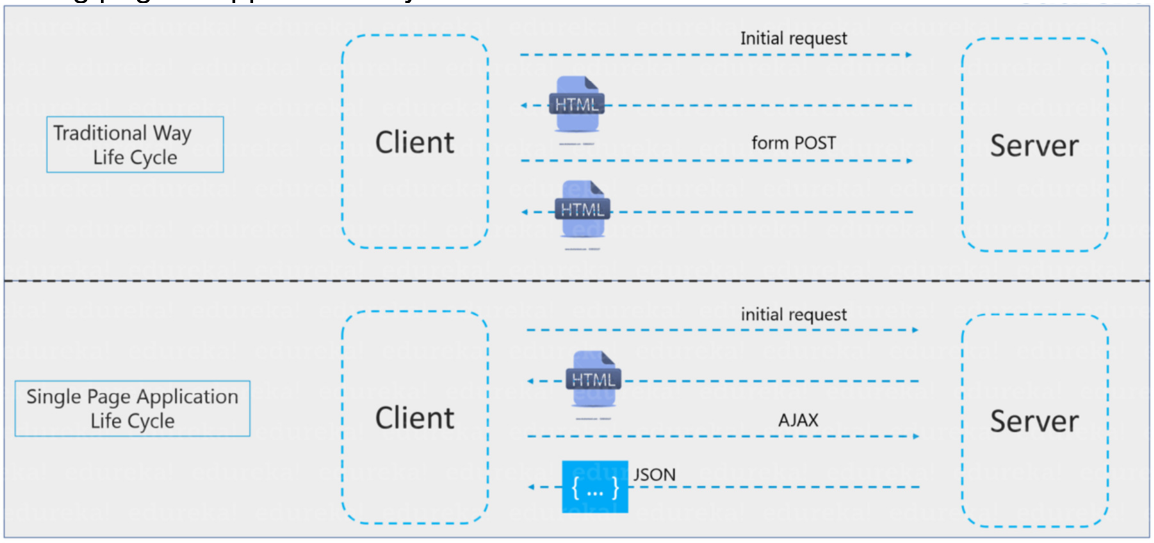
SCS1620 User Interface Technologies
Unit V – AngularJS and ReactJS

Introduction to Angular 4.0 - Needs & Evolution – Features – Setup and Configuration – Components and Modules – Templates – Change Detection - Directives – Data Binding - Pipes – Nested Components. Template Driven Forms - Model Driven Forms or Reactive Forms - Custom Validators. Introduction to ReactJS- React Components- Build a simple React component- React internals – Component inter communication- Component composition- Component styling

Introduction

Angular is a client-side JavaScript framework that was specifically designed to help developers build SPAs (Single Page Applications) in accordance with best practices for web development.Single-page application (or SPA) are applications that are accessed via a web browser like other websites but offer more dynamic interactions resembling native mobile and desktop apps. The most notable difference between a regular website and SPA is the reduced amount of page refreshes. SPAs have a heavier usage of AJAX- a way to communicate with back-end servers without doing a full page refresh to get data loaded into our application. As a result, the process of rendering pages happens mostly on the client-side.



Single Page Application vs Traditional Web Application

Angular is a TypeScript-based open-source front-end web application platform led by the Angular Team at Google and by a community of individuals and corporations. Angular is a complete rewrite from the same team that built AngularJS.

- The architecture of an Angular application is different from AngularJS. The main building blocks for Angular are modules, components, templates, metadata, data binding, directives, services and dependency injection. We will be looking at it in a while.
- Angular was a complete rewrite of AngularJS.
- Angular does not have a concept of “scope” or controllers instead, it uses a hierarchy of components as its main architectural concept.
- Angular has a simpler expression syntax, focusing on “[ ]” for property binding, and “( )” for event binding

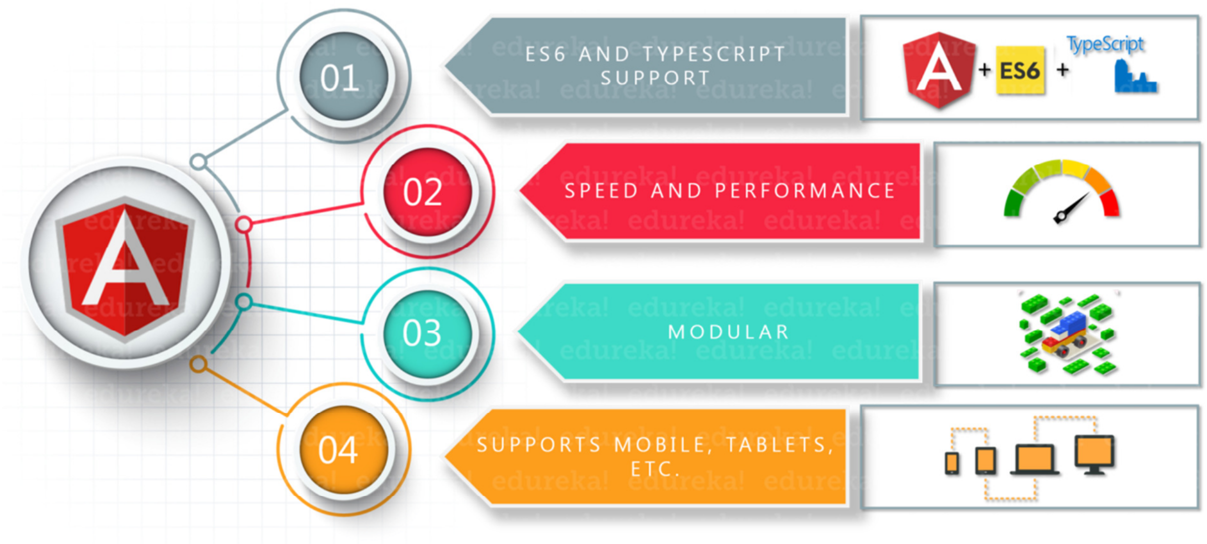
- **Mobile development** – Desktop development is much easier when mobile performance issues are handled first. Thus, Angular first handles mobile development.
- **Modularity** – Angular follows modularity. Similar functionalities are kept together in same modules. This gives Angular a lighter & faster core.

Angular recommends the use of Microsoft's TypeScript language, which introduces the following features:

- Class-based Object Oriented Programming
- Static Typing

TypeScript is a superset of ECMAScript 6 (ES6) and is backward compatible with ECMAScript 5. Angular also includes the benefits of ES6:

  - Iterators
  - For/Of loops
  - Reflection
  - Improved dependency injection – bindings make it possible for dependencies to be named
  - Dynamic loading
  - Asynchronous template compilation
  - Simpler Routing
  - Replacing controllers and $scope with components and directives – a component is a directive with a template
- Support reactive programming using RxJS



Features of Angular

Cross Platform

- **Progressive web apps**: It uses modern web platform capabilities to deliver an app-like experience. It gives high performance, offline, and zero-step installation. So, working with Angular is pretty much easy.
- **Native:** Builds native mobile apps with strategies using Ionic Framework, NativeScript, and React Native.
- **Desktop:** Create desktop-installed apps across Mac, Windows, and Linux using the same Angular methods you've learned for the web plus.

