|  |  |  |  |
| --- | --- | --- | --- |
| **Weeks** | **E.No.** | **Name of the Experiment** | **Page No.** |
| Week1 | 1 | **Creating Environment for Angular JS :**  The step by step installation process to create environment for Angular JS.  **and creating basic applications:**  Using Angular CLI create applications and customization |  |
| **Exercise:**  Create a simple multiplier Angular JS application which will multiply two numbers and display the result. |  |
| Week 2 | 2 | **Creating and Communicating Between Components:**  **Parent to Child: Sharing Data via Input**  **Child to Parent: Sharing Data via ViewChild with AfterViewInit**  **Child to Parent: Sharing Data via Output() and EventEmitter** |  |
| **Exercise:**  create a Website with Custom Components. |  |
| Week 3 | 3 | **Exploring and Applications of Angular Templates**  **Experiment3.1**: Write a program to create template inline into app.component itself using template property of @Component decorator.  **Experiment 3.2**: Write a program to create external template. |  |
| **Exercise:**  **Exercise 3.1:** Write an inline template to display your name and address.  **Exercise 3.2:** Write an external template to display student id and name. |  |
| Week 4 | 4 | **Manipulating Data With Pipes:**  Experiment4.1: IMPLEMENTING ALL DEFAULT PIPES: Uppercase, Lowercase, Date, Currency, Json and Slice.  Experiment 4.2: Create custom PIPES to multiply two numbers. |  |
| **Exercise4.1:**  **Create a custom pipe to find square root.** |  |
| Week5 | 5 | Experiment5:Creating Directives and Advanced Components |  |
| Exercise: |  |
| Week 6 | 6 | Creation and implementation of Templates in JSXExperiment:Write a React JSX application to print number table of 12. |  |
| Exercise:6.1Write HTML on multiple lines, put the HTML inside parentheses. **6.2** Write "Hello" if x is less than 10, otherwise "Goodbye": |  |
| Week 7 | 7 | **Applications building using reactJS and redux.**  **Experiment:**  7.1 Write an redux based react JS application to concept of state using implement increment and decrement counter. |  |
| **Exercise 7.1:** Implement redux to check whether user is logged in or not. |  |
| Week 8 | 8 | **Experiment 8.1:**  **Creating and implementing modules**  Create a USER Module. Use this USER module to check logging credential of a user.  **Experiment 8.2:**  **File systems, Buffers, Stream and Events**   1. Create a file, write few data in the file, append data, read the file and close the file. 2. Allocate buffer for 1024 bytes, write data string into it and print length. 3. Create Write and Read streams. 4. Write two functions “fun1” and “fun2” and register it to event listener. |  |
| **Exercise 8.1:**  **create date module which should return current date in “”YYYY-MM-DD” format.**  **Exercise 8.2:**  **Create a file and write “Mallareddy University” and append ” AIML Department” to the same file.**  **Exercise 8.3:**  **Allocate a buffer of 100 bytes and write “Mallareddy University” into the buffer. Print the length of the buffer.** |  |
| Week 9 | 9 | **Experiment 9:**  **Creating HTTP server and interacting with client using Node JS.**  Create a HTTP server which will forward a HTML page in response of a client request. |  |
| **Exercise: Create a HTTP server which will respond by sending message “server is active”.** |  |
| Week10 | 10 | **Develop Application to store, maintain data using Node JS and database**.  **Experiment 10:** Create contact form to store data email and query using node js express and monodb. |  |
| **Exercise:**  Create contact form to store data name and address using node js express and monodb. |  |
| Week 11 | 11 | **Create, delete, update collection of Mongodb documents. Different approaches to migrate data Mongodb.**  **Experiment 11.1:**  Experiment 11.1: Create database employee. Create a collection in employee database and insert following data. Update salary of “BInay”. Delete the record for “Ajay”.  **Experiment 11.2:**  Experiment 11.2: create a Json file containing above data. And import the data into mongodb database. |  |
| **Exercise 11.1:**  Create database employee. Create a collection in students database and insert following data. Update marks of “Smith”. Delete the record for “Binod”.  **Experiment 11.2:**  create a Json file containing employee data. And import the data into mongodb database. |  |
| Week 12 | 12 | **Experiment 12: Create a student database and using angular JS & node JS collect, update and delete student data**. |  |
| **Exercise 12: Create a employee web application with Angular JS, node JS and mongodb.** |  |

**Week 1**

**Experiment 1: Creating Environment for Angular JS and creating basic applications.**

**Introduction:**

**AngularJS** is a very powerful JavaScript Framework. It is used in Single Page Application (SPA) projects. It extends HTML DOM with additional attributes and makes it more responsive to user actions. AngularJS is open source, completely free, and used by thousands of developers around the world. It is licensed under the Apache license version 2.0.

|  |  |
| --- | --- |
| REQUIREMENTS | DETAILS |
| Node.js | Angular requires an [active LTS or maintenance LTS](https://nodejs.org/about/releases) version of Node.js.  For information about specific version requirements, see the engines key in the [package.json](https://unpkg.com/browse/@angular/core/package.json) file.  For more information on installing Node.js, see [nodejs.org](https://nodejs.org/). If you are unsure what version of Node.js runs on your system, run node -v in a terminal window. |
| npm package manager | Angular, the Angular CLI, and Angular applications depend on [npm packages](https://docs.npmjs.com/getting-started/what-is-npm) for many features and functions. To download and install npm packages, you need an npm package manager. This guide uses the [npm client](https://docs.npmjs.com/cli/install) command line interface, which is installed with Node.js by default. To check that you have the npm client installed, run npm -v in a terminal window. |

**To install Angular on your local system, you need the following:**

**Example and Solution**

Install the Angular CLI

You use the Angular CLI to create projects, generate application and library code, and perform a variety of ongoing development tasks such as testing, bundling, and deployment.

To install the Angular CLI, open a terminal window and run the following command:

content\_copynpm install -g @angular/cli

On Windows client computers, the execution of PowerShell scripts is disabled by default. To allow the execution of PowerShell scripts, which is needed for npm global binaries, you must set the following [execution policy](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_execution_policies):

content\_copySet-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned

Carefully read the message displayed after executing the command and follow the instructions. Make sure you understand the implications of setting an execution policy.

