypes` type-checking only happens in development mode.

Course.js
CourseApp.js

Props Validation

- Some of the prominent validators available include:
  - `PropTypes.bool`
  - `PropTypes.number`
  - `PropTypes.string`
  - `PropTypes.func`
  - `PropTypes.array`
  - `PropTypes.object`
  - `PropTypes.string.isRequired` – `isRequired` can be chained to any prop validator

Props Validation

- Multiple Types
  - `PropTypes.oneOf` – the prop is limited to a specific set of values

```
prodCondition : PropType.oneOf(['New', 'Used', 'NA'])
```

  - `PropTypes.oneOfType` – the prop should be one of a specified set of types

```
courseDuration : PropType.oneOfType([
                   PropType.string,
                   PropType.number
                   ])
```

Props Validation

- Custom validators
  - `prop-types` allow custom validation functions for type-checking
  - The validation function accepts three arguments
    - **props** – an object containing all the props passed to the component
    - **propName** – the name of the prop to be validated
    - **componentName** – the name of the component
  - It should return an `Error` object if the validation fails.

Course1.js
Course1App.js

Component **State**

- Most components should take in *props* and render.
- Components also offer ***state*** and this is used to store information about the component that can change over time.
- A ***state*** change implicitly re-renders the component.
- The ***state*** object should only contain minimal amount of data needed for the UI.
- ***state*** is available only in components that extend the **`Component`** class.

Component-Demo04.js

Props and State

- **Props** and **State** are core concepts in React.
- Both are plain JavaScript objects.
- Both can have default values.
- Both should be accessed using **`this`** (`this.props` or `this.state`)
  - Both are **readonly** when using `this`
- Changes to **Props** and/or **State** trigger React to re-render components and potentially update the DOM in the browser.

Props and State

- **Props**
  - Props are passed into the component from above (generally parent component)
  - Props are intended as configuration values passed into the component (like arguments passed to a function)
  - Props are immutable to the component receiving them.
- **State**
  - State is a serializable representation of data (JS object) generally associated with the UI.
  - Only class based components can define and use state.
  - State should always start with a default value.
  - State can only be mutated by the component that contains it.
  - *State should be avoided if possible*

Handling Events

- Handling events with React elements is very similar to handling events with DOM elements.
- React events are named using *camelCase* notation.
- In JSX a function is passed as the event handler as opposed to a string.
- The property name for an event starts with 'on'

```
<button onClick={this.clickHandler}>Click Me</button>
```

Notice the absence of parentheses after the method name

Component-Demo05.js

Handling Events

- React creates a **`SyntheticEvent`** for each event that contains the details for the event.
  - **`SyntheticEvent`** is a cross-browser wrapper around the browser's native event.
  - The **`nativeEvent`** attribute of the **`SyntheticEvent`** can be used to access the underlying browser event.
- The **`SyntheticEvent`** instance is passed into the event handlers/callback functions for the event.
- React normalizes events so that they behave consistently across browsers.
- Events in React are triggered on the *bubbling* phase.
  - To trigger the event on the capture phase suffix the term "`Capture`" to the event name (`onClick` becomes `onClickCapture`)

Component-Demo06.js

React Events

EventType	Events
Keyboard	onKeyDown, onKeyPress, onKeyUp
Focus	onChange, onInput, onSubmit
Form	onFocus, onBlur
Mouse	onClick, onContextMenu, onDoubleClick, onMouseDown, onMouseOver, onMouseOut, onMouseUp
Clipboard	onCopy, onCut, onPaste
Selection	onSelect
Image	onload
Transition	onTransitionEnd

Working with **State**

- "State" is a reserved term and is set in class-based components.
- State facilitates changes from within the component.
- Working with a component State involves setting a component's default state, accessing the current state and updating it.
- A component has its state updated using **`this.setState()`**
  - **`setState()`** is provided by the Component class
  - **`setState()`** accepts an object as an argument and merges it with the existing state.
- A state change internally deals with calling re-renders.
  - All child components are also re-rendered as appropriate.