Speed and Performance

- **Code generation**: Angular turns your templates into code that's highly optimized for JavaScript virtual machines, giving you all the benefits of hand-written code with the productivity of a framework.
- **Universal:** Any technology can be used with Angular for serving the application like node.js, .NET, PHP and other servers.
- **Code splitting:** Angular apps load quickly with the new Component Router, which delivers automatic code-splitting, so users only load code required to render the view they request.
-

Productivity

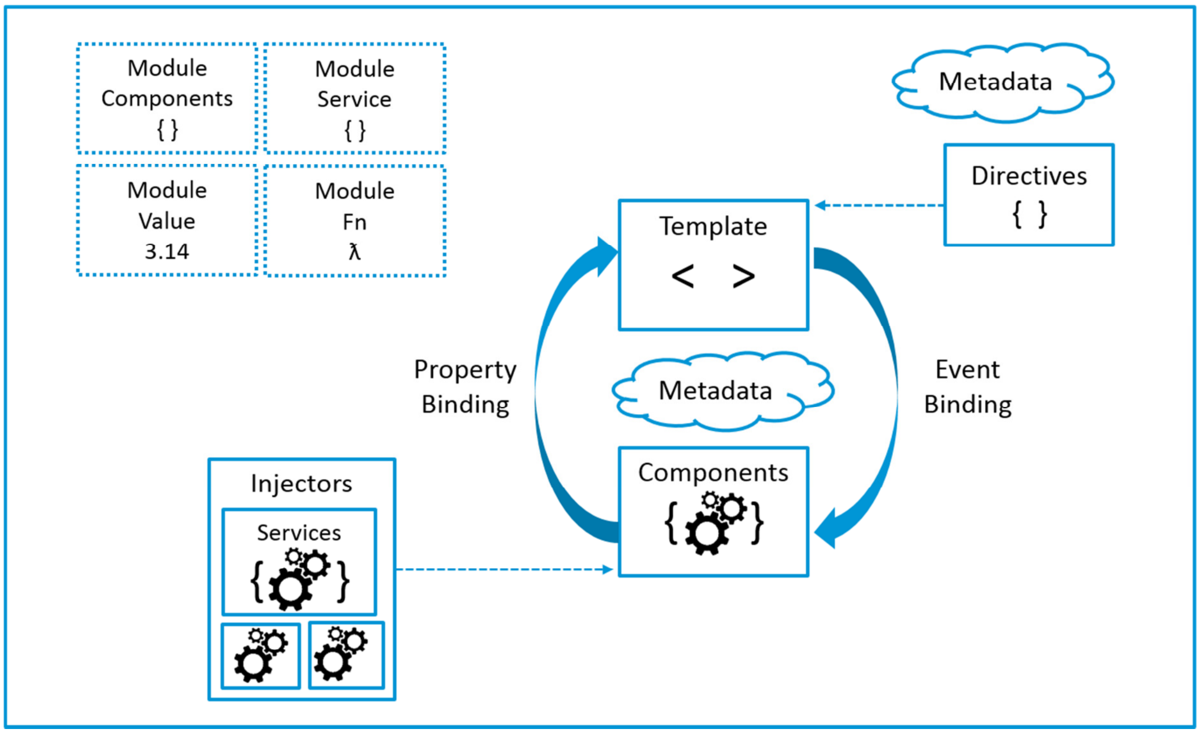
- **Templates**: Quickly create UI views with simple and powerful template syntax.
- **Angular CLI:** Command line tools: Can easily and quickly start building components, adding components, testing them, and then, instantly deploy them using Angular CLI.
- **IDEs:** Get intelligent code completion, instant errors, and other feedback in popular editors and IDEs like Microsoft's VS Code.

Full Development Story

- **Testing**: With Karma for unit tests, you can identify your mistake on the fly and Protractor makes your scenario tests run faster and in a stable manner.

Building Blocks of Angular

- The main building blocks of Angular are:
- Modules
- Components
- Templates
- Metadata
- Data binding
- Directives
- Services
- Dependency injection



AngularJS Architecture

Modules

Angular apps are modular and to maintain modularity, there exists *Angular modules* or *NgModules*. Every Angular app contains at least one Angular module, i.e. the root module. Generally, it is named as *AppModule*. The *root module* can be the only module in a small application. Most of the apps have multiple modules. A module is a cohesive block of code with a related set of capabilities which have a specific application domain or a workflow. Any angular module is a class with @NgModule decorator.

Decorators are functions that modify JavaScript classes. Decorators are basically used for attaching metadata to classes so that, it knows the configuration of those classes and how they should work. *NgModule* is a decorator function that takes metadata object whose properties describe the module. The properties are:

- ***declarations:*** The classes that are related to views and it belong to this module. There are three classes of Angular that can contain view: components, directives and pipes. We will talk about them in a while.
- ***exports:*** The classes that should be accessible to the components of other modules.
- ***imports:*** Modules whose classes are needed by the component of this module.
- ***providers:*** Services present in one of the modules which is to be used in the other modules or components. Once a service is included in the providers it becomes accessible in all parts of that application
- ***bootstrap:*** The *root component* which is the main view of the application. This root module only has this property and it indicates the component that is to be bootstrapped.

Template of root module (i.e. ***src/app/app.module.ts***):

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
```

```
@NgModule({
        imports:[ BrowserModule ],
        providers: [ BookList ],
        declarations: [ AppComponent ],
        exports: [],
        bootstrap: [ AppComponent ]
   })
 export class AppModule { }
```

A root module generally doesn’t *export* it’s class because as root module is the one which imports other modules & components to use them. The *AppModule is bootstrapped* in a *main.ts* file, where the bootstrap module is specified and inside the bootstrap module, contains the bootstrap component.

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';
import { environment } from './environments/environment';
if (environment.production) {
        enableProdMode();
}
platformBrowserDynamic().bootstrapModule(AppModule);
```

Angular libraries

Angular gives a collection of JavaScript modules (library modules) which provide various functionalities.
Each Angular library has @angular prefix, like

@angular/core,
@angular/compiler,
@angular/compiler-cli,
@angular/http,
@angular/router.

using the npm package manager the libraries can be installed and import parts of them with JavaScript import statements.

Example:

```
        import { Component } from '@angular/core';
```

Components

A *component* controls one or more section on the screen called a *view*. For example, for a movie list application, you can have components like App Component (*the bootstrapped component)*, Movielist Component, Movie Description Component, etc.
Inside the component, component’s application logic is defined i.e. how does it support the view—inside a class.
The class interacts with the view through an API of properties and methods.
Every app has a main component which is bootstrapped inside the main module, i.e AppComponent.

```
import { Component } from '@angular/core';
@Component({
        selector:'app-root',
        templateUrl:'./app.component.html',
        styleUrls: ['./app.component.css']
  })
export class AppComponent{
title = 'app works!';
}
```

Templates
A template is a form of HTML tags that tells Angular about how to render the component.
A template looks like regular HTML, except for a few differences.
Example

```
        <app-navbar></app-navbar>
        <div class ="container">
                <flash-messages></flash-messages>
                <router-outlet></router-outlet>
        </div>
```

Metadata
Metadata tells Angular how to process a class.
To inform Angular that MovieList Component is a component, metadata is attached to the class.
In TypeScript, attach metadata by using a decorator.

```
import { Component, OnInit } from '@angular/core';
@Component({
        selector: 'app-movies',
        templateUrl: './movies.component.html',
        styleUrls: ['./movies.component.css']
  })
```

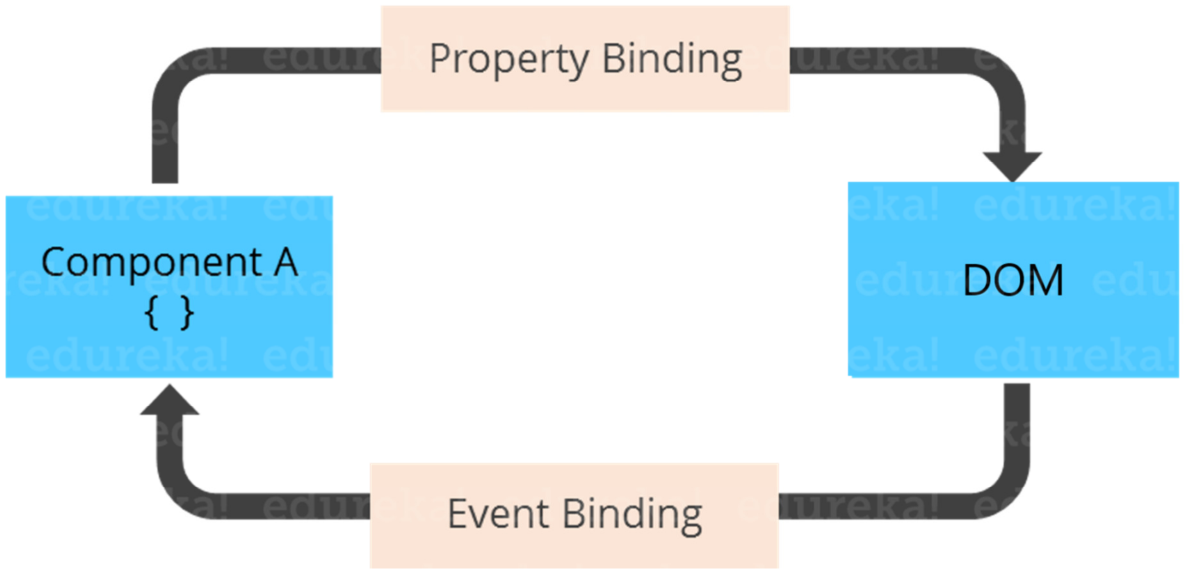
Here is the @Component decorator, which identifies the class immediately below it as a component class. The @Component decorator takes the required configuration object which Angular needs to create and present the component and its view.
The most important configurations of @Component decorator are:

- *selector*: Selector tells Angular to create and insert an instance of this component where it finds <app-movies> tag. For example, if an app's HTML contains <app-movies></app-movies>, then Angular inserts an instance of the MovieListComponent view between those tags.
- *templateUrl*: It contains the path of this component's HTML template.
- *providers*: An array of **dependency injection providers** for services that the component requires. This is one way to tell Angular that the component's constructor requires a *MovieService* to get the list of movies to display.

The metadata in the @Component tells Angular where to get the major building blocks to be specified for the component. *The template, metadata, and component together describe a view.*

Data Binding

If framework is not used, data values are to be pushed into the HTML controls and turn user responses into some actions and value updates.
Writing such push/pull logic is tedious, error-prone, and a nightmare to read. Angular supports data binding, a mechanism for coordinating parts of a template with parts of a component.
Add binding markup to the template HTML and inform Angular how to connect both sides.
Each form has a direction — to the DOM, from the DOM, or in both directions.



```
<li> {{movie.name}}</li>
<movie-detail [movie]="selectedMovie"></movie-detail>
<li (click)="selectMovie(Movie)"></li>
```

- The {{movie.name}} *interpolation* displays the component's name property value within the <li> element.
- The [movie] *property binding* passes the value of selectedMovie from the parent MovieListComponent to the movie property of the child MovieDetailComponent.
- The (click) *event binding* calls the component's selectMovie method when the user clicks a movies's name.

Two-way data binding is an important part as it combines property and event binding in a single notation, using the ngModel directive.

```
<input [(ngModel)]="movie.name">
```

In two-way binding, a data property value flows to the input box from the component as with property binding. The user's changes also flow back to the component, resetting the property to the latest value, as with event binding. Angular processes all data bindings once per JavaScript event cycle, from the root of the application component tree through all child components.
Data binding plays an important role in communication between a template and its component. Data binding is also important for communication between parent and child components.

Directives

- Angular templates are dynamic. When Angular renders them, it transforms the DOM according to the instructions given by directives.
- A directive is a class with a @Directive decorator. A component is a directive-with-a-template; a @Component decorator is actually a @Directive decorator extended with template-oriented features.
- While a component is technically a directive, components are so distinctive and central to Angular applications that this architectural overview separates components from directives.
- Two other kinds of directives exist: structural and attribute directives.
- Directive tends to appear within an element tag as attributes do, sometimes by name but more often as the target of an assignment or a binding.
- **Structural** directives alter layout by adding, removing, and replacing elements in DOM.

Two built-in structural Directives

```
<li *ngFor="let movie of movies"></li>
<movie-detail *ngIf="selectedMovie"></movie-detail>
```

*ngFor tells Angular to retrieve one <li> per movie in the movies
*ngIf includes the MovieDetail component only if a selected movie exists.

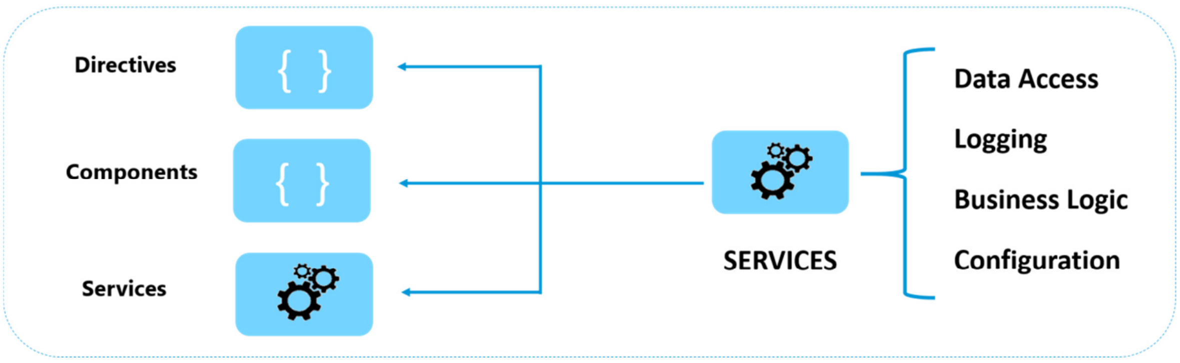
Attribute directives alter the appearance or behavior of an existing element.
In templates, attributes are like regular HTML attributes.
The ngModel directive, which implements two-way data binding, is an example of an attribute directive.
ngModel modifies the behavior of an existing element by setting its display value property and responding to change events.

```
<input [(ngModel)]="movie.name">
```

Angular has a few more directives that either alter the layout structure (for example, ngSwitch) or modify aspects of DOM elements and components (for example, ngStyle and ngClass).
Custom directives can also be created.

Services
Service is a broad category encompassing any value, function, or feature that the application is in need of.
A service is typically a class with a well-defined purpose.
Anything can be a service.
Examples include:

- logging service
- data service
- message bus
- tax calculator
- application configuration



Angular has no definition of a service.
There is no service base class, and no place to register a service.
Yet services are fundamental to any Angular application.
Components are the consumers of services.

Environmental Setup

Nodejs
NPM (Will be installed along with NodeJs)

To install angularjs

```
npm install –g @angular/cli
```

To check the installation versions run the following in command line mode.

```
node –v
npm –v
ng -v
```

