

RAGHU ENGINEERING COLLEGE

(Autonomous)

(Approved by AICTE, New Delhi, Permanently Affiliated to JNTU Kakinada,
Accredited by NBA & Accredited by NAAC with A grade)

DAKAMARRI (V), BHEEMILI (M), VISAKHAPATNAM, ANDHRA PRADESH-531162.



WEB DEVELOPMENT

Skill Advanced Course

20CS5204

AR20- B.Tech (Common to CSM and CSD Specialization)

III-B. Tech. I-Semester

LABORATORY MANUAL

Prepared By:

Dr. B S Panda

Professor

Department of CSE

Email: panda.bs@raghuenggcollege.in

EXPERMIEMNT LIST

Exercise 1- HTML (Basic Tags), lists, tables

- a) Create web page using basic html tags.
- b) Create an html page which describe the usage of different types of lists

Exercise 2- Table, Creating forms

- a) Design a webpage to list a table of contents at left side of web page, when user click on any content at left side its change should appear on right side of that web page (navigation done at right side).
- b) Creation of forms using HTML, for example design the static web pages required for Sign in and Signup form using HTML. Create a “Signup form” with the following fields
 - I. Name (Text field)
 - II. Password (password field)
 - III. E-mail id (text field)
 - IV. Phone number (text field)
 - V. Sex (radio button)
 - VI. Date of birth (3 select boxes)
 - VII. Languages known (check boxes – English, Telugu, Hindi, Tamil)
 - VIII. Address (text area)

Exercise 3- CSS types

- a) Design a web page using Inline Style Sheet.
- b) Design a web page using Internal Style Sheet
- c) Design a web page using External Style Sheet.

Exercise 4- class, id, and different types of styles in CSS

- a) Design a web page using CSS and use class selector and id observe the difference
- b) Design a webpage using CSS background, border, margins, padding etc.,
- c) Design a web page using Bootstrap.

Exercise 5- Types of Java Scripts and form validations

- a) Create a webpage embedding java script (Both internal and external) in html.
- b) Design a sign-in form and validate its field by using java script.

Exercise 6- TypeScript, variables, arrow functions, classes and inheritance

- a) Write a simple typescript program which describes the different types of variables.
- b) Write a typescript which describes arrow functions.
- c) Write a typescript program for creation of a class and also perform simple inheritance.

Exercise 7- Introduction to Angular, Data binding and built-in directives

- a) Design a sample web page using Angular, and prints “Hello Angular” message.
- b) Design a sample web page using Angular, which illustrates the usage of Data binding, such as Interpolation, property binding, event binding and two-way binding.
- c) Design an angular application which involves Built-In Directives
 - ngIf
 - ngFor
 - ngSwitch

Exercise 8- Components, Nested components, Pipes, Dependency Injection

- a) Split an angular application using components to communicate from parent to child component.
- b) Design an angular application which tells the usage of pipes.
- c) Design a simple angular application which describes the usage of Dependency Injection.

Exercise 9- Services, HTTP Service

- a) How angular services are reusable classes which can be injected in components when it's needed, give an example.
- b) Design an angular application which describes HTTP service.

Exercise 10- Routing, Angular forms

- a) Implementation of Routing, parameterized routing in Angular.
- b) Write an angular code to describe angular forms and perform built-in validations.

Exercise 11- Introduction to NodeJS, Built-in modules (http, fs, events modules)

- a) **Create NodeJS applications, which depicts how client server communication done using http module.**
- b) **Create NodeJS programs which perform file operations using fs module.**
- c) Write a NodeJS application to perform custom events using events module.

Exercise 12- CRUD operations in mongoDB with Node

Write a program in NodeJS to check whether the database has been created successfully or not in mongoDB?

Write a program in NodeJS to insert document in mongoDB?

Write a program in NodeJS to update document in mongoDB?

Write a program in NodeJS to delete single document in mongoDB?

Write a program in NodeJS to display multiple documents in mongoDB?

Course Outcomes:

By the end of this lab, the student is able to

- Designing static web pages by applying different styles.
- Apply dynamic validations on designed web pages using java script.
- Design a single-page applications (SPA) using Angular.
- Know different modules in Node and handling client requests by the Node.

TEXT BOOK AND REFERENCES:

1. HTML5 Black Book.
2. Full Stack Java Script Development with MEAN By Colin J. IHRIG & ADAM BRETZ.

Exercise 1- HTML (Basic Tags), lists, tables

- a) Create web page using basic html tags.
- b) Create an html page which describe the usage of different types of lists

a) Example1: Basic html tags:

```
<html>
<head>
<title> my first html program </title>
</head>
<body bgcolor="red">
<h1> <center> this is my details </center></h1>
<hr>
<b> Dr. B S Panda </b><br>
<u> Dept of CSE </u><br>
<i> REC </i>
</body>
</html>
```

a) Example2: basic image and hyperlink tags

```
<html>
<head>
<title> Add image & Link </title>
</head>
<body bgcolor = "#985CCDD">
<h1> <center> MY IMAGE </center> </h1>
<hr color ="red">
<center><img src = "RE0127.jpg" border ="7" height = "150" width = "150">
<h4> <center> add a hyper link </center></h4>
<a href = "logo.png"> Raghulogo </a><br>
<a href = "abc.pdf"> my file </a>
</center>
</body>
</html>
```

b) Example3: usage of different type of lists:

```
<html>
<head>
<title> add list in a page </title>
</head>
<body bgcolor = "cyan">
<h1> MY LIST </h1>
<marquee scrollldelay = "100" bgcolor= "yellow"> ORDER LIST </marquee>
<hr color ="black">
<ol type ="a">
<li> CSE </li>
<li> CSM </li>
<li> CSO </li>
<li> CSD </li>
</ol>
<marquee scrollldelay = "50" bgcolor = "red" behavior= "alternate"> UNORDER LIST </marquee>
<hr color ="cyan">
<ul type ="circle">
<li> CSE </li>
<li> CSM </li>
<li> CSO </li>
<li> CSD </li>
</ul>
</body>
</html>
```