Create a workspace and initial application

You develop apps in the context of an Angular [workspace](https://angular.io/guide/glossary#workspace).

To create a new workspace and initial starter app:

1. Run the CLI command ng new and provide the name my-app, as shown here:

content\_copyng new my-app

1. The ng new command prompts you for information about features to include in the initial app. Accept the defaults by pressing the Enter or Return key.

The Angular CLI installs the necessary Angular npm packages and other dependencies. This can take a few minutes.

The CLI creates a new workspace and a simple Welcome app, ready to run.

Run the application

The Angular CLI includes a server, for you to build and serve your app locally.

1. Navigate to the workspace folder, such as my-app.
2. Run the following command:

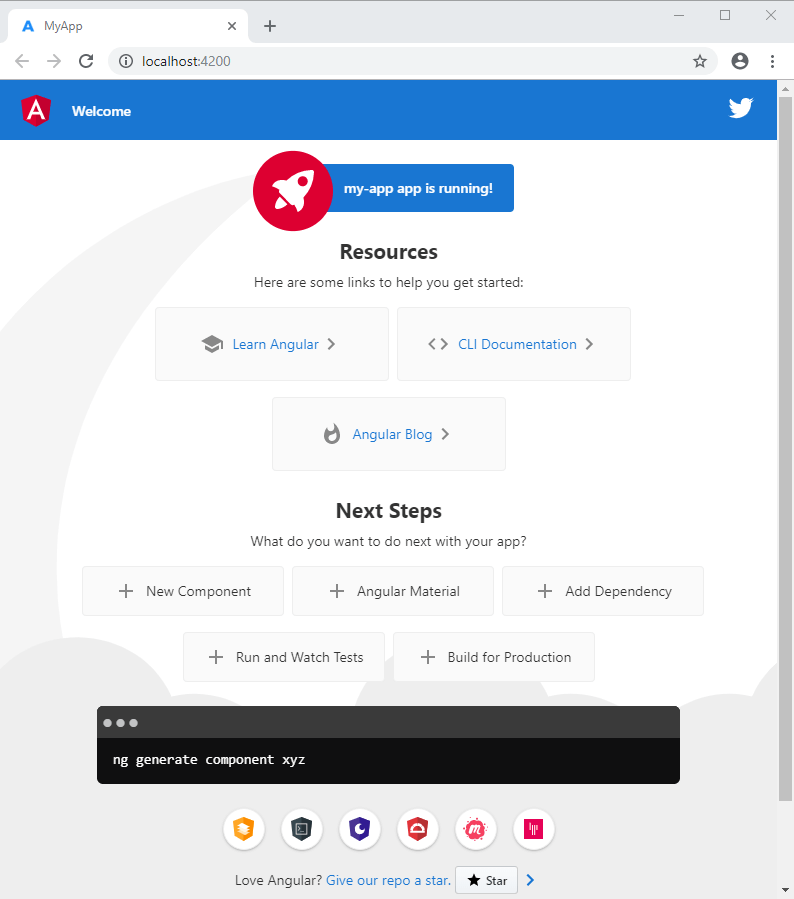
cd my-app

ng serve --open

The ng serve command launches the server, watches your files, and rebuilds the app as you make changes to those files.

The --open (or just -o) option automatically opens your browser to http://localhost:4200/.

If your installation and setup was successful, you should see a page similar to the following.



Before creating actual *Hello World !* application using AngularJS, let us see the parts of a AngularJS application. An AngularJS application consists of following three important parts −

* **ng-app** − This directive defines and links an AngularJS application to HTML.
* **ng-model** − This directive binds the values of AngularJS application data to HTML input controls.
* **ng-bind** − This directive binds the AngularJS Application data to HTML tags.

Creating AngularJS Application

Step 1: Load framework

Being a pure JavaScript framework, it can be added using <Script> tag.

<script

src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js">

</script>

Step 2: Define AngularJS application using ng-app directive

<div ng-app = "">

...

</div>

Step 3: Define a model name using ng-model directive

<p>Enter your Name: <input type = "text" ng-model = "name"></p>

Step 4: Bind the value of above model defined using ng-bind directive

<p>Hello <span ng-bind = "name"></span>!</p>

Executing AngularJS Application

Use the above-mentioned three steps in an HTML page.

testAngularJS.htm

[Live Demo](http://tpcg.io/LCqKRc)

<html>

<head>

<title>AngularJS First Application</title>

</head>

<body>

<h1>Sample Application</h1>

<div ng-app = "">

<p>Enter your Name: <input type = "text" ng-model = "name"></p>

<p>Hello <span ng-bind = "name"></span>!</p>

</div>

<script src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js">

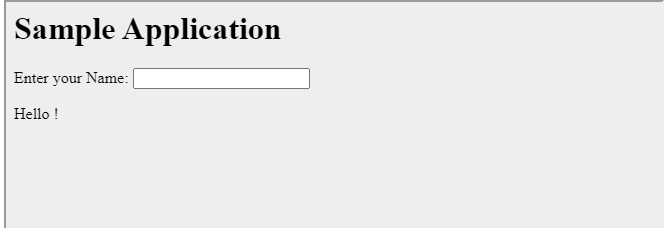
</script>

</body>

</html>

Output

Open the file *testAngularJS.htm* in a web browser. Enter your name and see the result.



**Exercise and Solution**

**Create a simple multiplier Angular JS application which will multiply two numbers and display the result**

.

**Solution :**

The following is the HTML code with AngularJS for the above multiplier example.

Example: First AngularJS Application

<!DOCTYPE html>

<html>

<head>

<title>First AngularJS Application</title>

<script src= "~/Scripts/angular.js"></script>

</head>

<body **ng-app** >

<h1>First AngularJS Application</h1>

Enter Numbers to Multiply:

<input type="text" **ng-model="Num1"** /> x <input type="text" **ng-model="Num2"** />

<span ng-model="Num1 \* Num2">{{Num1 \* Num2}}</span>

</body>

</html>

**Result :**



**Week 2**

**Experiment 2: Creating and Communicating Between Components**

**Introduction**

**What are components in Angular?**

Components are the **basic building blocks** of Angular applications.

In a simpler manner, we can say components are UI/ View elements along with the mechanism to show and operate on data.