To create a new project:
ng new angular <appName>

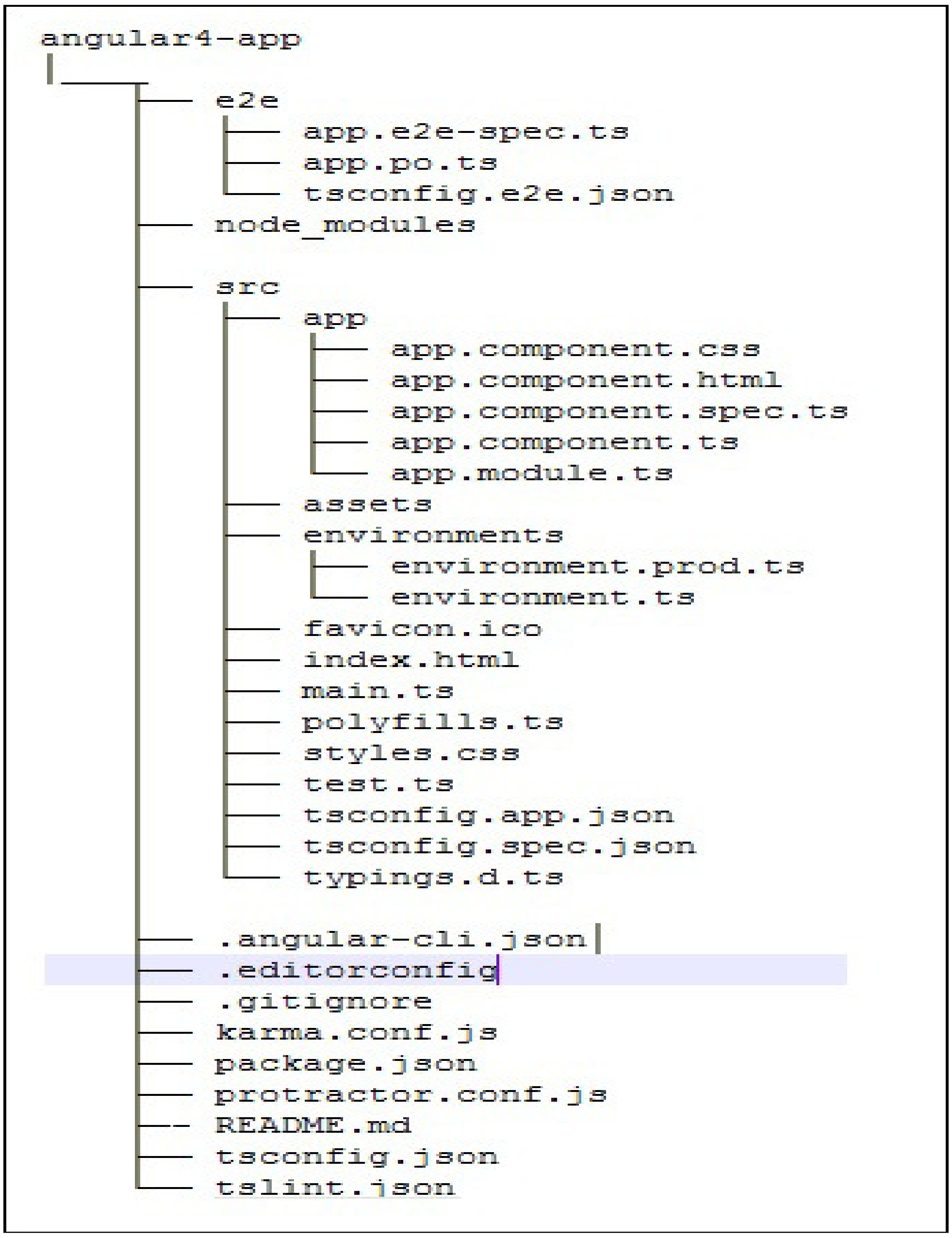
To start the server
ng serve

By default the web servers gets started in the port 4200

To test, launch the browser:

```
localhost:4200/
```

The project directory structure of an angularjs is as follows:



```
angular4-app
|_____
      ├── e2e
      │     ├── app.e2e-spec.ts
      │     ├── app.po.ts
      │     └── tsconfig.e2e.json
      ├── node_modules
      │
      ├── src
      │     ├── app
      │     │     ├── app.component.css
      │     │     ├── app.component.html
      │     │     ├── app.component.spec.ts
      │     │     ├── app.component.ts
      │     │     └── app.module.ts
      │     ├── assets
      │     ├── environments
      │     │     ├── environment.prod.ts
      │     │     └── environment.ts
      │     ├── favicon.ico
      │     ├── index.html
      │     ├── main.ts
      │     ├── polyfills.ts
      │     ├── styles.css
      │     ├── test.ts
      │     ├── tsconfig.app.json
      │     ├── tsconfig.spec.json
      │     └── typings.d.ts
      │
      ├── .angular-cli.json
      ├── .editorconfig
      ├── .gitignore
      ├── karma.conf.js
      ├── package.json
      ├── protractor.conf.js
      ├── README.md
      ├── tsconfig.json
      └── tslint.json
```

The Angular 4 app folder has the following **folder structure** –

- **e2e** – end to end test folder. Mainly e2e is used for integration testing and helps ensure the application works fine.
- **node_modules** – The npm package installed is node_modules. You can open the folder and see the packages available.
- **src** – This folder is where we will work on the project using Angular 4.

The Angular 4 app folder has the following **file structure** –

- **.angular-cli.json** – It basically holds the project name, version of cli, etc.
- **.editorconfig** – This is the config file for the editor.
- **.gitignore** – A .gitignore file should be committed into the repository, in order to share the ignore rules with any other users that clone the repository.
- **karma.conf.js** – This is used for unit testing via the protractor. All the information required for the project is provided in karma.conf.js file.
- **package.json** – The package.json file tells which libraries will be installed into node_modules when you run npm install.

- **protractor.conf.js** – This is the testing configuration required for the application.
- **tsconfig.json** – This basically contains the compiler options required during compilation.
- **tslint.json** – This is the config file with rules to be considered while compiling.

The **src folder** is the main folder, which **internally has a different file structure**.

app

It contains the files described below. These files are installed by angular-cli by default.

- **app.module.ts** – If you open the file, you will see that the code has reference to different libraries, which are imported. Angular-cli has used these default libraries for the import – angular/core, platform-browser. The names itself explain the usage of the libraries.

They are imported and saved into variables such as **declarations, imports, providers**, and **bootstrap**.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})

export class AppModule { }
```

declarations – In declarations, the reference to the components is stored. The Appcomponent is the default component that is created whenever a new project is initiated. We will learn about creating new components in a different section.

imports – This will have the modules imported as shown above. At present, BrowserModule is part of the imports which is imported from @angular/platform-browser.

providers – This will have reference to the services created. The service will be discussed in a subsequent chapter.

bootstrap – This has reference to the default component created, i.e., AppComponent.

- **app.component.css** – You can write your css structure over here. Right now, we have added the background color to the div as shown below.

```
.divdetails{
  background-color: #ccc;
}
```

- **app.component.html** – The html code will be available in this file.

```
<!--The content below is only a placeholder and can be replaced.-->
<div class = "divdetails">
```

```
  <div style = "text-align:center">
    <h1>
      Welcome to {{title}}!
    </h1>
    <img width = "300" src =
"data:image/svg+xml;base64,PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNv...">
  </div>
  <h2>Here are some links to help you start: </h2>
  <ul>
    <li>
      <h2>
        <a target = "_blank" href="https://angular.io/tutorial">Tour of Heroes</a>
      </h2>
    </li>
    <li>
      <h2>
        <a target = "_blank" href = "https://github.com/angular/angular-cli/wiki">
          CLI Documentation
        </a>
      </h2>
    </li>
    <li>
      <h2>
        <a target="_blank" href="http://angularjs.blogspot.ca/">Angular blog</a>
      </h2>
    </li>
  </ul>
</div>
```

This is the default html code currently available with the project creation.

- **app.component.spec.ts** – These are automatically generated files which contain unit tests for source component.
- **app.component.ts** – The class for the component is defined over here. You can do the processing of the html structure in the .ts file. The processing will include activities such as connecting to the database, interacting with other components, routing, services, etc.

The structure of the file is as follows –

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app';
}
```

Assets

You can save your images, js files in this folder.

Environment
This folder has the details for the production or the dev environment. The folder contains two files.

- environment.prod.ts
- environment.ts

Both the files have details of whether the final file should be compiled in the production environment or the dev environment.
The additional file structure of Angular 4 app folder includes the following –
favicon.ico : This is a file that is usually found in the root directory of a website.
index.html : This is the file which is displayed in the browser.

```
<!doctype html>
<html lang = "en">
  <head>
    <meta charset = "utf-8">
    <title>HTTP Search Param</title>
    <base href = "/">
    <link href = "https://fonts.googleapis.com/icon?family=Material+Icons"
rel="stylesheet">
    <link href = "https://fonts.googleapis.com/css?family=Roboto|Roboto+Mono"
rel="stylesheet">
    <link href = "styles.c7c7b8bf22964ff954d3.bundle.css" rel="stylesheet">
    <meta name = "viewport" content="width=device-width, initial-scale=1">
    <link rel = "icon" type="image/x-icon" href="favicon.ico">
  </head>

  <body>
    <app-root></app-root>
  </body>
</html>
```

The body has **<app-root></app-root>**. This is the selector which is used in **app.component.ts** file and will display the details from app.component.html file.

main.ts is the file from where we start our project development. It starts with importing the basic module which we need. Right now if you see angular/core, angular/platform-browser-dynamic, app.module and environment is imported by default during angular-cli installation and project setup.