Component-Demo07.js

Component interaction – Sharing Events

- Stateless function components can be provided methods also as *props*.
- This allows the invocation of method that can change the *state* in another component that does not have direct access to the *state*.

```
<User myEvent = {this.clickHandler} userName =
{this.state.Users[0].name}>Demi God</User>

<p onClick={props.myEvent}>User Name : {props.userName}</p>
```

User3.js
Component-Demo08.js

User4.js
Component-Demo09.js

Styling Components

Styling Components

- No user-facing web application is complete without styles.
- There are different approaches to style a component:
  - CSS Stylesheet
    - Import the .css file and use the `className` property on elements to attach styles
  - Inline styles
    - Styles are created inline as JS objects and assigned to elements using the `style` property
  - CSS-in-JS
    - Similar to inline styles except that the style objects are in a separate JS module
  - CSS Modules

CSS Stylesheet

CSSDemo.css

```
.CSSDemo {
	margin: 40px;
	border: 5px dashed blue;
}

.CSSDemo_content {
	font-size: 15px;
	text-align: center;
}
```

CSSDemo.js

```
<div className="CSSDemo">
	<p className="CSSDemo_content">Content</p>
</div>
```

CSSDemo.css
CSSDemo.js

Inline styling

```
const CSSDemoStyle = {
     margin: '40px',
     border: '5px dashed blue'
};

<div style={CSSDemoStyle}>
    Content
</div>
```

CSSDemo1.js

CSSDemo1.js

CSS-in-JS

- This is a pattern where the CSS is composed using JavaScript instead of being defined in CSS files.
- This functionality is not a part of React but provided by third party libraries.

```
export default {
CSSDemo_CSSInJS: {
margin: '40px',
border: '5px dashed blue'}
}
```

CSSinJS.js

```
import CSSJS from './CSSinJS';

<div style={CSSJS.CSSDemo_CSSInJS}>
<p style={CSSJS.CSSDemo_Content_CSSInJS}>CSS in JS</p>
</div>
```

CSSDemo2.js

CSSinJS.js
CSSDemo2.js

CSS Modules

- A CSS Module is a CSS file in which all class names are scoped locally by default.

CSSModule.css

```
:local(.CSSModule) {
margin: 40px;
border: 5px dashed blue;
}
```

CSSDemo3.js

```
import styles from './CSSModule.css';

<div className={styles.CSSModule}>Content</div>
```

CSSModule.css
CSSDemo3.js

React and Bootstrap

- React being a view library, does not have any built-in mechanism to help create designs that are responsive and intuitive.
- A front end design framework like Bootstrap can help alleviate these concerns.
- Integrating Bootstrap with React allows developers to use Bootstrap's grid system and various other components.

Adding Bootstrap for React

- Bootstrap can be added to the React application in three common ways:
  - Using the Bootstrap CDN
    - No installs required
  - Bootstrap as a dependency
    - A common option to add Bootstrap to the React application
    - Bootstrap, jquery and popper.js need to be installed using npm
  - React Bootstrap Package
    - A package that has rebuilt Bootstrap components to work as React components.

React Bootstrap Package

- There are a few libraries that create a React specific implementation of Bootstrap.
- `reactstrap` is a library that gives the ability to use Bootstrap components in React.
  - The module includes components for typography, icons, buttons etc.
- Install `bootstrap` and `reactstrap`

```
npm install --save bootstrap
npm install --save reactstrap
```

- Import Bootstrap CSS in `src/index.js`

```
import 'bootstrap/dist/css/bootstrap.min.css';
```

BootStrapDemo.js
BootStrapDemoApp.js

Forms

Forms

- Forms are integral to any modern application.
- They serve as a basic medium for users to interact with the application.
- Developers rely on forms for various capabilities such as securely logging in the user, building a cart, searching a product list etc.
- React does not provide comprehensive form validation support out of the box.
- Unlike other DOM elements, HTML form elements work differently in react.

Forms in React

- There are two types of form input in react.
- **Uncontrolled input**
  - Like traditional HTML form inputs, they remember what is typed.
  - `ref` is used to get the form values.
- **Controlled input**
  - This is when the react component that renders a form also controls what happens to the form on subsequent user input.
  - As the form value changes, the component that renders the form saves the value in its state.