The Angular Team defines a component as the **Patch of the scree**n that we **call a View** which declares the **reusable building blocks** for an application.

So, A component is anything which is visible on the screen, which can be reused again and again.

**#Prerequisites**

* Node installed on your machine
* NPM installed on your machine
* Basic Knowledge of JavaScript
* Installing Angular CLI 8 : **npm install -g @angular/cli**
* Creating your Angular 8 Project : **ng new angular-example**
* Creating Components : **ng g c payment-components**
* Run on local Server on http://localhost:4200 : **ng serve**

**3 ways to communicate data between Angular components**

* Parent to Child: Sharing Data via **Input**
* Child to Parent: Sharing Data via **ViewChild** with **AfterViewIni**t
* Child to Parent: Sharing Data via **Output()** and **EventEmitter**

**1. Parent to Child: Sharing Data via Input**

To share data from parent to child component via **@Input** Decorator.

@Input **links a property** of a child component **with a value** that was given by the parent component.

For e.g :

Create 2 components **parent.component.ts** and **child.component.ts**

Let's create a simple string variable in our **parent.component.ts** file, It will contain the message that we want to give to our child.

Add bellow code in **parent.component.ts** file :

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

@Component({

selector: 'app-parent',

template: `

<app-child [childMessage]="parentMessage"></app-child>

styleUrls: ['./parent.component.css']

})

export class ParentComponent{

parentMessage = "message from parent";

constructor() { } }

**parent.component.ts**

**Add bellow code in child.component.ts file :**

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

@Component({

selector: 'app-child',

template: ` Say {{ message } `,

styleUrls: ['./child.component.css']

})

export class ChildComponent {

@Input() childMessage: string;

constructor() { }

}

**child.component.ts**

**2. Child to Parent: Sharing Data via ViewChild with AfterViewInit**

**ViewChild** allows a child component to be injected into a parent component.

It will give the parent access to its attributes and functions.

A Child won’t be available to give access until the **view** has been **initialized.**

This means we need to implement the **AfterViewInit** lifecycle hook to receive the data from the child.

Add bellow code in **parent.component.ts** file :

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

import { ChildComponent } from "../child/child.component";

@Component({

selector: 'app-parent',

template: `

Message: {{ message }}

<app-child></app-child>

`,

styleUrls: ['./parent.component.css']

})

export class ParentComponent implements AfterViewInit {

@ViewChild(ChildComponent) child;

constructor() { }

Message:string;

ngAfterViewInit() {

this.message = this.child.message

}

}

**parent.component.ts**

Add bellow code in **child.component.ts**

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

@Component({

selector: 'app-child',

template: ` `,

styleUrls: ['./child.component.css']

})

export class ChildComponent {

message = 'Hello Angular!';

constructor() { }

}

**child.component.ts**

**3. Child to Parent: Sharing Data via Output() and EventEmitter**

Another way to share data is **to emit data** from the **child component** which can be **listed by** the **parent component.**

This approach is ideal when you want to share data changes that occur on things like **button clicks**, **form entries**, and other user events.

For e.g.

In the parent, we create a function to **receive the message** and set it equal to the **message variable.**

In the child, we declare a **messageEvent** variable with the **Output decorato**r and set it equal to a **new event emitter**.

Then we create a function named **sendMessage** that calls emit on this event with the message we want to send. Lastly, we create a button to **trigger this function**.

The parent can now **subscribe** to this **messageEvent** that’s outputted by the child component, then run the **receive message** function whenever this **event occurs.**

Add bellow code in **parent.component.ts** file :

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

@Component({

selector: 'app-parent',

template: `

Message: {{message}}

<app-child (messageEvent)="receiveMessage($event)"></app-child>

`,

styleUrls: ['./parent.component.css']

})

export class ParentComponent {

constructor() { }

Message:string;

receiveMessage($event) {

this.message = $event

}

}

**parent.component.ts**

Add bellow code in **child.component.ts** file :

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

@Component({

selector: 'app-child',

template: `

<button (click)="sendMessage()">Send Message</button>

`,

styleUrls: ['./child.component.css']

})

export class ChildComponent {

message: string = "Hey Angular!"

@Output() messageEvent = new EventEmitter<string>();

constructor() { }

sendMessage() {

this.messageEvent.emit(this.message)

}

}

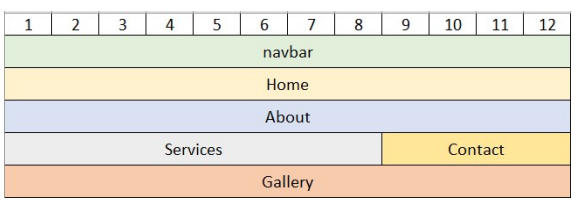
**child.component.ts**

**#Conclusion**

* Here we define how to **share data between component** in Angular
* We can share data from **parent to child** via **@Input**
* We can share data from **child to parent** via **ViewChild**
* We can share data from **child to parent** via **@Output** for **button clicks** or **Form entries**

**Exercise and Solution:**

* 1. **create a Website with Custom Components like**

****

**Solution:**

**Create Custom Components**

ng g c [component name]

Use the following commands to create required Angular Components:

ng g c home ◊ check home.component.ts for selector: app-home

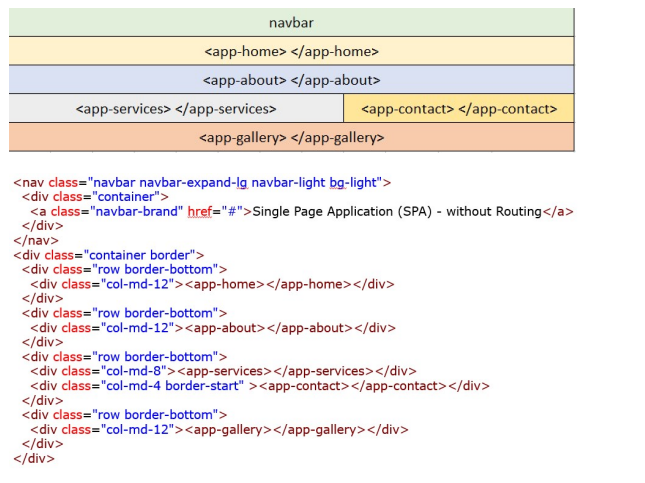
ng g c about ◊ check about.component.ts for selector: app-about

ng g c services ◊ check services.component.ts for selector: app-services

ng g c contact ◊ check contact.component.ts for selector: app-contact

ng g c gallery ◊ check gallery.component.ts for selector: app-gallery

**Add Custom Components in app.component.html file**

****

**Result :**

****

**Week 3**

**Exploring and Applications of Angular Templates**

**Experiment 3.1: Write a program to create template inline into app.component itself using template property of @Component decorator.**

**Experiment 3.2: Write a program to create external template.**

**Template**

Templates in Angular represents a view whose role is to display data and change the data whenever an event occurs. It's default language for templates is HTML.