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';
import { environment } from './environments/environment';
if (environment.production) {
  enableProdMode();
}
platformBrowserDynamic().bootstrapModule(AppModule);
```

The **platformBrowserDynamic().bootstrapModule(AppModule)** has the parent module reference **AppModule**. Hence, when it executes in the browser, the file that is called is index.html. Index.html internally refers to main.ts which calls the parent module, i.e., AppModule when the following code executes –

platformBrowserDynamic().bootstrapModule(AppModule);

When AppModule is called, it calls app.module.ts which further calls the AppComponent based on the boostrap as follows –

bootstrap: [AppComponent]

In app.component.ts, there is a **selector: app-root** which is used in the index.html file. This will display the contents present in app.component.html.

polyfill.ts
This is mainly used for backward compatibility.

styles.css
This is the style file required for the project.

test.ts
Here, the unit test cases for testing the project will be handled.

tsconfig.app.json
This is used during compilation, it has the config details that need to be used to run the application.

tsconfig.spec.json
This helps maintain the details for testing.

typings.d.ts
It is used to manage the TypeScript definition.

Components

Major part of the development with Angular 4 is done in the components. Components are basically classes that interact with the .html file of the component, which gets displayed on the browser. We have seen the file structure in one of our previous chapters. The file structure has the app component and it consists of the following files –

- **app.component.css**
- **app.component.html**
- **app.component.spec.ts**
- **app.component.ts**
- **app.module.ts**

The above files were created by default when we created new project using the angular-cli command.
If you open up the **app.module.ts** file, it has some libraries which are imported and also a declarative which is assigned the appcomponent

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

To create a new Component:

```
ng g component new-cmp
```

```
installing component
  create src\app\new-cmp\new-cmp.component.css       // CSS
  create src\app\new-cmp\new-cmp.component.html      // HTML
  create src\app\new-cmp\new-cmp.component.spec.ts   // unit testing
  create src\app\new-cmp\new-cmp.component.ts        // module, properties, etc
  update src\app\app.module.ts                       // Updation of root app
```

In the app.component.html

```
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
<!--To include the design of the new components -->
  <app-new-cmp></app-new-cmp>
  <app-my-new-component></app-my-new-component>
</div>
```

Modules

Module in Angular refers to a place where you can group the components, directives, pipes, and services, which are related to the application.

In case you are developing a website, the header, footer, left, center and the right section become part of a module.

To define module, we can use the **NgModule**. When you create a new project using the Angular –cli command, the ngmodule is created in the app.module.ts file by default

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { FormsModule } from @angular/forms;
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]  // Starting the execution
})
export class AppModule { }
```

To import NgModule
import { NgModule } from '@angular/core';

Data Binding
Curly braces are used for data binding. {{}} – this process is called as interpolation.

The variable in the **app.component.html** file is referred as {{title}} and the value of title is initialized in the **app.component.ts** file and in **app.component.html**, the value is displayed.

Dropdown to create a list of departments in the browser.

In the app.component.ts, declare the array deptt and assign values to it.

```
export class AppComponent {
 title = 'CSE - Angular JS Demo';
 deptt = ["CSE", "IT",
         "ECE", "EEE", "ETCE", "EIE",
         "MECH", "AERO", "PROD", "AUTO",
         "BME", "BTE", "BIN", "CHEM",
         "CIVIL", "ARCH"];
         isAvailable = true;
}
```

In the app.component.html, add the following code:

```
div>
   Departments :
   <select>
     <option *ngFor="let i of deptt">{{i}}</option>
   </select>
 </div>
<div>
   <span *ngIf="isAvailable; then condition1 else condition2">Condition is
valid.</span>
```

```
    <ng-template #condition1>Condition is valid</ng-template>
    <ng-template #condition2>Condition is invalid</ng-template>
</div>
```

Handling events

In app.component.html

```
<div>
    Departments :
    <select (change) = "changedeptt($event)">
      <option *ngFor="let i of deptt">{{i}}</option>
    </select>
  </div>
<button (click) = "myClickFunction($event)">Click Me</button>
```

In app.component.ts

```
myClickFunction(event) {
    alert("Button is clicked");
    console.log(event);
 }
 changedeptt(event) {
    console.log("Changed month from the Dropdown");
    console.log(event);
 }
```

Directives

Directives in Angular is a **js** class, which is declared as **@directive**. We have 3 directives in Angular. The directives are listed below –

Component Directives:

These form the main class having details of how the component should be processed, instantiated and used at runtime.

Structural Directives

A structure directive basically deals with manipulating the DOM elements. Structural directives have a * sign before the directive. For example, ***ngIf** and ***ngFor**.

Attribute Directives

Attribute directives deal with changing the look and behavior of the dom element.

Pipes

The | character is used to transform data.

```
{{ Welcome to Angular 4 | lowercase}}
```

It takes integers, strings, arrays, and date as input separated with **|** to be converted in the format as required and display the same in the browser.

```
<b>{{title | uppercase}}</b><br/>
<b>{{title | lowercase}}</b>
```

Built-in pipes are:

- Lowercasepipe
- Uppercasepipe
- Datepipe

- Currencypipe
- Jsonpipe
- Percentpipe
- Decimalpipe
- Slicepipe

Example
app.component.ts:

```
todaydate = new Date();
jsonval = {school:'Computing', deptt:'CSE', address:{course1:'B.E.', course2:'M.E.',
course3: ‘B.Sc’, course4:’PhD’}};
```

app.component.html

```
<div style = "width:100%;">
  <div style = "width:40%;float:left;border:solid 1px black;">
    <h1>Uppercase Pipe</h1>
    <b>{{title | uppercase}}</b><br/>
    <h1>Lowercase Pipe</h1>
    <b>{{title | lowercase}}</b>
    <h1>Currency Pipe</h1>
    <b>{{6589.23 | currency:"USD"}}</b><br/>
    <b>{{6589.23 | currency:"USD":true}}</b> //Boolean true is used to get the sign
of the currency.
    <h1>Date pipe</h1>
    <b>{{todaydate | date:'d/M/y'}}</b><br/>
    <b>{{todaydate | date:'shortTime'}}</b>
    <h1>Decimal Pipe</h1>
    <b>{{ 454.78787814 | number: '3.4-4' }}</b> // 3 is for main integer, 4 -4 are for
integers to be displayed.
  </div>
  <div style = "width:40%;float:left;border:solid 1px black;">
    <h1>Json Pipe</h1>
    <b>{{ jsonval | json }}</b>
    <h1>Percent Pipe</h1>
    <b>{{00.54565 | percent}}</b>
    <h1>Slice Pipe</h1>
    <b>{{deptt | slice:2:6}}</b>
    // here 2 and 6 refers to the start and the end index
  </div>
</div>
```

Routing
Routing basically means navigating between pages. You have seen many sites with links that direct you to a new page. This can be achieved using routing. Here the pages that we are referring to will be in the form of components.

app.module.ts

```
import { RouterModule} from '@angular/router';

imports: [
    BrowserModule,
    RouterModule.forRoot([
      {
         path: 'new-cmp',
         component: NewCmpComponent
      }
   ])
  ],
```

new-cmp.component.ts

```
export class NewCmpComponent implements OnInit {
  newcomponent = "Using Routing through Components";
  constructor() { }
  ngOnInit() {
  }
}
```

new-cmp.component.html

```
<p>
  New Component Successfully Created...
  {{newcomponent}}
</p>
```

app.component.html

```
<a routerLink = "new-cmp">New component</a>
<router-outlet></router-outlet>
```

Forms:

- Template Driven Form
- Model Driven Form

With a template driven form, most of the work is done in the template; and with the model driven form, most of the work is done in the component class.

To create Template driven form. To create a simple login form and add the email id, password and submit the button in the form.