Uncontrolled components

- The form data is stored in the DOM and not within the component.
- Elements like **`<input>`** and **`<textarea>`** maintain their own state and update them when the input value changes.

```
<input type="text" name="name" ref="name" />
```

- The DOM can be queried for the value of an input field using a **`ref`**.
- The value needs to be pulled from the field when the form is submitted.

```
this.refs.name.value
```

UncontrolledForm.js
UnControlledFormApp.js

Uncontrolled components

- Default Values
  - In an uncontrolled component, React can be used to specify the initial value and leave the subsequent updates uncontrolled.
  - A `defaultValue` attribute can be used instead of the value
  - `checkbox` and `radio` button support `defaultChecked, select` and `textarea` support `defaultValue`

```
<input type="text" name="txtFname" ref="txtFname"
                                    defaultValue="Doe"/>
<input type="radio" name="optGender" ref="optGender"
                     value="male" defaultChecked/>
```

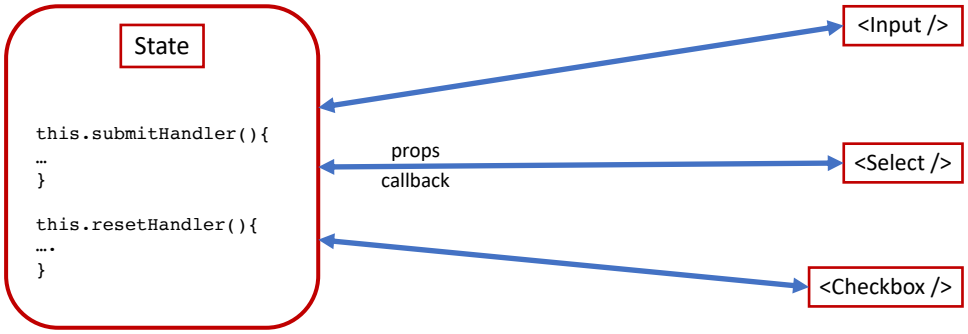
UncontrolledForm1.js
UnControlledFormApp.js

Controlled components

- The form data is handled by the component (React) rather than the DOM.
- The internal component state is always in sync with the view.
- The best practice includes:
  - Define elements in the `render()` using values from the state
  - Capture changes of a form element using `onChange()` as they happen.
  - Update the internal state in the event handler
  - New values are saved in state and the view is updated by a new `render()`

Controlled components

- Each form element gets a component of its own called dumb component(s).
- The container component maintains the state.
- This approach provides the component better control over form control elements and form data.



Controlled components

- React's composition model allows React components into smaller reusable code.
  - Each component represents an independent functional unit.
- Form container can be class based component and form elements can be functional/stateless components.
  - Container component handles state management, form submission.
- The dumb components are presentational components that contain the actual DOM markup.
  - These components receive data and callbacks as props

formContainer.js, formContainerApp.js
Button.js, Input.js, Checkbox.js, Select.js

Controlled components - validation

- React being a front-end library can facilitate building instant validation into a form component.
- State can also be used to help implement validation.
- Validation can be custom built or can be implemented with the help of pre-built packages such as '`validator`'

```
import fieldValidator from 'validator'
fieldValidator.isEmpty('')  //true
fieldValidator.isEmail('smith@home.net')   //true
```

formContainer1.js, formContainerApp.js
Button.js, Input.js, Checkbox.js, Select.js

Under the hood

Component Lifecycle

- React is a component based JavaScript library.
- React web applications are a collection of independent components that run based on the interaction made with them.
- Every React component has a lifecycle of its own.
  - Cases include; do something before or after a component has rendered
  - Avoid a re-render
- Lifecycle of a component can be defined as a series of methods that are invoked in different stages of the component's existence.
- Lifecycle methods are only accessible in stateful/class-based components

Component Lifecycle Stages

- All React component's lifecycle methods can be split into four phases:
  - **Initialization**
    - This is the stage where the component is constructed with defaults and initial values for Props and State.
  - **Mounting**
    - This is the process that occurs when the component is being inserted into the DOM
  - **Updating**
    - This is a stage when the state or properties of the component are updated and the application is repainted
  - **Unmounting**
    - The final stage of the component lifecycle when it is removed from the DOM