**Using of Template**

Templates separate view layer from the rest of the framework so we can change the view layer without breaking the application.

How to create template

There are two ways of defining templates,

1. Inline Template
2. External Template

**Inline Template**

**Experiment 3.1: Write a program to create template inline into app.component itself using template property of @Component decorator.**

**Solution:**

**Step 1: At command prompt type:**

**ng new myapp**

**Step 2: Go to src/app folder**

First Open app.component.ts file and add the below line of code into @Component decorator,

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

@Component({

selector: 'app-root',

template:`<h1> This is inline template example</h1>

<div>

<div class="place">

Place:{{ user.Place }}

</div>

<div class="name">

Name:{{user.name}}

</div>

</div>`,

styles: [`

h1 { font-weight: normal; color:red;}

.place {font-weight: bold; color:blue; border: 5px outset red;}

.name {font-weight: bold; color:green; border: 5px outset red;}

`]

})

export class AppComponent {

title = 'myapp';

user={Place:"Hyderabad",

name:"Mallareddy University"};

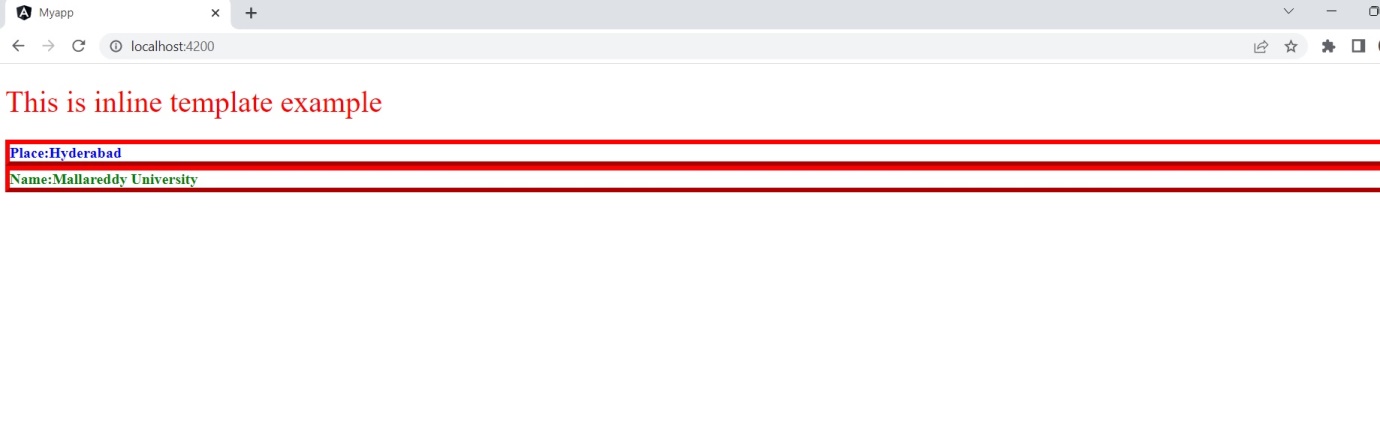
}

**Step 3:** To execute type the following command

ng serve –open

In the experiment the view and style are all inline. We don’t use any additional file to render into the main file.

Now run the application and you will see the below output screen,



**Experiment 3.2: Write a program to create external template.**

**Solution:**

**Step 1: create a new application as:**

**ng new myapp2**

**Step 2: Got to /myapp2/src/app**

**Step 3: Open the file app.component.ts**

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'myapp2';

country\_detail={country:"India", capital:"Delhi"}

}

Step 4: Open app.component.css and write

h1 { font-weight: normal; color:red;}

.country {font-weight: bold; color:blue;}

.capital {font-weight: bold; color:green;}

Step 5: Open app.component.html and write:

<h1> This is External Template Example</h1>

<div>

<div class="country">

Country Name:{{country\_detail.country}}

</div>

<div class="capital">

Country Name:{{country\_detail.capital}}

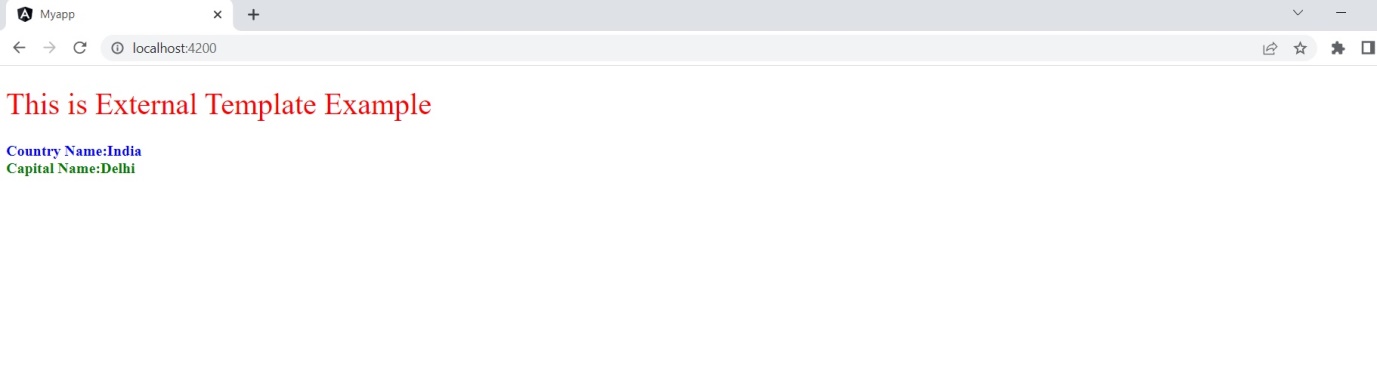
</div>

</div>

Step 6: Type the following command execute

ng serve –open

**Output:**



**Elements of Templates**

1. HTML
2. Interpolation
3. Template Expressions
4. Template Statements

Let's start with the explanation of each one of the template elements

**HTML**

Angular uses HTML as a template language.