To start with, we need to import to FormsModule from **@angular/core** which is done in **app.module.ts**

In app.module.ts

```
import { FormsModule } from '@angular/forms';

imports: [
    BrowserModule,
    FormsModule
  ],
```

In app.component.ts

```
export class AppComponent {
  title = 'Forms using AngularJS 4.0';
  onClickSubmit(data) {
    alert("Entered Email id : " + data.emailid + data.passwd);
 }
```

In app.component.html

```
<form #userlogin = "ngForm" (ngSubmit) = "onClickSubmit(userlogin.value)" >
  Email Id: <input type = "text" name = "emailid" placeholder = "emailid" ngModel>
  <br/><br/>
  Password: <input type = "password" name = "passwd" placeholder = "passwd"
ngModel>
  <br/><br/>
  <input type = "submit" value = "Submit">
</form>
```

Using Model Forms
In app.module.ts

```
import { ReactiveFormsModule } from '@angular/forms';
imports: [
    BrowserModule,
    ReactiveFormsModule,
 ],
```

In app.component.ts

```
import { FormGroup, FormControl } from '@angular/forms';
```

In cmp-model-form.components.ts

```
import { FormGroup, FormControl } from '@angular/forms';
export class CmpModelFormComponent implements OnInit {
  formdata;
  emailid;
  password;
  constructor() { }

  ngOnInit() {
    this.formdata = new FormGroup({
       emailid: new FormControl("test@gmail.com"),
       passwd: new FormControl("abcd1234")
    });
  }
  onClickSubmit(data) {
    this.emailid = data.emailid;
    this.password = data.passwd;
  }
}
```

In cmp-model-form.component.html

```
<div>
```

```
  <form [formGroup]="formdata" (ngSubmit) = "onClickSubmit(formdata.value)" >
     <input type="text" class="fortextbox" name="emailid" placeholder="emailid"
        formControlName="emailid">
     <br/>
     <input type="password" class="fortextbox" name="passwd"
        placeholder="passwd" formControlName="passwd">
     <br/>
     <input type="submit" class="forsubmit" value="Log In">
  </form>
</div>
<p>
  Email entered is : {{emailid}}
  <br/><br/>
  Password is : {{password}}
</p>
```

Form Validation

import Validators from @angular/forms

Angular has built-in validators such as mandatory field, minlength, maxlength, and pattern. These are to be accessed using the Validators module.

In cmp-validate-form.components.ts

```
import { FormGroup, FormControl, Validators} from '@angular/forms';

export class CmpValidateFormComponent implements OnInit {
  emailid;
  passwd;
  formdata;
  ngOnInit() {
     this.formdata = new FormGroup({
        emailid: new FormControl("", Validators.compose([
           Validators.required,
           Validators.pattern("[^ @]*@[^ @]*")
        ])),
        // Custom Validation Function
        passwd: new FormControl("", this.passwordvalidation)
     });
  }
  passwordvalidation(formcontrol) {
    console.log(formcontrol.value);
    console.log("Password:" + formcontrol.value.length);
    if (formcontrol.value.length < 5)
       return {"passwd" : true};
    }
  onClickSubmit(data) {this.emailid = data.emailid;}
}
```

In cmp-validate-form.component.html

```
<div>
  <form [formGroup] = "formdata" (ngSubmit) = "onClickSubmit(formdata.value)" >
     <input type = "text" class = "fortextbox" name = "emailid" placeholder = "emailid"
        formControlName = "emailid">
     <br/>
     <input type = "password" class = "fortextbox" name = "passwd"
        placeholder = "passwd" formControlName = "passwd">
     <br/>
     <input type = "submit" [disabled] = "!formdata.valid" class = "forsubmit"
        value = "Log In">
  </form>
</div>

<p>
  Email entered is : {{emailid}}
</p>
```

ReactJS

React is a front-end library developed by Facebook. It is used for handling the view layer for web and mobile apps. ReactJS allows us to create reusable UI components. It is currently one of the most popular JavaScript libraries and has a strong foundation and large community behind it.

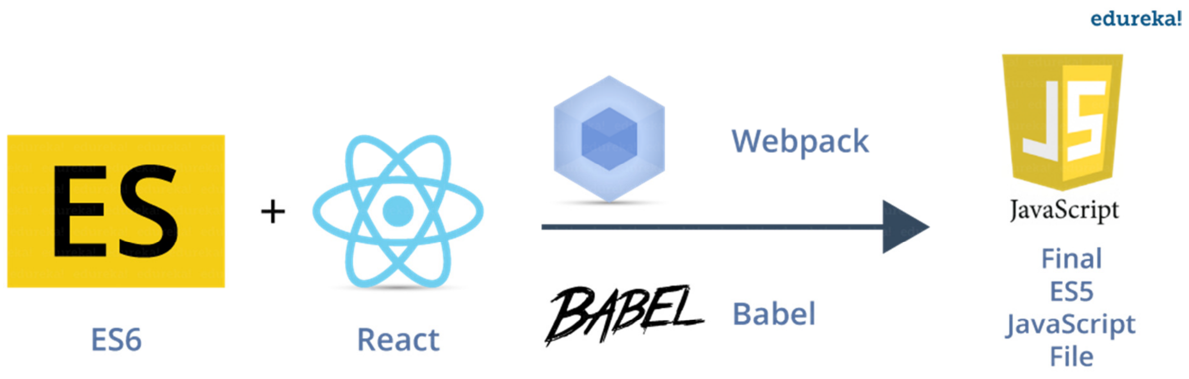
According to React official documentation, following is the definition –

React is a library for building composable user interfaces. It encourages the creation of reusable UI components, which present data that changes over time. Lots of people use React as the V in MVC. React abstracts away the DOM from you, offering a simpler programming model and better performance. React can also render on the server using Node, and it can power native apps using React Native. React implements one-way reactive data flow, which reduces the boilerplate and is easier to reason about than traditional data binding.

Evolution Of React

React is a JavaScript library used to build the user interface for web applications. React was initially developed and maintained by the folks at Facebook, which was later used in their products (WhatsApp & Instagram). Now it is an open source project with an active developer community. Popular websites like Netflix, Airbnb, Yahoo!Mail, KhanAcademy, Dropbox and many more use React to build their UI. Modern websites are built using MVC (model view controller) architecture. React is the 'V' in the MVC which stands for view, whereas the architecture is provided by Redux or Flux. React native is used to develop mobile apps, the Facebook mobile app is built using React native.

Facebook's annual F8 Developer conference 2017, saw two promising announcements: React Fiber and ReactVR. React Fiber is a complete rewrite of the previous release focusing on incremental rendering and quick responsiveness, React Fiber is backward compatible with all previous versions. ReactVR is built on top of React Native frameworks, it enables developing UI with the addition of 3D models to replicate 360-degree environment resulting in fully immersive VR content.



React Features

- **JSX** – JSX is JavaScript syntax extension. It isn't necessary to use JSX in React development, but it is recommended. JSX Allows us to include 'HTML' in the same file along with ‘JavaScript’ (HTML+JS=JSX). Each component in React generates some HTML which is rendered by the DOM.

- **ES6 (ES2015) -** The sixth version of JavaScript is standardized by ECMA (European Computer Manufacturers Association) International in 2015. Hence the language is referred to as ECMAScript. ES6 is not completely supported by all modern browsers.
- **ES5(ES2009)** - This is the fifth JavaScript version and is widely accepted by all modern browsers, it is based on the 2009 ECMA specification standard. Tools are used to convert ES6 to ES5 during runtime.
- **Webpack -** A module bundler which generates a build file joining all the dependencies.
- **Babel -** This is the tool used to convert ES6 to ES5. This is done because not all web browsers can render React (ES6+JSX) directly.
- **Components** – React is all about components. You need to think of everything as a component. This will help you maintain the code when working on larger scale projects.
- **Unidirectional data flow and Flux** – React implements one-way data flow which makes it easy to reason about your app. Flux is a pattern that helps keeping your data unidirectional.
- **License** – React is licensed under the Facebook Inc. Documentation is licensed under CC BY 4.0.

React Advantages

- Uses virtual DOM which is a JavaScript object. This will improve apps performance, since JavaScript virtual DOM is faster than the regular DOM.
- Can be used on client and server side as well as with other frameworks.
- Component and data patterns improve readability, which helps to maintain larger apps.