Exercise 2:**a) Example Design a table:**

```

<html>
<head>
<title>Time table</title>
</head>
<body> <h1 align="center">TimeTable</h1>
<hr> <br>
<table align="center" width="100" height="50" cellspacing="5"
cellpadding="5" border="3" bgcolor="#EEF3D3" bordercolor="red">
<tr> <th>III</th> <th>9:30 -10:20</th> <th>10:20 - 11:10</th>
<th>11:10 - 12:00</th> <th>12:00 - 1:00</th> <th>1:00 - 1:50</th>
<th>1:50 - 2:40</th>
</tr> <tr> <td>MON</td> <td colspan="2" align="center">DAA</td>
<td>ML</td> <td rowspan="6">LUNCH BREAK</td> <td>SKILL</td>
<td bgcolor="cyan">LIB</td>
</tr> <tr> <td>TUE</td> <td colspan="2" align="center">CN</td>
<td bgcolor="skyblue">CRT/ML(T)</td> <td colspan="2"
align="center">SKILL LAB</td>
</tr> <tr> <td>WED</td> <td colspan="2" align="center">SKILL</td>
<td bgcolor="skyblue">CRT/DAA(T)</td> <td colspan="2" align="center">DAA LAB</td>
</tr> <tr> <td>THU</td> <td colspan="2" align="center">CN</td> <td>SPORTS</td> <td colspan="2"
="2">DBMS LAB</td>
</table>
</body>
</html>

```

Output:

TimeTable

III	9:30 -10:20	10:20 - 11:10	11:10 - 12:00	12:00 - 1:00	1:00 - 1:50
MON	DAA	ML	LUNCH BREAK	SKILL	LIB
TUE	CN	CRT/ML(T)		SKILL LAB	
WED	SKILL	CRT/DAA(T)		DAA LAB	
THU	CN	SPORTS		DBMS LAB	

b) Example Design a sign-up form:

```
<html>
<title>Registraion Page</title>
<form action="submit.htm">
<center>
<table border=1 bgcolor="yellow" width = "350" height = "350">
    <caption><h1>Registration Form</h1></caption>
    <tr><td>Name : </td><td><Input type=text name=txtname size=20></td></tr>
    <tr><td>Password :</td><td><Input type=password name=txtpwd size=10></td></tr>
    <tr><td>Email : </td><td><Input type=text name=txtemail size=20></td></tr>
    <tr><td>Phone number : </td><td><Input type=text name=txtphone size=12></td></tr>
    <tr><td>Sex : </td><td><Input type=radio name=radsex value=M>Male <Input type=radio
name=radsex value=F>Female</td></tr>
    <tr><td>Date of Birth : </td><td><select name=day>
        <option value="10">1</option>
        <option value="22">2</option>
        <option value="30">2</option>
    </select>
    <select name=month>
        <option value="Jan">Jan</option>
        <option value="Feb">Feb</option>
        <option value="Mar">Mar</option>
    </select>
    <select name=Year>
        <option value="2002">2002</option>
        <option value="2020">2020</option>
        <option value="2024">2024</option>
    </select> </td></tr>
    <tr><td>Languages Known : </td><td><Input type=checkbox name=telugu value=telugu>Telugu
<Input type=checkbox name=Hindi value=Hindi>Hindi <Input type=checkbox name=English
value=English>English</td></tr>
    <tr><td><Input type=Submit name=btnsubmit value=Submit></td><td><Input type=Reset
name=btnreset value=Reset></td></tr>
</table>
</center>
</form>
</html>
```

Output:

Registration Forn

Name :	<input type="text"/>
Password :	<input type="password"/>
Email :	<input type="text"/>
Phone number :	<input type="text"/>
Sex :	<input checked="" type="radio"/> Male <input type="radio"/> Female
Date of Birth :	<input type="text" value="1"/> <input type="text" value="Jan"/> <input type="text" value="2020"/>
Languages Known :	<input type="checkbox"/> Telugu <input type="checkbox"/> Hindi <input checked="" type="checkbox"/>

Exercise 3: CSS Types

Example-a: Inline Style Sheet

An inline CSS is used to apply a unique style to a single HTML element.
An inline CSS uses the **style** attribute of an HTML element.

```
<html>
<body>
<h1 style="color:blue;">INLINE CSS</h1>
<p style="color:red;">Dear Students Welcome to CSS </p>
</body>
</html>
```

Example-b: Internal Style sheet

An internal CSS is used to define a style for a single HTML page.
An internal CSS is defined in the **<head>** section of an HTML page, within a **<style>** element.

```
<html>
<head>
<style>
body {background-color: cyan;}
    h1 {color: blue;}
    p {color: red;}
</style>
</head>
<body>

<h1>Internal Style Sheet</h1>
<p>This is a paragraph.</p>

</body>
</html>
```

Example-c: External Style sheet

An external style sheet is used to define the style for many HTML pages.

To use an external style sheet, add a link to it in the **<head>** section of each HTML page

```
<html>
<head>
<link rel="stylesheet" href="styles.css">
</head>
<body>

<h1>External Style Sheet</h1>
<p>This is a paragraph.</p>

</body>
</html>
```

Style.css

```
body {  
  background-color: pink;  
}  
h1 {  
  color: green;  
}  
p {  
  color: red;  
}
```

Exercise 4

Example-a: CSS SELECTORS CLASS & ID

The id selector uses the id attribute of an HTML element to select a specific element.

The id of an element is unique within a page, so the id selector is used to select one unique element!

To select an element with a specific id, write a hash (#) character, followed by the id of the element.

Example

```
<!DOCTYPE html>  
<html>  
<head>  
<style>  
#bsp {  
  text-align: center;  
  color: red;  
}  
</style>  
</head>  
<body>  
  
<p id="bsp">Hello World!</p>  
<p>This paragraph is not affected by the style.</p>  
  
</body>  
</html>
```

The CSS class Selector

The class selector selects HTML elements with a specific class attribute.

To select elements with a specific class, write a period (.) character, followed by the class name.

Example

```
<!DOCTYPE html>  
<html>  
<head>  
<style>  
.center {  
  text-align: center;  
  color: red; }  
</style>  
</head>  
<body>  
  
<p class="center">Hello World!</p>  
<p>This paragraph is not affected by the style.</p>  
  
</body>  
</html>
```



```

</style>
</head>
<body>

<h1 class="center">Red and center-aligned heading</h1>
<p class="center">Red and center-aligned paragraph.</p>

</body>
</html>

```

Example-b: CSS border / padding /margin

```

<html>
<head>
<style>
h1 {
  border: 3px solid blue;
  padding: 20px;
}
p {
  border: 4px solid red;
  padding: 30px;
}
b{
  border: 2px solid green;
  margin: 50px;
}

</style>
</head>
<body>

<h1>This is a heading</h1>

<p>This is a first paragraph.</p>
<p>This is a second paragraph.</p>

<b>This is a text.</b>
<b> welcome to margin & padding of CSS </b>

</body>
</html>

```

Output

This is a heading

This is a first paragraph.

This is a second paragraph.