Component Lifecycle

- *Initialization* in many cases is set up as a part of mounting.
- *Mounting*
  - `constructor()`
    - Called before the component is mounted
    - Call `super(props)`
    - Initialize local state by assigning an object to `this.state`
    - Bind event handler methods
  - `render()`
    - Prepare and structure JSX code
    - Should not modify component state
    - Returns the same result every time when invoked
  - `componentDidMount()`
    - Invoked immediately after the component is inserted into the tree
    - Can be used to load data from a remote endpoint

Component Lifecycle

- *Updating*
  - `shouldComponentUpdate()`
    - Lets React know if a component's output is not affected by the current change in state or props.
    - Defaults to `true`
  - `render()`
  - `getSnapshotBeforeUpdate()`
    - Invoked before the most recently rendered output is committed to the DOM
  - `componentDidUpdate()`
    - Invoked immediately after updating occurs.
    - Can be used to operate on the DOM when the component has been updated

Component Lifecycle

- Unmounting
  - `componentWillUnmount()`
    - Invoked immediately before a component is unmounted and destroyed.
    - This is ideally used to perform cleanup activity
    - Once a component instance is unmounted it will not be mounted again
    - This denotes the end of the lifecycle

Clock.js
ClockApp.js

Routing

Routing

- Navigating from one page view to another is critical in an Single Page Application.
- Single Page Applications dynamically rewrite the current page instead of requesting entirely new pages from the server.
- SPA helps make web applications behave like desktop or mobile applications.
- Router provides a mechanism to distinguish between each page of the application.
- React Router is used to implement routing.

React Router

- The React Router is installed by using `npm`

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

- In the context of a browser environment React Router provides two kinds of routes
  - **`BrowserRouter`**
    - Helps build classic URLs and is recommended by default
    - May require additional server configuration to handle dynamic requests
  - **`HashRouter`**
    - Builds URLs with the hash
    - Can be a good solution for static websites

React Router

- The three important components when working with the React Router are:
  - BrowserRouter
    - Wraps all Route components
  - Link
    - Used to generate links to the routes
  - Route
    - Responsible for showing or hiding the component they contain

BrowserRouter

- A `BrowserRouter` component can have only one child element.
- The routing feature generally has to work for the entire application and therefore the `App` component is placed inside the `BrowserRouter` in `index.js`

```
import {BrowserRouter as Router} from 'react-router-dom';

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

RouterIndex.js

Structuring the application

- The `App` component can be used to structure the application for modularity and comprehension.

```
<div className='app'>
    <h1>React Router Demo</h1>
    <NavBar />
    <Outlet />
</div>
```

- The `NavBar` component can contain the navigation menu and `Outlet` component can be used to depict the content of each view.

App.css
RouterApp.js

Link

- The `Link` component is used to trigger new routes.
- The `Link` component can be added to point at different routes with the `to` attribute.
- `NavLink` is a special version of the `Link` component that helps add styling attributes to the rendered element if the URL is matched.

```
import { NavLink } from 'react-router-dom';

<ul>
<li><NavLink to='/'>Home</NavLink></li>
</ul>
```

NavBar.js

Route

- The `Route` component is used to create a route.
- It renders a component when the URL matches the Route's path.
- Multiple Routes can be grouped inside a `Switch` component to render only the first child `Route` that matches the current URL.

```
<Route path='/' component={Home}></Route>
```

- The `Route` component expects a path `prop` that describes the `path` that the route matches.
- The `component` prop is used to specify that component that should be rendered when a route matches.
- In order to have the render only when the path is an exact match use the `exact` prop on each of the routes

Outlet.js

The views

- All the components referred to in the `Route` return JSX
- This JSX will be rendered in the DOM when the routes for each component match the current URL.

```
const Home = () => {
      return(
            <div className='home'>
                  <h2>Welcome to our home on the web</h2>
                  <p> Feel free to browse around and learn more
                        about us.</p>
            </div>
            )
      }
```

Home.js, About.js, Contact.js

Dynamic routes using Parameters

- URL parameters help render the same component based on its dynamic URL.