React Limitations

- Covers only the view layer of the app, hence you still need to choose other technologies to get a complete tooling set for development.
- Uses inline templating and JSX, which might seem awkward to some developers.

Building Blocks of ReactJS

- Components
- Props
- State
- State Lifecycle
- Event handling
- Keys

Components

- The entire application can be modeled as a set of independent components. Different components are used to serve different purposes. This enables us to keep logic and views separate. React renders multiple components simultaneously. Components can be either stateful or stateless.

- Before we start creating components, we need to include a few ‘import’ statements.
- In the first line, we have to instruct JavaScript to import the ‘react’ library from the installed ‘npm’ module. This takes care of all the dependencies needed by React.

import React from 'react';

- The HTML generated by the component needs to be displayed on to the DOM, we achieve this by specifying a render function which tells React where exactly it needs to be rendered (displayed) on the screen. For this, we make a reference to an existing DOM node by passing a container element.
- In React, the DOM is part of the ‘react-dom’ library. So in the next line, we have to instruct JavaScript to import ‘react-dom’ library from the installed npm module.

import ReactDOM from 'react-dom';

Example

- Create a component named ‘MyComponent’ which displays a welcome message.
- Pass the component instance ‘<MyComponent\>’ to React along with its container ‘<div >’ tag.

```
const MyComponent =()=> {      {
        return <h2>Way to go you just created a component!!</h2> ;
        }
}
ReactDOM.render(<MyComponent/>, document.getElementById('root'));
```

Props

- “All the user needs to do is, change the parent component’s state, while the changes are passed down to the child component through props.”
- Props is a shorthand for properties. React uses ‘props’ to pass attributes from ‘parent’ component to ‘child’ component.
- Props are the argument passed to the function or component which is ultimately processed by React.

Example:

```
function Message(props) {
   return <h1>Department of {props.deptt}, School of Computing</h1>;
}
function App() {
   return (
      <div>
         <Message deptt="Computer Science and Engineering" />
         <Message deptt="Information Technology" />
         <Message deptt="Computer Science" />
      </div>
   );
}
ReactDOM.render(<App/>,document.getElementById('root'));
```

State

- State allows us to create components that are dynamic and interactive.
- State is private, it must not be manipulated from the outside.
- It is generally used with data that is bound to change.
- For Example, when we click a toggle button it changes from ‘inactive’ to ‘active’ state.
- State is used only when needed, make sure it is used in render() otherwise don’t include it in the state. We do not use ‘state’ with static components.
- The state can only be set inside the constructor.
- Binding is needed explicitly, as by default the event is not bound to the component.

```
class Toggle extends React.Component {
    constructor(value) {
            super(value);
            this.state = {isToggleOn: true};
            this.handleClick = this.handleClick.bind(this);
    } }
```

Event Handling and State Manipulation

Whenever an event such as a button click or a mouse hover occurs, we need to handle these events and perform the appropriate actions. This is done using event handlers.

While State is set only once inside the constructor it can however, be manipulated through “setState” command. Whenever we call “handleclick” function based on the previous state, “isToggleOn” function is switched between “active” and “inactive” state.

The OnClick attribute specifies the function to be executed when the target element is clicked. In the example, whenever “onclick” is selected it is required to transfer control to handleClick() which switches between the two states.

```
    handleClick() {
            this.setState(prevState =>({
                    isToggleOn: !prevState.isToggleOn
            }));
    }
    render() {
            return(
                    <button onClick={this.handleClick}>
                    {this.state.isToggleOn ? 'ON': 'OFF'} );
    }
```

State Lifecycle

- It is required to initialize resources to components according to their requirements. This is called “mounting” in React.
- It is critical to clear these resources taken by the components when they are destroyed.
- This is because performance can be managed and unused resources can be destroyed. This is called “unmounting” in React.

- It is not essential to use state lifecycle methods, but use them if you wish to take control of the complete resource allocations and retrieval process.
- State lifecycle methods component DidMount() and componentWillUnmount() are used to allocate and release resources respectively.

```
Class Time extends React.component {
        constructor(value) {
                super(value);
                this.state = {date: new Date()};
        }
```

Create an object called Timer ID and set an interval of 2 seconds. Now, this is the time interval based on which the page is refreshed.

```
componentDidMount() {
        this.timerID = setInterval( () => this.tick(),2000);
}
```

Here the interval is the timeframe after which the resources are cleared and the component should be destroyed. Performing such manipulations on the dataset using 'state' can be viewed as an optimal approach.

```
        componentWillUnmount() {clear interval(this.timerID);}
```

A timer is set to call tick() method once every two seconds. An object with current Date is passed to set state. Each time React calls the render() method, this.state.date value is different. React then displays the updated time on the screen.

```
tick(){this.setState({date:new Date()});}
render() {
        return (
                <div>
                        <h2>The Time is {this.state.date.toLocaleTimeString()}.</h2>
                </div>);
        }
        ReactDOM.render( <Time />, document.getElementById('root') );
}
```

Keys

- Keys in React provide identity to components.
- Keys are the means by which React identifies components uniquely.
- For individual components keys are not required as react takes care of key assignment according to their rendering order.
- However, a strategy is required to differentiate between thousands of elements in a list. It is required to assign the elements 'keys'.
- To access the last component in a list using keys, the time required to traverse the entire list sequentially is saved.
- Keys serve to keep track of which items have been manipulated.

- Keys should be given to the elements inside the array to give the elements a stable identity.

Example:
- Create an array ‘Data’ with four items,
- Assign each item the index ‘i’ as the key.
- By defining the key as a property(‘Prop’)
- Use the JavaScript ‘map’ function to pass the key on each element of the array and return the result to the ‘content’ component.

```
class App extends React.Component {
constructor() {
super();
this.state = { data: [
        {item: 'Java', id: '1'},
        {item: 'React', id: '2'},
        {item: 'Python',id: '3'},
        {item: 'C#', id: '4'} ] }
render() {
      return (
                <div>
                        <div>
                                {this.state.data.map((dynamicComponent, i) =>
                                        <Content    key    =    {i}    componentData    =
                                {dynamicComponent}/>)}
            </div>
         </div>
      );
   }
}
class Content extends React.Component {
   render() {
      return (
                        <div>
                                <div>{this.props.componentData.component}</div>
                                <div>{this.props.componentData.id}</div>
         </div>
      );
   }
}
ReactDOM.render(<App/>,document.getElementById('root'));
```

Software Requirements
I For Beginner
nodejs
https://github.com/facebook/create-react-app

II Detailed Requirements

```
npm init
npm install –g babel
npm install –g babel-cli
npm install webpack –save
npm install webpack-dev-server –save
npm install react –save
npm install react-dom –save
npm install babel-core
npm install babel-loader
npm install babel-preset-react
npm install babel-preset-es2015
npm install webpack-dev-server -g
npm install webpack-cli –D
```

Example 1:

```
npm install –g create-react-app
create-react-app first-react-app
cd first-react-app
npm start
```

In Browser, http://localhost:3000

Example 2:

index.html

```
<!DOCTYPE html>
<html lang = "en">
   <head>
      <title>My First React App</title>
   </head>
   <body>
      <div id = "app"></div>
      <script src = "index.js"></script>
   </body>
</html>
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';
ReactDOM.render(<App />, document.getElementById('app'));
```

webpack.config.js

```
var config = {
   entry: './main.js',
   output: {
      path:'/',
      filename: 'index.js',
   },
   devServer: {
      inline: true,
      port: 8080
   },
   module: {
      rules: [
         {
            test: /\.jsx?$/,
            exclude: /node_modules/,
            loader: 'babel-loader',
            query: {
               presets: ['es2015', 'react']
            }
         }
      ]
   }
}
module.exports = config;
```

App.jsx

```
import React from 'react';
class App extends React.Component {
   render() {
      return (
         <div>
            Hello World!!!
         </div>
      );
   }
}
export default App;
```

In package.json, add the following statement under scripts.

```
  "scripts": {
    "start": "webpack-dev-server --hot"
  },
```

```
npm start
```