Example-c: Bootstrap

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/css/bootstrap.min.css">
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/js/bootstrap.min.js"></script>
</head>
<body>

<div class="container">
  <h2>Dropdowns</h2>

  <div class="dropdown">
    <button class="btn btn-primary dropdown-toggle" type="button" data-
toggle="dropdown">Dropdown Example
    <span class="caret"></span></button>
    <ul class="dropdown-menu">
      <li><a href="#">HTML</a></li>
      <li><a href="#">CSS</a></li>
      <li><a href="#">JavaScript</a></li>
    </ul>
  </div>
</div>

</body>
</html>
```

output:



Exercise 5 – Types of Java Scripts and form validations

a) Create a webpage embedding java script in html

Example-a:

```
<html>
<head>
<title> case change using JS </title>
<body bgcolor='orange'>
<script type = "text/javascript">
var str =prompt("enter string ");
document.write(str.toUpperCase());
document.write("<br>",str.toLowerCase());
</script>
</body>
</html>
```

Example-b: Design a sign-in form and validate its field by using java script

```
<html>
<script>
function validateform(){
```

```

var name=document.myform.name.value;
var password=document.myform.password.value;

if (name==null || name==""){
    alert("Name can't be blank");
    return false;
}else if(password.length<6){
    alert("Password must be at least 6 characters long.");
    return false;
}
}
}
</script>
<body>
<h1 style="color:blue; margin-top:130px;text-align: center;"> Form Validation using JS </h1>
<form align = "center" name="myform" method="post" action="abc.jsp" onsubmit="return validateform()" >
Name: <input type="text" name="name"><br/>
<br>Password: <input type="password" name="password"><br/>
<br><input type="submit" value="register">
</form>
</body>
</html>

```

Output:



Exercise 6 - Typescript, variables, arrow functions, classes and inheritance

Example-a: Write a simple typescript program which describes the different types of variables.

```

var nme:string = "BSPanda";
var score1:number = 50;
var score2:number = 42.50
var sum = score1 + score2
console.log("name"+nme)
console.log("first score: "+score1)

```

```
console.log("second score: "+score2)
console.log("sum of the scores: "+sum)
```

The output of the above program is given below –

```
name:BSPanda
first score:50
second score:42.50
sum of the scores:92.50
```

Example-b: Write a typescript which describes arrow functions.

To type out the keyword **function** every time you need to create a function. Instead, you first include the parameters inside the () and then add an arrow => that points to the function body surrounded in { }

Ex1:

```
let sum = (a, b) => {
    return a + b;
};
console.log(sum(20, 30));           //returns 50
```

Arrow function without a parameter

Ex2:

```
let Print = () => console.log("Hello Welcome to TypeScript!");
Print();           //output:      Hello Welcome to TypeScript!
```

Example-c: Write a typescript program for creation of a class and also perform simple inheritance.

Inheritance is an aspect of OOPs languages, which provides the ability of a program to create a new class from an existing class. It is a mechanism which acquires the **properties** and **behaviors** of a class from another class. The class whose members are inherited is called the **base class**, and the class that inherits those members is called the **derived/child/subclass**. In child class, we can override or modify the behaviors of its parent class.

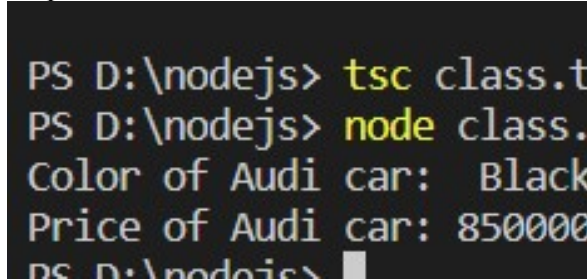
```
class Car {
    Color:string
    constructor(color:string) {
        this.Color = color
    }
}
class Audi extends Car {
    Price: number
    constructor(color: string, price: number) {
        super(color);
        this.Price = price;
    }
}
```

```

display():void {
    console.log("Color of Audi car: " + this.Color);
    console.log("Price of Audi car: " + this.Price);
}
}
let obj = new Audi(" Black", 8500000 );
obj.display();

```

Output



```

PS D:\nodejs> tsc class.t
PS D:\nodejs> node class.
Color of Audi car: Black
Price of Audi car: 8500000
PS D:\nodejs>

```

Exercise 7 – Introduction to Angular, Data binding and built-in directives

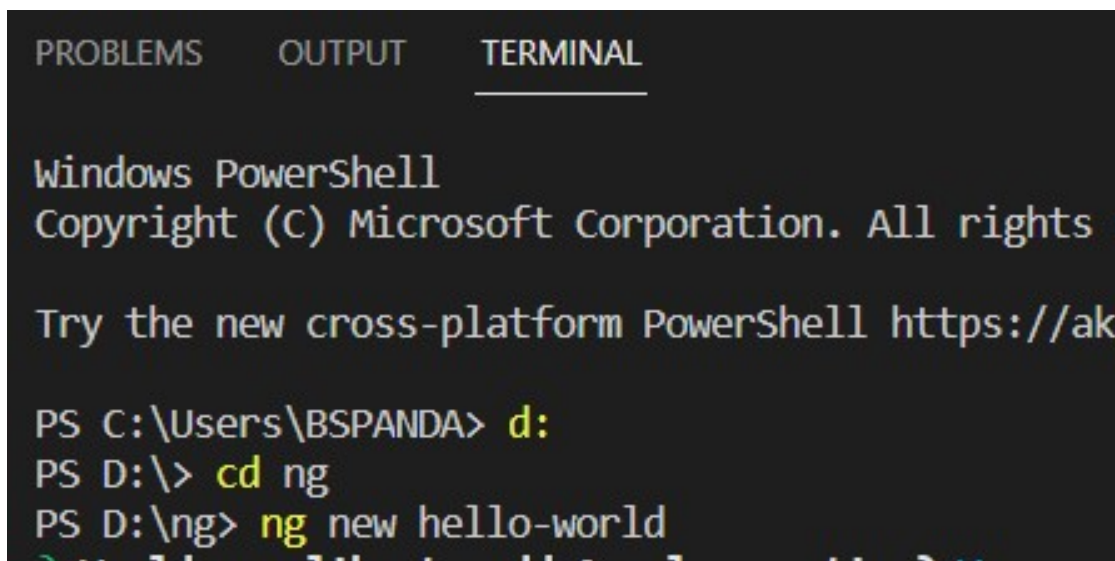
- Design a sample web page using Angular, and print “Hello Angular” message.

Creating the Angular HelloWorld Application

Step 1

Create a folder for your application in the desired location on your system and open it on VSCode. Open a new terminal and type in the following command to create your app folder.

**ng create hello-world or
ng new hello-world**