**Interpolation**

Interpolation is one of the forms of data binding where we can access a component’s data in a template. For interpolation, we use double curly braces {{ }}.

**Template Expressions**

The text inside {{ }} is called as template expression.

Ex,

1. {{Expression}}

Angular first evaluates the expression and returns the result as a string. The scope of a template expression is a component instance. That means, if we write {{ Name }}, Name should be the property of the component to which this template is bound to.

**Template Statements**

Template Statements are the statements which respond to a user event.

1. (event) = {{Statement}}

**Exercise 3.1: Write an inline template to display your name and address**

Solution:

**Step 1: At command prompt type:**

**ng new myapp**

**Step 2: Go to src/app folder**

First Open app.component.ts file and add the below line of code into @Component decorator,

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

@Component({

selector: 'app-root',

template:`<h1> My Location Details</h1>

<div>

<div class="name">

Place:{{ user.name }}

</div>

<div class="address">

Name:{{user.address}}

</div>

</div>`,

styles: [`

h1 { font-weight: normal; color:red;}

.name {font-weight: bold; color:blue; }

.address {font-weight: bold; color:green;}

`]

})

export class AppComponent {

title = 'myapp';

user={name:"Ajay Pandey",

address:"Hyderbad"};

}

**Output:**

**Exercise 3.2: Write an external template to display student id and name.**

**Solution:**

**Step 1: create a new application as:**

**ng new myapp**

**Step 2: Got to /myapp/src/app**

**Step 3: Open the file app.component.ts**

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'myapp';

student\_detail={id:101, name:”Sanjay Paul”}

}

Step 4: Open app.component.css and write

h1 { font-weight: normal; color:red;}

.stud\_id {font-weight: bold; color:blue;}

.stud\_name {font-weight: bold; color:green;}

Step 5: Open app.component.html and write:

<h1> This is External Template Example</h1>

<div>

<div class="stud\_id">

Student ID:{{student\_detail.id}}

</div>

<div class="stud\_name">

Student Name:{{student\_detail.name}}

</div>

</div>

Step 6: Type the following command execute

ng serve –open

**Week 4**

**Experiment4: Manipulating data using pipes**

## Experiment 4.1: IMPLEMENTING ALL DEFAULT PIPES: Uppercase, Lowercase, Date, Currency, Json and Slice.

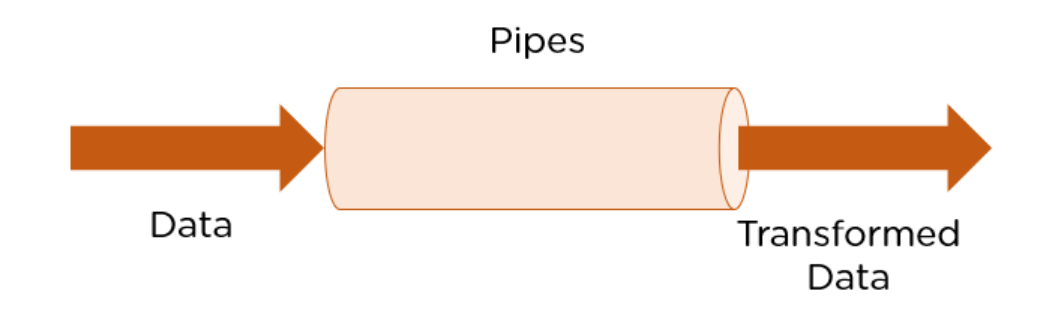
# Experiment 4.2: Create custom PIPES to multiply two numbers.

**INTRODUCTION**

Imagine if you could change the way certain information looked on the screen. This could be with respect to styling, size, or format. You can do all that and more with Angular Pipes. Angular Pipes allows its users to change the format in which data is being displayed on the screen. For instance, consider the date format. Dates can be represented in multiple ways, and the user can decide which one to use with the help of Angular Pipes.

## What are Angular Pipes?

Angular Pipes transform the output. You can think of them as makeup rooms where they beautify the data into a more desirable format. They do not alter the data but change how they appear to the user.



Technically, pipes are simple functions designed to accept an input value, process, and return a transformed value as the output. [Angular](https://www.simplilearn.com/tutorials/angular-tutorial/what-is-angular) supports many built-in pipes. However, you can also create custom pipes that suit your requirements. Some salient features include:

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

Some commonly used predefined Angular pipes are:

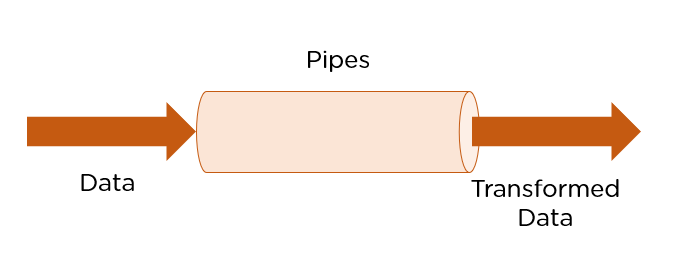
1. DatePipe: Formats a date value.
2. UpperCasePipe: Transforms text to uppercase.
3. LowerCasePipe: Transforms text to lowercase.
4. CurrencyPipe: Transforms a number to the currency string.
5. PercentPipe: Transforms a number to the percentage string.
6. DecimalPipe: Transforms a number into a decimal point string.

## Using Built-in Angular Pipes

Imagine if you could change the way certain information looked on the screen. This could be with respect to styling, size, or format. You can do all that and more with Angular Pipes. Angular Pipes allows its users to change the format in which [data](https://www.simplilearn.com/what-is-data-article) is being displayed on the screen. For instance, consider the date format. Dates can be represented in multiple ways, and the user can decide which one to use with the help of Angular Pipes.