In browser, launch the app as **localhost:8080**

Example 2: /src/App.js

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';
import PropTypes from 'prop-types';
import logo from './logo.svg';
import './App.css';

class App extends Component {
           constructor(props) {
      super(props);

      this.state = {
        header: "Header from state...",
        content: "Content from state..."
      }

            this.test = {
                    data: []
            }
            this.setTestHandler = this.setTestHandler.bind(this);
            this.forceUpdateHandler = this.forceUpdateHandler.bind(this);
            this.findDomNodeHandler = this.findDomNodeHandler.bind(this);
    }
    forceUpdateHandler() {
      this.forceUpdate();
    };
    findDomNodeHandler() {
      var myDiv = document.getElementById('myDiv');
      ReactDOM.findDOMNode(myDiv).style.color = 'green';
    }
    setTestHandler() {
      var item = "setTest..."
      var myArray = this.test.data.slice();
            myArray.push(item);
            this.test.data=myArray;
      this.setState({data: myArray})
    };
   render() {
            var i=1;
            var mystyle= {
                    fontSize: 100,
                    color: '#FF0000'
            }
     return (
      <div className="App">
           {//Single Line Comment...
           }
                    {/* Multi line Comment */}
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1 className="App-title">Welcome to React</h1>
        </header>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save to reload.
        </p>
                  <p> Welcome to ReactJS</p>
                  <h1>{1+1}</h1>
                  <h1>{i == 1 ? 'True!' : 'False'}</h1>
                  <h1 style={mystyle}>User defined Style</h1>
                          {/* Properties */}
                  <h1>{this.state.header}</h1>
        <h2>{this.state.content}</h2>
```

```
                    {/* Properties with Validation */}
               <h1> Hello, {this.props.name} </h1>
          <h3>Array: {this.props.propArray}</h3>
          <h3>Bool: {this.props.propBool ? "True..." : "False..."}</h3>
          <h3>Func: {this.props.propFunc(3)}</h3>
          <h3>Number: {this.props.propNumber}</h3>
          <h3>String: {this.props.propString}</h3>
               <button onClick = {this.setTestHandler}>SET STATE</button>
          <h4>State Array: {this.test.data}</h4>
               <button onClick = {this.forceUpdateHandler}>FORCE
UPDATE</button>
          <h4>Random number: {Math.random()}</h4>
               <button onClick = {this.findDomNodeHandler}>FIND DOM
NODE</button>
          <div id = "myDiv">NODE</div>
        </div>
      );
  }
}
App.propTypes = {
  name: PropTypes.string,
  propArray: PropTypes.array.isRequired,
  propBool: PropTypes.bool.isRequired,
  propFunc: PropTypes.func,
  propNumber: PropTypes.number,
  propString: PropTypes.string,
};
App.defaultProps = {
  name: 'CSE',
  propArray: [1, 2, 3, 4, 5],
  propBool: true,
  propFunc: function(e) {
    return e
  },
  propNumber: 1,
  propString: "String value..."
}
export default App;
```

Component Life Cycle Methods

- **componentWillMount** is executed before rendering, on both the server and the client side.
- **componentDidMount** is executed after the first render only on the client side. This is where AJAX requests and DOM or state updates should occur. This method is also used for integration with other JavaScript frameworks and any functions with delayed execution such as **setTimeout** or **setInterval**. We are using it to update the state so we can trigger the other lifecycle methods.

- **componentWillReceiveProps** is invoked as soon as the props are updated before another render is called. We triggered it from **setNewNumber** when we updated the state.
- **shouldComponentUpdate** should return **true** or **false** value. This will determine if the component will be updated or not. This is set to **true** by default. If you are sure that the component doesn't need to render after **state** or **props** are updated, you can return **false** value.
- **componentWillUpdate** is called just before rendering.
- **componentDidUpdate** is called just after rendering.
- **componentWillUnmount** is called after the component is unmounted from the dom. We are unmounting our component in **main.js**.

Example
index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import registerServiceWorker from './registerServiceWorker';
ReactDOM.render(<App />, document.getElementById('root'));
registerServiceWorker();
setTimeout(() => {
  ReactDOM.unmountComponentAtNode(document.getElementById('root'));},
10000);
```

App.js

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
          constructor(props) {
    super(props);
    this.state = {
        data: 0
    }
      this.setNewNumber = this.setNewNumber.bind(this)
    };
    setNewNumber() {
      this.setState({data: this.state.data + 1})
    }
  render() {
    return (
        <div>
            <button onClick = {this.setNewNumber}>INCREMENT</button>
            <Content myNumber = {this.state.data}></Content>
        </div>
    );
  }
}
class Content extends React.Component {
    componentWillMount() {
      console.log('Component WILL MOUNT!')
    }
    componentDidMount() {
      console.log('Component DID MOUNT!')
    }
    componentWillReceiveProps(newProps) {
      console.log('Component WILL RECEIVE PROPS!')
    }
    shouldComponentUpdate(newProps, newState) {
      return true;
    }
    componentWillUpdate(nextProps, nextState) {
      console.log('Component WILL UPDATE!');
    }
    componentDidUpdate(prevProps, prevState) {
      console.log('Component DID UPDATE!')
    }
    componentWillUnmount() {
      console.log('Component WILL UNMOUNT!')
    }
    render() {
      return (
        <div>
            <h3>{this.props.myNumber}</h3>
        </div>
      );
    }
}
export default App;
```

Forms using ReactJS
To set an input form with value = {this.state.data}.
This allows to update the state whenever the input value changes.
onChange event that will watch the input changes and update the state accordingly.

Example: App.js

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
        constructor(props) {
                super(props);
                this.state = {
                        data: 'Initial data...'
                }
                this.updateState = this.updateState.bind(this);
        };
   updateState(e) {
    this.setState({data: e.target.value});
   }
   render() {
                return (
                        <div><br/>
                                <input type = "text" value = {this.state.data}
                                  onChange = {this.updateState} />
                                <h4>{this.state.data}</h4>
                        </div>
                        );
                }
        }
export default App;
```

Keys:

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
        constructor(props) {
                super(props);
                this.state = {
                        keydata:[
                                {component: 'First...',  id: 1 },
                                {component: 'Second...',  id: 2 },
                                {component: 'Third...',  id: 3 }      ]
                }
                this.updateState = this.updateState.bind(this);
        };
    updateState(e) {
      this.setState({data: e.target.value});
    }
    render() {
                    return (
                            <div><br/>
                                    <input type = "text" value = {this.state.data}
                                      onChange = {this.updateState} />
                                    <h4>{this.state.data}</h4>
                                     <div>
            {this.state.keydata.map((dynamicComponent, i) => <Content
               key = {i} componentData = {dynamicComponent}/>)}
          </div>
</div>
                            );
                    }
            }
class Content extends React.Component {
   render() {
      return (
         <div>
            <div>{this.props.componentData.component}</div>
            <div>{this.props.componentData.id}</div>
         </div>
      );
   }
}
export default App;
```

Router

Install the library:
npm install react-router-dom

Home.js

```
import React, { Component } from 'react';
class Home extends Component {
   render() {
      return (
         <div>
            <h2>Home</h2>
         </div>
      );
   }
}
export default Home;
```

Login.js

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';
class Login extends Component {
   render() {
      return (
         <div>
            <h2>Login</h2>
         </div>
      );
   }
}
export default Login;
```

App.js

```
import React, { Component } from 'react';
import { BrowserRouter as Router, Switch, Route, Link } from 'react-router-dom';
import Home from './Home';
import Login from './Login';
import './App.css';
class App extends Component {
    render() {
                  return (
                           <Router>
```

```
                <div>
            <h2>Welcome to React Router</h2>
            <ul>
               <li><Link to={'/'}>Home</Link></li>
               <li><Link to={'/Login'}>Login</Link></li>
            </ul>
            <hr />

            <Switch>
               <Route exact path='/' component={Home} />
               <Route exact path='/Login' component={Login} />
            </Switch>
         </div>
      </Router>
        </div>
                          );
                 }
        }
export default App;
```