```

PROBLEMS  OUTPUT  TERMINAL

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/powershell

PS C:\Users\BSPANDA> d:
PS D:\> cd ng
PS D:\ng> ng new hello-world

```

When the command is run, Angular creates a skeleton application under the folder. It also includes a bunch of files and other important necessities for the application.

Step 2

To run the application, change the directory to the folder created, and uses the ng command.

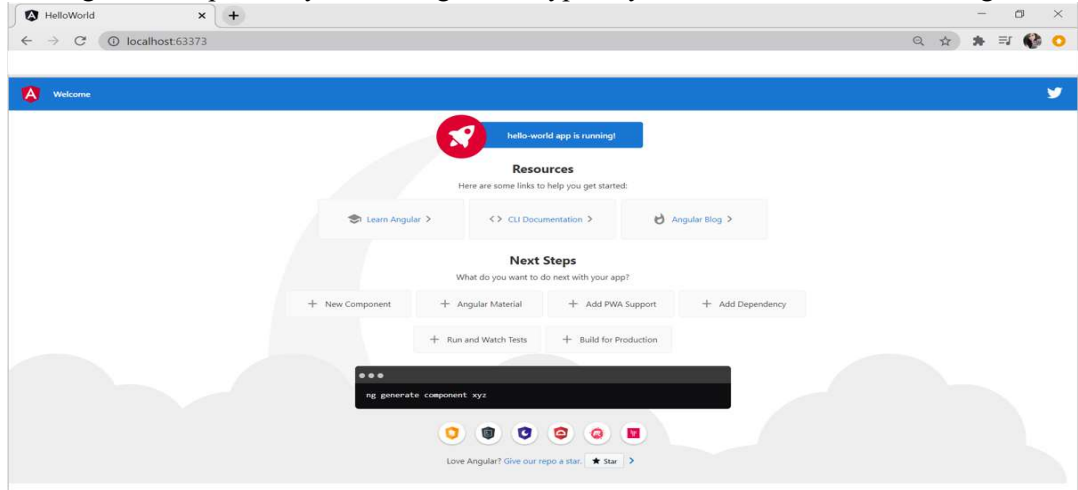
cd hello-world

ng serve

Once run, open your browser and navigate to localhost:4200. If another application is running on that address, you can simply run the command.

ng serve—port

It will generate a port for you to navigate to. Typically, the browser looks something like this



You can leave the ng serve command running in the terminal, and continue making changes. If you have the application opened in your browser, it will automatically refresh each time you save your changes. This makes the development quick and iterative.

Basics of an Angular App

At its core, any Angular application is still a Single-Page Application (SPA), and thus its loading is triggered by a main request to the server. When we open any URL in our browser, the very first request is made to our server. This initial request is satisfied by an HTML page, which then loads the necessary JavaScript files to load Angular as well as our application code and templates.

Root HTML - index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>HelloWorld</title>
  <base href="/">
  <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 root component looks very pristine and neat, with barely any references or dependencies. The only main thing in this file is the **<app-root>** element. This is the marker for loading the

application. All the application code, styles, and inline templates are dynamically injected into the index.html file at run time by the **ng serve** command.

The Entry Point - main.ts

The second important part of our bootstrapping piece is the main.ts file. The index.html file is responsible for deciding which files are to be loaded. The main.ts file, on the other hand, identifies which Angular module is to be loaded when the application starts.

```
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)
.catch(err => console.error(err));
```

Most of the code in the main.ts file is generic, and you will rarely have to touch or change this entry point file. Its main aim is to point the Angular framework at the core module of your application and let it trigger the rest of your application source code from that point.

Main Module - app.module.ts

This is where your application-specific source code starts from. The application module file can be thought of as the core configuration of your application, from loading all the relevant and necessary dependencies, declaring which components will be used within your application, to marking which is the main entry point component of your application.

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

Root Component – AppComponent

The app.component.ts is the actual Angular code that drives the functionality of the application.

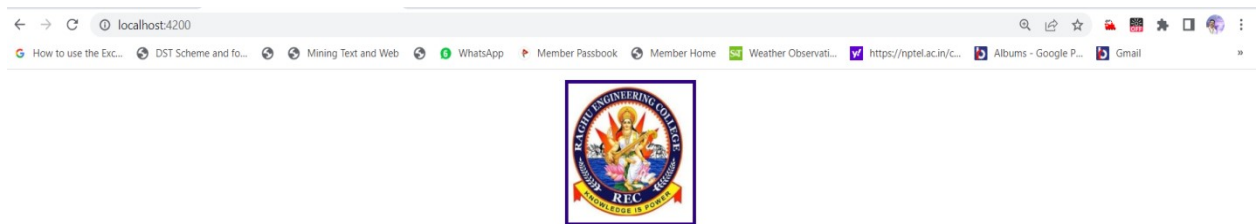
```
import { Component } from '@angular/core';
```

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

Back in our app.component.html file of the Angular HelloWorld application, you can delete all the code and type in something that you wish to render on the browser.

```
<img src= "assets/logo.png" class="center" width="200" height="80" display:block>
<h1>Hello! Students Welcome to the Angular tutorial</h1>
```

The browser looks like this.



Hello! Students Welcome to the Angular tutorial

b) Design a sample web page using Angular, which illustrates the usage of Data binding, such as Interpolation, property binding, event binding and two-way binding.

One-way Data Binding

One-way data binding will bind the data from the component to the view (DOM) or from view to the component. One-way data binding is unidirectional. You can only bind the data from component to the view or from view to the component.

From Component to View

There are different techniques of data binding which use one-way data binding to bind data from component to view. Below are some of the techniques, which uses one-way data binding.

- **Interpolation Binding:** Interpolation refers to binding expressions into marked up language.
- **Property Binding:** Property binding is used to set a property of a view element. The binding sets the property to the value of a template expression.
- **Attribute Binding:** Attribute binding is used to set a attribute property of a view element.
- **Class Binding:** Class binding is used to set a class property of a view element.
- **Style Binding:** Style binding is used to set a style of a view element.

//File Name: app.component.ts

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



```

@Component({
  selector: 'app-root',
  template: `
    <div>
      <strong>{{firstName}}</strong>
      <strong>{{lastName}}</strong>
    </div>
  `
})
export class AppComponent {
  firstName: string = "B S";
  lastName:string = "Panda";
}

```

Two-way Data Binding in Angular

Two-way data binding in Angular will help users to exchange data from the component to view and from view to the component. It will help users to establish communication bi-directionally.

Two-way data binding can be achieved using a ngModel directive in Angular. The below syntax shows the data binding using (ngModel), which is basically the combination of both the square brackets of property binding and parentheses of the event binding.

```
<input type="text" [(ngModel)] = 'val' /> //Html file
```

Before using ngModel to achieve two-way data binding, it's very important to import the FormsModule from @angular/forms in app.module.ts file as shown below. FormsModule will contain the ngModule directive.