## What are Angular Pipes?

Angular Pipes transform the output. You can think of them as makeup rooms where they beautify the data into a more desirable format. They do not alter the data but change how they appear to the user.



Technically, pipes are simple functions designed to accept an input value, process, and return a transformed value as the output. [Angular](https://www.simplilearn.com/tutorials/angular-tutorial/what-is-angular) supports many built-in pipes. However, you can also create custom pipes that suit your requirements. Some salient features include:

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

Some commonly used predefined Angular pipes are:

1. DatePipe: Formats a date value.
2. UpperCasePipe: Transforms text to uppercase.
3. LowerCasePipe: Transforms text to lowercase.
4. CurrencyPipe: Transforms a number to the currency string.
5. PercentPipe: Transforms a number to the percentage string.
6. DecimalPipe: Transforms a number into a decimal point string.

## Using Built-in Angular Pipes

## EXAMPLE 1:

**SAMPLE PROGRAM:**

Display the text given to uppercase using pipes.

In the **app.component.ts** file, we have defined the title variable –

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Angular 4 Project!';

}

The following line of code goes into the **app.component.html** file.

<html>

<body>

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

<b>{{title | lowercase}}</b>

</body>

</html>

**OUTPUT:**

Text

Description automatically generated

## Experiment 4.1: IMPLEMENTING ALL DEFAULT PIPES: Uppercase, Lowecase, Date, Currency, Json and Slice

## Solution:

## The following line of code will help us define the required variables in app.component.ts file −

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'My First App';

todaydate = new Date();

jsonval = {name:'Rox', age:'25', address:{a1:'Mumbai', a2:'Karnataka'}};

months = ["Jan", "Feb", "Mar", "April", "May", "Jun",

"July", "Aug", "Sept", "Oct", "Nov", "Dec"];

}

**We will use the pipes in the app.component.html file.**

<!--The content below is only a placeholder and can be replaced.-->

<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>

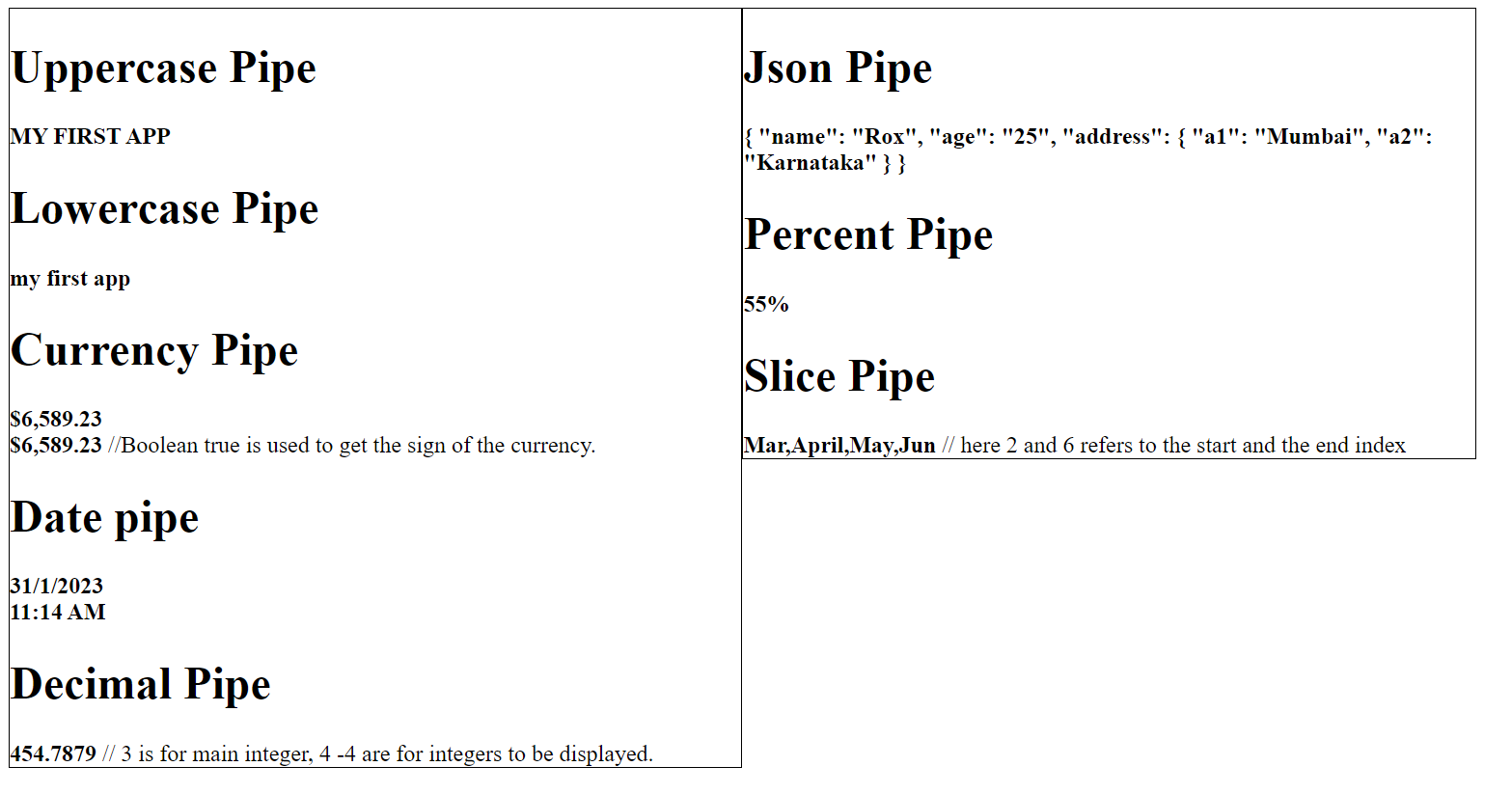
<b>{{months | slice:2:6}}</b>

// here 2 and 6 refers to the start and the end index

</div>

</div>

**OUTPUT:**



# Experiment 4.2: Create custom PIPES to multiply two numbers.

**Solution:**

Angular 2 also has the facility to create custom pipes. The general way to define a custom pipe is as follows.