```
<Route exact path='/team/:id' component={Team}></Route>

<li><NavLink to='/team/2'>Team</NavLink></li>

const Team = ({match}) => {
const memberID = match.params.id
```

NavBar1.js, Outlet1.js, Team.js

Talking to the Server

Fetching Data – AJAX Requests

- React itself does not have any bias towards a particular approach of fetching data.
- React may not even know if there is a server in the picture!
- React simply renders components using data from two places: props and state.
- In order to use some data from a server that data needs to be in the component's prop or state.

A HTTP library

- To fetch data from the server a HTTP library is required.
- There are multiple libraries available:
  - Use promises? Go with axios
  - Use callbacks? Check superagent
- *There are competing libraries and there is no "best"*
- It is ***believed*** that Axios and React pair nicely together.
- Install `axios` using `npm`

```
npm install axios --save
```

Using Axios

- Axios offers methods for all the HTTP verbs (GET, POST, PUT, DELETE).
- It performs automatic JSON data transformation.

```
import axios from ‘axios’;

state = {
Post : ''
}

axios.get('http://jsonplaceholder.typicode.com/posts/2')
.then(resp => {
          this.setState({Post:resp.data.title})
})
```

Post.js

Testing

Testing

- Unit Testing
  - Unit testing refers to testing individual pieces of code.
  - In React, Unit tests typically do not require a browser.
- Functional Testing
  - This involves testing the behavior of a component.
  - Functional tests usually run in isolation.
- Integration Testing
  - This tests the entire application and attempts to replicate an experience the end-user would have when using the application.

Testing - Tools

- Jest
  - Introduced by Facebook for component tests in React.
  - Jest is not only a testing library but also a test runner.
- Enzyme
  - Enzyme was introduced by Airbnb for component tests in React
  - Enzyme is used to render components, access elements, props etc and simulate events.
- Storybook
  - This can be used to render components in their different states.
  - Can be used to ensure components behave correctly in their different states

Testing

- `create-react-app` application comes with Jest as a test runner and assertion library.
  - It is the *official* test runner
- Jest includes all the goodness of Jasmine with its own improvements.
- Jest supports TypeScript and contains most of the functionality in one package and is the best option for React application testing.
- Enzyme is popular for its React testing utility functions that simplify writing assertions.

Jest Basics

- *App.test.js*

```
describe('Addition', () => {
      it('knows that 2 and 2 make 4', () =>
      { expect(2 + 2).toBe(4);
      });
});
```

- The test can be run using

```
npm test
```

Enzyme

- *Enzyme* is built to support different versions of React.
- *Enzyme* introduced adapters to gel well with React.
  - An adapter appropriate to the version of React being used is to be installed.

```
npm install --save-dev enzyme
npm install --save-dev enzyme-adapter-react-16
```

- `setupTests.js` file tells *Jest* and *Enzyme* the adapters that will be used

```
import { configure } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';
configure({ adapter: new Adapter() });
```

Enzyme

- App.test.js

```
import React from 'react';
import { shallow } from 'enzyme';
import App from './App';
  describe('App component', () => {
     it('has text Hello World', () => {
     const wrapper = shallow(<App />);
     const text = wrapper.find('p').text();
     expect(text).toEqual('Hello World');
  });
});
```

Summary

- Writing tests is an essential part of software development.
- React components are very *testable*
- With *Jest* and *Enzyme* developers can run declarations by gaining access to properties, state and child props of React components.
- *Jest* tooling is developer-friendly and intuitive.

Session Summary

- React is a JavaScript **library** for building user interfaces.
- Declarative views make code predictable.
- React helps build encapsulated components that manage their own state.
- React uses one-way data binding which makes code more predictable and simplifies debugging.
- React makes it faster and easier to build client facing web interfaces leveraging newer JavaScript features.

Resources

- Awesome React - https://github.com/enaqx/awesome-react
  - A collection of awesome things regarding React ecosystem
- React components - https://github.com/brillout/awesome-react-components
  - A collection of react components
- Books
  - Learn ReactJS fast
  - React in action
  - Learning Web Development with React and Bootstrap