```

//Filename app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from "@angular/forms";
import { AppComponent } from './app.component';
import { FormsModule } from "@angular/forms";
@NgModule({
  imports: [BrowserModule, FormsModule],
  declarations: [ AppComponent],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

If you do not import the FormsModule, then you will get Template parse errors and it will result in this error:

"Can't bind to 'ngModel' since it is not a known property of 'input'".

After importing the FormsModule, you can go ahead and bind data using (ngModel) as shown below.

```

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

```

```

selector: 'app-root',
template: `
  Enter the value : <input [(ngModel)] = 'val'>
  <br>
  Entered value is: {{val}}
`,
})
export class AppComponent {
  val: string = '';
}

```

output:



c) Design an angular application which involves Built-in Directives.

Built-in directives

Directives are classes that add additional behavior to elements in your Angular applications. Use Angular's built-in directives to manage forms, lists, styles, and what users see.

The different types of Angular directives are as follows:

DIRECTIVE TYPES	DETAILS
Components	Used with a template. This type of directive is the most common directive type.
Attribute directives	Change the appearance or behavior of an element, component, or another directive.
Structural directives	Change the DOM layout by adding and removing DOM elements.

Built-in structural directives

Structural directives are responsible for HTML layout. They shape or reshape the DOM's structure, typically by adding, removing, and manipulating the host elements to which they are attached.

COMMON BUILT-IN STRUCTURAL DIRECTIVES	DETAILS
NgIf	Conditionally creates of sub views from the template.

COMMON BUILT-IN STRUCTURAL DIRECTIVES	DETAILS
NgFor	Repeat a node for each item in a list.
NgSwitch	A set of directives that switch among alternative views.

Directives preceded with a * like *ngIf, *ngFor, and *ngSwitchCase are structural directives
Structural directives modify the DOM by adding or removing certain elements

The *ngIf structural directive will remove or show an element based on whether the variable it is bound to evaluates to true or false

The *ngFor structural directive will loop over an array of data and create a DOM element for each element in the array, stamping it with the specific values for each array element (if interpolations are being used)

The *ngSwitchCase structural directive will test if the value it is bound to matches the value being switched, and if it matches it will display the element (kind of like having multiple *ngIf directives, but cleaner)

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

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

app.component.spec.ts

import { TestBed } from '@angular/core/testing';
import { RouterTestingModule } from '@angular/router/testing';
import { AppComponent } from './app.component';

describe('AppComponent', () => {
  beforeEach(async () => {
    await TestBed.configureTestingModule({
```

```

    imports: [
      RouterTestingModule
    ],
    declarations: [
      AppComponent
    ],
  }).compileComponents();
});

it('should create the app', () => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.componentInstance;
  expect(app).toBeTruthy();
});

it('should have as title \'ngif\'', () => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.componentInstance;
  expect(app.title).toEqual('ngif');
});

it('should render title', () => {
  const fixture = TestBed.createComponent(AppComponent);
  fixture.detectChanges();
  const compiled = fixture.nativeElement as HTMLElement;
  expect(compiled.querySelector('.content span')?.textContent).toContain('ngif app is running!');
});
});

//Component.ts          example *ngfor

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'angular-project';
  friendslist = [
    { name: 'BSPanda', age: 50 },
    { name: 'Dhoni', age: 45 },
    { name: 'Rohit', age: 36 },
    { name: 'Akash', age: 24 },
    { name: 'Ashish', age: 12 },
  ]
}

```

```
//Component.html
<ul>
  <li *ngFor="let item of friendslist">
    {{ item.name }} is {{ item.age }} years old.
  </li>
</ul>
```

```
//Component.ts          example *ngif
```

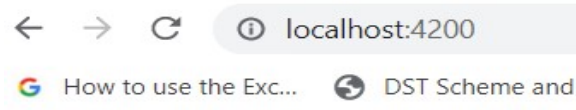
```
<ul *ngIf="isVisible">
  <li *ngFor="let item of friendslist">
    {{ item.name }} is {{ item.age }} years old
  </li>
</ul>
```

```
<button (click)="hideList()">
  Hide list
</button>
```

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

```
isVisible: boolean = true;
```

```
hideList(){
  this.isVisible = false;
}
```



Output:

- BSPanda is 50 years old
- Dhoni is 45 years old
- Rohit is 36 years old
- Akash is 24 years old
- Ashish is 12 years old

```
//*ngswitch app.component.ts file
```

```
class item {
  name: string;
  val: number;
}
export class AppComponent
{
  items: item[] = [{name: 'One', val: 1}, {name: 'Two', val: 2}, {name: 'Three', val: 3}];
  selectedValue: string= 'One';
}
```

```
//the app.component.html file
```

```

<select [(ngModel)]="selectedValue">
  <option *ngFor="let item of items;" [value]="item.name">{{ item.name }}</option>
</select>
<div class='row' [ngSwitch]="selectedValue">
  <div *ngSwitchCase="One">One is Pressed</div>
  <div *ngSwitchCase="Two">Two is Selected</div>
  <div *ngSwitchDefault>Default Option</div>
</div>

```

Exercise-8 – Components, Nested components, Pipes, Dependency Injection

a) Split angular application using components to communicate from parent to child component.

Angular makes the communication between components very easy.
Let's create two components:

1. parent
2. child

In the parent component, declare the property that you want to receive in the child component, say 'ParentId'. While including the child component inside the parent component, bind the 'ParentId' property to the child component using property binding.

//Parentcomponent.ts

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

```
@Component({
  selector: 'app-root',
```

```
  // code for parent component view.
  template: `
    <div style="text-align: center;">
      <h1>
        parent component - {{parentid}}
      </h1>
    </div>`
```

```
  // Bind the ParentId to child component.
```

```
  <child [id] = "parentid"></child>
`,
  styleUrls: []
})
export class AppComponent {
```

```
  // This property is to be sent to the child component.
  parentid: number = 1;
}
```

//Child component.ts

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

```
@Component({
  selector: 'child',
  template: `
```

```

<div style="text-align: center;">
<h2>child component</h2>

// property successfully received from the parent component.
parent id is {{id}}
</div>
`,
styles: []
})
export class TestComponent implements OnInit {

// @input decorator used to fetch the
// property from the parent component.