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'Pipename'})

export class Pipeclass implements PipeTransform {

transform(parameters): returntype { }

}

Where,

* **'Pipename'** − This is the name of the pipe.
* **Pipeclass** − This is name of the class assigned to the custom pipe.
* **Transform** − This is the function to work with the pipe.
* **Parameters** − This are the parameters which are passed to the pipe.
* **Returntype** − This is the return type of the pipe.
* Let’s create a custom pipe that multiplies 2 numbers. We will then use that pipe in our component class.
* **Step 1** − First, create a file called multiplier.pipe.ts.
* **Step 2** − Place the following code in the above created file.

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({

name: 'multiplier'

})

export class MultiplierPipe implements PipeTransform {

transform(value: number, multiply: string): number {

let mul = parseFloat(multiply);

return mul \* value

}

}

Following points need to be noted about the above code.

* We are first importing the Pipe and PipeTransform modules.
* Then, we are creating a Pipe with the name 'Multiplier'.
* Creating a class called MultiplierPipe that implements the PipeTransform module.
* The transform function will then take in the value and multiple parameter and output the multiplication of both numbers.

**Step 3** − In the app.component.ts file, place the following code.

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'custom\_pipe';

}

**Note** − In our template, we use our new custom pipe.

**Step 4** − Ensure the following code is placed in the app.module.ts file.

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

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

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

import { AppComponent } from './app.component';

import { MultiplierPipe } from './multiplier.pipe';

@NgModule({

declarations: [

AppComponent,

MultiplierPipe

],

imports: [

BrowserModule,

AppRoutingModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

**Step 5**- Ensure the following code is placed in the app.component.html file.

<h1>Nothing is working <p>Multiplier: {{2 | multiplier: "10"}}</p> </h1>

Following things need to be noted about the above code.

* We need to ensure to include our Multiplier Pipe module.
* We also need to ensure it is included in the declarations section.

Once you save all the code changes and refresh the browser, you will get the following output.

**OUTPUT:**

****

**EXERCISE 4.1: Create a custom pipe to find square root.**

**Square Root:**

To create a custom pipe, we have created a new **ts** file. Here, we want to create the **sqrt** custom pipe. We have given the same name to the file and it looks as follows −

app.sqrt.ts

import {Pipe, PipeTransform} from '@angular/core';

@Pipe ({

name : 'sqrt'

})

export class SqrtPipe implements PipeTransform {

transform(val : number) : number {

return Math.sqrt(val);

}

}

To create a custom pipe, we have to import Pipe and Pipe Transform from Angular/core. In the @Pipe directive, we have to give the name to our pipe, which will be used in our .html file. Since, we are creating the sqrt pipe, we will name it sqrt.

As we proceed further, we have to create the class and the class name is **SqrtPipe**. This class will implement the **PipeTransform**.

The transform method defined in the class will take argument as the number and will return the number after taking the square root.

Since we have created a new file, we need to add the same in **app.module.ts.** This is done as follows −

**app.module.ts**

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

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

import { AppComponent } from './app.component';

import { NewCmpComponent } from './new-cmp/new-cmp.component';

import { ChangeTextDirective } from './change-text.directive';

import { SqrtPipe } from './app.sqrt';

@NgModule({

declarations: [

SqrtPipe,

AppComponent,

NewCmpComponent,

ChangeTextDirective

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

We have created the **app.sqrt.ts** class. We have to import the same in **app.module.ts** and specify the path of the file. It also has to be included in the declarations as shown above.

Let us now see the call made to the sqrt pipe in the **app.component.html** file.

<h1>Custom Pipe</h1>

<b>Square root of 25 is: {{25 | sqrt}}</b>

<br/>

<b>Square root of 729 is: {{729 | sqrt}}</b>

**Output:**

Graphical user interface, text, application

Description automatically generated

**Week 5**

**Expreiment 5** Creating Directives and Advanced Components

**CREATING DIRECTIVES:**

AngularJS directives are used to extend HTML. They are special attributes starting with **ng**-prefix. Let us discuss the following directives −

# The ng-app Directive

The **ng-app** directive is a starting point of AngularJS Application. It initializes the AngularJS framework automatically. AngularJS framework will first check for ng-app directive in a HTML document after the entire document is loaded and if ng-app is found, it bootstraps itself and compiles the HTML template.

Typically ng-app directives should be placed at the root of an HTML document e.g. <html> or <body> tag, so that it can control the entire DOM hierarchy.

However, you can place it in any DOM element.

The AngularJS framework will only process the DOM elements and its child elements where the ng-app directive is applied. Consider the following example.

Example: 5.1

**Step 1: At command prompt type:**

**ng new myapp**

**Step 2: Open to src/app.component.ts**

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'myapp';

name = ['Bill','Steve','Ram'];

}

**Step 3: Open to src/app.component.html**

<!DOCTYPE html>

<html>

<head>

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.7.9/angular.min.js"></script>

<style>

div {

border: 1px solid green;

width: 100%;

height: 50px;

display: block;

margin-bottom: 10px;

text-align: center;

background-color: yellow;

}

</style>

</head>

<body ng-app="myApp" ng-init="students=['Bill','Steve','Ram']">

<ol>

<li ng-repeat="name in students">

{{name}}

</li>

</ol>

<div ng-repeat="name in students">

{{name}}

</div>

</body>

</html>

**Step 4: Open cmd type ng serve --open**

**Step 5: It shows the output in** [**http://localhost:4200**](http://localhost:4200)

**OUTPUT:**

1. Bill,Steve,Ram

Bill,Steve,Ram

Example: 5.2

**Step 1: At command prompt type:**

**ng new myapp**

**Step 2: Open to src/app.component.ts**

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'myapp';

}

**Step 3: Open to src/app.component.html**

<!DOCTYPE html>

<html>

<head>

<title>ng-app Directive</title>

<script src="../Scripts/angular.min.js"></script>

</head>

<body >

<div>

{{2/2}}

</div>

<div id="myDiv" **ng-app**>

{{5/2}}

<div>

{{10/2}}

</div>

</div>

<div>

{{2/2}}

</div>

</body>

</html>

**Step 4: Open cmd type ng serve --open**

**Step 5: It shows the output in** [**http://localhost:4200**](http://localhost:4200)

**OUTPUT:**

1

2.5

5

1

**Angular Directives:**

|  |  |
| --- | --- |
| ng-app | Auto bootstrap AngularJS application. |
| ng-init | Initializes AngularJS variables |
| ng-model | Binds HTML control's value to a property on the $scope object. |
| ng-controller | Attaches the controller of MVC to the view. |
| ng-bind | Replaces the value of HTML control with the value of specified AngularJS expression. |
| ng-repeat | Repeats HTML template once per each item in the specified collection. |
| ng-show | Display HTML element based on the value of the specified expression. |
| ng-readonly | Makes HTML element read-only based on the value of the specified expression. |
| ng-disabled | Sets the disable attribute on the HTML element if specified expression evaluates to true. |
| ng-if | Removes or recreates HTML element based on an expression. |
| ng-click | Specifies custom behavior when an element is clicked. |