@Input()
id: number;

ngOnInit(): void {
} }

```

Output:

parent component

child component

parent id is 1

b) Design an angular application which tells the usage of pipes.

- Pipes are defined using the pipe “|” symbol.
- Pipes can be chained with other pipes.
- Pipes can be provided with arguments by using the colon (:) sign.

Some commonly used predefined Angular pipes are:

- DatePipe: Formats a date value.
- UpperCasePipe: Transforms text to uppercase.
- LowerCasePipe: Transforms text to lowercase.
- CurrencyPipe: Transforms a number to the currency string.
- PercentPipe: Transforms a number to the percentage string.
- DecimalPipe: Transforms a number into a decimal point string.

The following component uses a date pipe to display the current date in different formats.

```

@Component({
  selector: 'date-pipe',
  template: `<div>
    <p>Today is {{today | date}}</p>
    <p>Or if you prefer, {{today | date:'fullDate'}}</p>
    <p>The time is {{today | date:'h:mm a z'}}</p>
  `
})

```

```

</div>`
})
// Get the current date and time as a date-time value.
export class DatePipeComponent {
  today: number = Date.now();
}

```

c) Design a simple angular application which describes the usages of Dependency Injection.

Dependency injection is the ability to add the functionality of components at runtime.

```

//app.service.ts
import {
  Injectable
} from '@angular/core';

@Injectable()
export class appService {
  getApp(): string {
    return "Hello world";
  }
}

//app.component.ts
import {
  Component
} from '@angular/core';

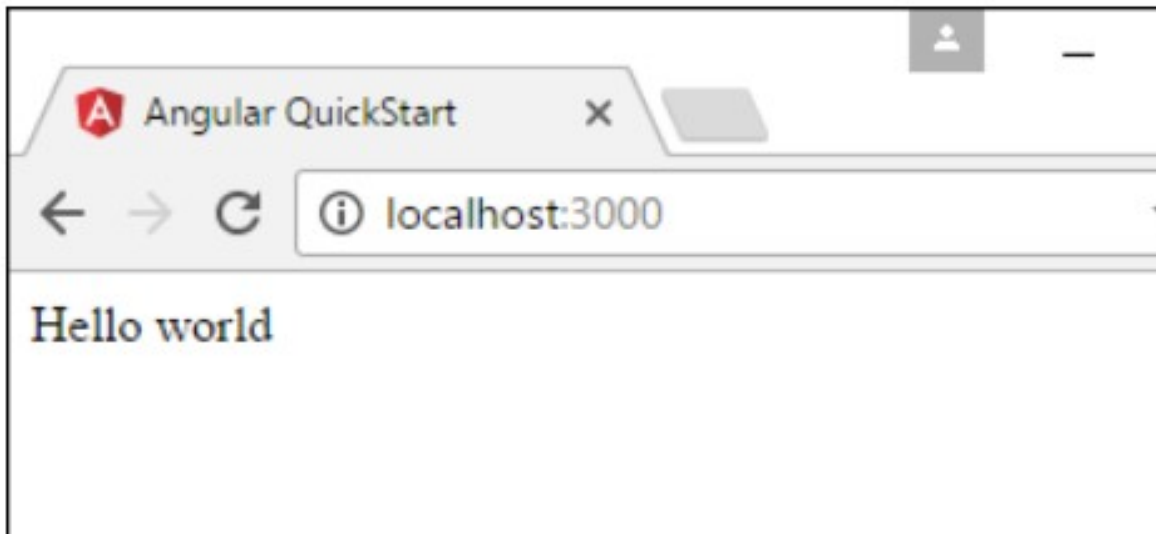
import {
  appService
} from './app.service';

@Component({
  selector: 'my-app',
  template: '<div>{{value}}</div>',
  providers: [appService]
})

export class AppComponent {
  value: string = '';
  constructor(private _appService: appService) { }
  ngOnInit(): void {
    this.value = this._appService.getApp();
  }
}

```

Output:



Exercise 9 – Services, HTTP Service

- a) **How angular services are reusable classes which can be injected in components when it's needed, given an example.**

Service is a piece of reusable code with a focused purpose.

//product.ts

```
export class Product {
  constructor(productID:number,  name: string ,  price:number) {
    this.productID=productID;
    this.name=name;
    this.price=price;
  }
  productID:number ;
  name: string ;
  price:number;
}
```

//product.service.ts

```
import {Product} from './product'
export class ProductService{
  public getProducts() {
    let products:Product[];
    products=[
      new Product(1,'Memory Card',500),
      new Product(1,'Pen Drive',750),
      new Product(1,'Power Bank',100)
    ]
    return products;
  }
}
```

//app.component.ts

```
import { Component } from '@angular/core';
import { ProductService } from './product.service';
```

```

import { Product } from './product';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})
export class AppComponent
{
  products:Product[];
  productService;
  constructor(){
    this.productService=new ProductService();
  }
  getProducts() {
    this.products=this.productService.getProducts();
  } }

```

b) Design an angular application which describes HTTP service.

Http Service will help us fetch external data, post to it, etc. We need to import the http module to make use of the http service.

//app.module.ts

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { HttpClientModule } from '@angular/http';
import { AppComponent } from './app.component';

```

```

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

```

//app.component.ts

```

import { Component } from '@angular/core';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';

```

```

@Component({
  selector: 'app-root',

```

```

    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
  })

  export class AppComponent {
    constructor(private http: Http) { }
    ngOnInit() {
      this.http.get("http://jsonplaceholder.typicode.com/users").
        map((response) => response.json()).
        subscribe((data) => console.log(data))
    }
  }
}

```

Exercise 10 – Routing Angular forms

Implementation of Routing parameterized routing in Angular.

product.component.ts

```

import { Component, OnInit } from '@angular/core';

import { ProductService } from './product.service';
import { Product } from './product';

@Component({
  templateUrl: './product.component.html',
})
export class ProductComponent
{
  products:Product[];

  constructor(private productService:ProductService){
  }
  ngOnInit() {
    this.products=this.productService.getProducts();
  }
}

```

product.component.html

```

<h1>Product List</h1>
<div class='table-responsive'>
  <table class='table'>
    <thead>
      <tr>
        <th>ID</th>
        <th>Name</th>
        <th>Price</th>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor="let product of products;">
        <td>{{product.productID}}</td>
        <td><a [routerLink]='["/product",product.productID]">{{product.name}} </a> </td>
        <td>{{product.price}}</td>
      </tr>
    </tbody>
  </table>
</div>

```

```
</table>
</div>
```

product.service.ts

```
import { Observable } from 'rxjs';
import { Product } from './Product'
export class ProductService {

    public getProducts() {

        let products:Product[];

        products=[
            new Product(1,'Memory Card',500),
            new Product(2,'Pen Drive',750),
            new Product(3,'Power Bank',100)
        ]
        return products;
    }
    public getProduct(id) {
        let products:Product[]=this.getProducts();
        return products.find(p => p.productID==id);
    } }
}
```

product.ts

```
export class Product {
    constructor(productID:number,  name: string ,  price:number) {
        this.productID=productID;
        this.name=name;
        this.price=price;
    }
    productID:number ;
    name: string ;
    price:number;
}
```

product-detail.component.ts

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { Router,ActivatedRoute } from '@angular/router';
import { ProductService } from './product.service';
import { Product } from './product';

@Component({
    templateUrl: './product-detail.component.html',
})
export class ProductDetailComponent
{
    product:Product;
    id;
    constructor(private _ActivatedRoute:ActivatedRoute,
                private _router:Router,
                private _productService:ProductService){
    }
    sub;
```

```

ngOnInit() {

  this.sub=this._ActivatedRoute.paramMap.subscribe(params => {
    console.log(params);
    this.id = params.get('id');
    let products=this._productService.getProducts();
    this.product=products.find(p => p.productID==this.id);
  });
}
ngOnDestroy() {
  this.sub.unsubscribe();
}
onBack(): void {
  this._router.navigate(['product']);
}
}

```

product-detail.component.html

```
<h3>Product Details Page</h3>
```

```

product : {{product.name}}
price : {{ product.price}}
<p>
  <a class='btn btn-default' (click)="onBack()">Back </a>
</p>

```

app.routing.ts

```

import { Routes } from '@angular/router';

import { HomeComponent } from './home.component'
import { ContactComponent } from './contact.component'
import { ProductComponent } from './product.component'
import { ErrorComponent } from './error.component'
import { ProductDetailComponent } from './product-detail.component'
export const appRoutes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'product', component: ProductComponent },
  { path: 'product/:id', component: ProductDetailComponent },
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: '**', component: ErrorComponent }
];

```

app.component.ts

```

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

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

```

app.component.html

```
div class="container">
```

```

<nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-header">
      <a class="navbar-brand" [routerLink]="['/']"><strong> {{title}} </strong></a>
    </div>
    <ul class="nav navbar-nav">
      <li><a [routerLink]="['home']">Home</a></li>
      <li><a [routerLink]="['product']">Product</a></li>
      <li><a [routerLink]="['contact']">Contact us</a></li>
    </ul>
  </div>
</nav>

```

```

<router-outlet></router-outlet>

```

```

</div>

```

app.module.ts

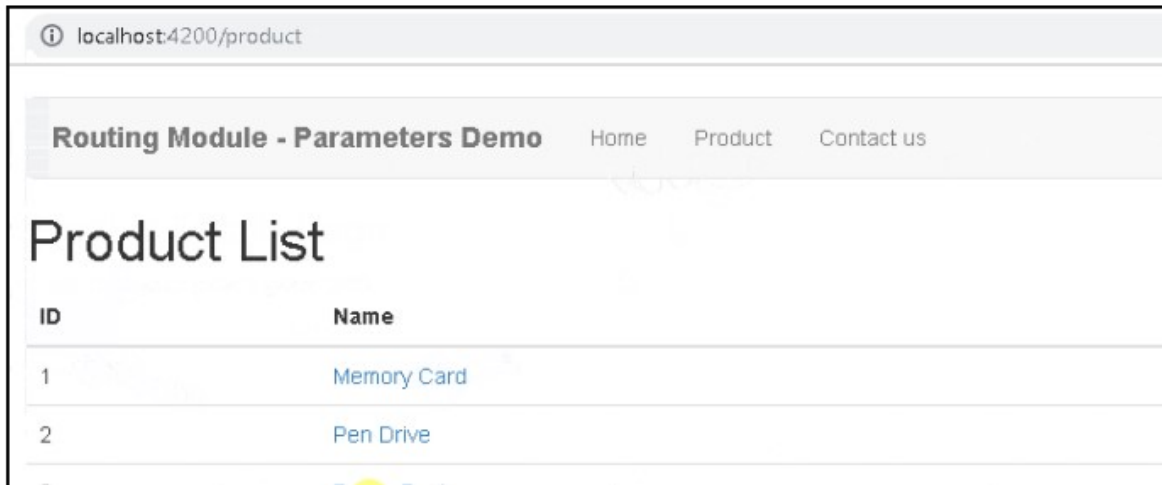
```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/http';
import { RouterModule } from '@angular/router';
import { AppComponent } from './app.component';
import { HomeComponent } from './home.component';
import { ContactComponent } from './contact.component';
import { ProductComponent } from './product.component';
import { ErrorComponent } from './error.component';
import { ProductDetailComponent } from './product-detail.component';
import { ProductService } from './product.service';
import { appRoutes } from './app.routes';

@NgModule({
  declarations: [
    AppComponent, HomeComponent, ContactComponent, ProductComponent, ErrorComponent, ProductDetailComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    RouterModule.forRoot(appRoutes)
  ],
  providers: [ProductService],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Output



Write an angular code to describe angular forms and perform built-in validations.

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

import { AppComponent } from './app.component';
import { ReactiveFormsModule } from '@angular/forms';

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

```
// app.component.ts
import { Component } from '@angular/core';
import { FormGroup, FormBuilder, Validators } from '@angular/forms';

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

```

title = 'Angular Form Validation Tutorial';
angForm: FormGroup;
constructor(private fb: FormBuilder) {
  this.createForm();
}
createForm() {
  this.angForm = this.fb.group({
    name: ['', Validators.required ],
    address: ['', Validators.required ]
  });
}
}
<!-- app.component.html -->

<div class="container">
  <h1>
    Welcome to {{title}}!!
  </h1>
  <form [formGroup]="angForm" novalidate>
    <div class="form-group">
      <label>Name:</label>
      <input class="form-control" formControlName="name" type="text">
    </div>
    <div *ngIf="angForm.controls['name'].invalid && (angForm.controls['name'].dirty ||
angForm.controls['name'].touched)" class="alert alert-danger">
      <div *ngIf="angForm.controls['name'].errors.required">
        Name is required.
      </div>
    </div>
    <div class="form-group">
      <label>Address:</label>
      <input class="form-control" formControlName="address" type="text">
    </div>
    <div *ngIf="angForm.controls['address'].invalid && (angForm.controls['address'].dirty ||
angForm.controls['address'].touched)" class="alert alert-danger">
      <div *ngIf="angForm.controls['address'].errors.required">
        email is required.
      </div>
    </div>
    <button type="submit"
      [disabled]="angForm.pristine || angForm.invalid" class="btn btn-success">
      Save
    </button>
  </form>
  <br />
  <p>Form value: {{ angForm.value | json }}</p>
  <p>Form status: {{ angForm.status | json }}</p>

```



```

</div>

//index.html
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>AngularProject</title>
  <base href="/">
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
beta/css/bootstrap.min.css" integrity="sha384-
/Y6pD6FV/Vv2HJnA6t+vsIU6fwYXjCFtcEpHbNJ0lyAFsXTsjBbfaDjzALeQsN6M"
crossorigin="anonymous">
  <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
<body>
  <app-root>Loading...</app-root>
</body>
</html>

```

Output:

Welcome to Angular Form Validation

Name:

Address:

Exercise 11 – Introduction to NodeJS, Build-in modules (http, fs, eventmodules)

NodeJS:

- Node.js is an open source server environment
- Node.js is free
- Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
- Node.js uses JavaScript on the server

Download Node.js:

- The official Node.js website has installation instructions for Node.js: <https://nodejs.org>
- Node.js has a set of built-in modules which you can use without any further installation. (for ex: fs, http, mongodb...)
- Initiate node js file (consider a file name “demo_http.js”):

C:\Users\Your Name> node demo_http.js

- The Node.js file system module allows you to work with the file system on your computer. To include the File System module, use the `require()` method:
`var fs = require('fs');`

a) Create NodeJS applications, which depicts how client server communication done using http module.

Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).

To include the HTTP module, use the `require()` method:

Code:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('Hello World!');
  res.end();
}).listen(8080);
```

output



b) Create NodeJS programs which perform file operations using fs module.

Common use for the File System module:

- Read files
- Create files
- Update files
- Delete files
- Rename files

Read Files:

The `fs.readFile()` method is used to read files on your computer.

Program: (MyFirstNodeJsApp11.js)

```
fs = require('fs');

fs.readFile('TextFile.txt',function (err, data) {
```

```
        if (err)
            throw err;
        console.log(data.toString());
    });
```

Save the code in a file called "MyFirstNodeJsApp1.js" and run the file as:

```
C:\mypath>node MyFirstNodeJsApp1.js
```

Write file:

```
var fs = require('fs');

fs.writeFile('mynewfile3.txt', 'Hello Nodejs!', function(err) {
    if (err) throw err;
    console.log('Saved!');
});
```

```
PS D:\nodejs> node nodewrite.js
Saved!
```

c) Write nodeJS application to perform custom events using events module.

Node.js has a built-in module, called "Events", where you can create-, fire-, and listen for- your own events.

To include the built-in Events module use the require() method.

Code:

```
var events = require('events');
var EventEmitter = new events.EventEmitter();

//Create an event handler:
var myEventHandler = function () {
    console.log('Welcome to nodeJs events module!');
}

//Assign the eventhandler to an event:
eventEmitter.on('scream', myEventHandler);

//Fire the 'scream' event:
eventEmitter.emit('scream');
```

Exercise 12 – CRUD operations in mongoDB with Node

- Write a program in NodeJS to check whether the database has been create successfully or not in MongoDB?
- Write a program in NodeJS to insert document in mongoDB?
- Write a program in NodeJS to update document in mongoDB?
- Write a program in NodeJS to delete single document in mongoDB?
- Write a program in NodeJS to display multiple documents in mongoDB?

MongoDB:

One of the most popular NoSQL database is MongoDB.

About MongoDB:

MongoDB is an open-source document database and leading NoSQL database. MongoDB is written in C++.

MongoDB is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document.

Database:

Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

Collection:

Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields

Document:

A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

The following table shows the relationship of RDBMS terminology with MongoDB.

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field

Download MongoDB from the following link.

<https://www.mongodb.com/download-center/community>

CRUD operations:**1) The insertOne()/insertMany() Method:**

To insert document/documents into MongoDB collection, you need to use MongoDB's insertOne()/insertMany()

Syntax:

>db.COLLECTION_NAME.insertOne(document)// to insert one document

Ex: db.student.insertOne({sid:101,sname:"abc"});

>db.COLLECTION_NAME.insertMany(array of document)

// to insert many documents

```
Ex: db.student.insertMany([ {sid:102,sname:"def"}, {sid:103,sname:"ghi"},  
    {sid:104,sname:"jkl"}]);
```

2) The findOne()/find()/findAll() Method

To query data from MongoDB collection, you need to use MongoDB's find() method.

Syntax:

```
>db.COLLECTION_NAME.find({condition},{projection of columns})
```

Ex:

```
1) db.student.find();
```

It will show all documents in student collection

```
2) db.student.find({sid:101});
```

It will display documents of sid 101 in student collection

```
3) db.student.find({sid:101},{sname:1})
```

It will display sname of student id 101 in student collection; here sname:1 means it only display sname, if sname is 0 means it could not display sname field remaining fields in student collection will be displayed. Note: in the output, _id field will be displayed by default, if you don't want to display in the output, give _id:0 in the projection of columns.

3) The updateOne()/updateMany() method:

The updateOne()/updateMany() method updates the values in the existing document.

Syntax:

```
>db.COLLECTION_NAME.updateOne(CONDITION, UPDATED_DATA)
```

It update only first document in the resultant collection.

```
>db.COLLECTION_NAME.updateMany(CONDITION, UPDATED_DATA)
```

It updates all documents in the resultant collection.

```
Ex: >db.student.update({sid:102},{ $set:{sname:"xyz"}})
```

It will update sname value to "xyz" of sid 102 in student collection

4) The removeOne()/removeMany() method:

MongoDB's remove() method is used to remove a document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag.

- deletion criteria – (Optional) deletion criteria according to documents will be removed.
- justOne – (Optional) if set to true or 1, then remove only one document.

Syntax:

```
>db.COLLECTION_NAME.removeOne(DELETION_CRITERIA)
```

```
Ex: > db.hello1.remove({"sid":"503"})
```

```
WriteResult({ "nRemoved" : 1 })
```

It removes only one document from set of documents which meet deletion criteria. If we use removeMany() it will remove all documents which meets deletion criteria.

Install MongoDB Driver:

Let us try to access a MongoDB database with Node.js.

To download and install the official MongoDB driver, open the Command Terminal and execute the following:

Download and install mongodb package:

```
C:\users\username>npm install mongodb
```

Now you have downloaded and installed a mongodb database driver.

Node.js can use this module to manipulate MongoDB databases:

```
var mongo = require('mongodb');
```

Creating a Database:

To create a database in MongoDB, start by creating a MongoClient object, and then specify a connection URL with the correct ip address and the name of the database you want to create.

MongoDB will create the database if it does not exist, and make a connection to it.

Save the code in a file called "MongoEx1.js" and run the file as:

```
C:\mypath>node MongoEx1.js
```

Before running our application, start mongo server, by giving the command mongod.

Important: In MongoDB, a database is not created until it gets content!

MongoDB waits until you have created a collection (table), with at least one document (record) before it actually creates the database (and collection).