## ng-model

The ng-model directive is used for two-way data binding in AngularJS. It binds <input>, <select> or <textarea> elements to a specified property on the [$scope](https://www.tutorialsteacher.com/angularjs/angularjs-scope) object. So, the value of the element will be the value of a property and vica-versa.

## ng-bind

The ng-bind directive binds the model property declared via $scope or ng-model directive or the result of an expression to the HTML element. It also updates an element if the value of an expression changes.

Example: 5.3

**Step 1: At command prompt type:**

**ng new myapp**

**Step 2: Open to src/app.component.ts**

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'myapp';

}

**Step 3: Open to src/app.component.html**

<!DOCTYPE html>

<html >

<head>

<script src="~/Scripts/angular.js"></script>

</head>

<body ng-app="">

<div>

5 + 5 = <span ng-bind="5 + 5"></span> <br />

Enter your name: <input type="text" ng-model="name" /><br />

Hello <span ng-bind="name"></span>

</div>

</body>

</html>

**Step 4: Open cmd type ng serve --open**

**Step 5: It shows the output in** [**http://localhost:4200**](http://localhost:4200)

**OUTPUT:**

5+5= 10  
Enteryourname:   
Hello Manasa

In the above example, ng-bind directive binds a result of an expression "5 + 5" to the <span>. The same way, it binds a value of a model property "name" to the <span>. The value of "name" property will be the value entered in a textbox.

## ng-repeat

The ng-repeat directive repeats HTML once per each item in the specified array collection.

## ng-if

The ng-if directive creates or removes an HTML element based on the Boolean value returned from the specified expression. If an expression returns true then it recreates an element otherwise removes an element from the HTML document.

## ng-readonly

The ng-readonly directive makes an HTML element read-only, based on the Boolean value returned from the specified expression. If an expression returns true then the element will become read-only, otherwise not.

## ng-disabled

The ng-disabled directive disables an HTML element, based on the Boolean value returned from the specified expression. If an expression returns true the element will be disabled, otherwise not.

The following example demonstrates ng-if, ng-readonly, and ng-disabled directives.

Example: 5.4

**Step 1: At command prompt type:**

**ng new myapp**

**Step 2: Open to src/app.component.ts**

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'myapp';

}

**Step 3: Open to src/app.component.html**

<!DOCTYPE html>

<html>

<head>

<script src="~/Scripts/angular.js"></script>

<style>

div {

width: 100%;

height: 50px;

display: block;

margin: 15px 0 0 10px;

}

</style>

</head>

<body ng-app ng-init="checked=true" >

Click Me: <input type="checkbox" ng-model="checked" /> <br />

<div>

New: <input ng-if="checked" type="text" />

</div>

<div>

Read-only: <input ng-readonly="checked" type="text" value="This is read-only." />

</div>

<div>

Disabled: <input ng-disabled="checked" type="text" value="This is disabled." />

</div>

</body>

</html>

**Step 4: Open cmd type ng serve --open**

**Step 5: It shows the output in** [**http://localhost:4200**](http://localhost:4200)

**OUTPUT:**

Click Me: 

New: 

Read-only: 

Disabled: 

**EXERCISE 5:**

Write a HTML program by using AngularJS directives such as ng-app, ng-init, ng- model, ng-bind and ng- repeat.

<!DOCTYPE html**>**

**<html>**

**<head>**

**<title>**AngularJS Directives**</title>**

**</head>**

**<body>**

**<h1>**Sample Application**</h1>**

**<div** ng-app = "" ng-init = "countries = [{locale:'en-IND',name:'India'}, {locale:'en-PAK',name:'Pakistan'}, {locale:'en-AUS',name:'Australia'}]"**>**

**<p>**Enter your Name: **<input** type = "text" ng-model = "name"**></p>**

**<p>**Hello **<span** ng-bind = "name"**></span>**!**</p>**

**<p>**List of Countries with locale:**</p>**

**<ol>**

**<li** ng-repeat = "country in countries"**>**

               {{ 'Country: ' + country.name + ', Locale: ' + country.locale }}

**</li>**

**</ol>**

**</div>**

**<script** src = "http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"**></script>**

**</body>**

**</html>**

OUTPUT:

Enter your Name: 

Hello AIML!

List of Countries with locale:

1. Country: India, Locale: en-IND
2. Country: Pakistan, Locale: en-PAK
3. Country: Australia, Locale: en-AUS

**Week 6**

**Creation and implementation of Templates in JSX**

**Experiment 6: Write a React JSX application to print number table of 12.**

**Introduction:**

JSX stands for JavaScript XML. It is simply a syntax extension of JavaScript. It allows us to directly write HTML in React (within JavaScript code). It is easy to create a template using JSX in React, but it is not a simple template language instead it comes with the full power of JavaScript.  
It is faster than normal JavaScript as it performs optimizations while translating to regular JavaScript. Instead of separating the markup and logic in separated files, React uses *components* for this purpose.

**Characteristics of JSX:**

* JSX is not mandatory to use there are other ways to achieve the same thing but using JSX makes it easier to develop react application.
* JSX allows writing expression in { }. The expression can be any JS expression or React variable.
* To insert a large block of HTML we have to write it in a parenthesis i.e, ().
* JSX produces react elements.
* JSX follows XML rule.
* After compilation, JSX expressions become regular JavaScript function calls.
* JSX uses camelcase notation for naming HTML attributes. For example, tabindex in HTML is used as tabIndex in JSX.

**Advantages of JSX**

· SX makes it easier to write or add HTML in React.

· JSX can easily convert HTML tags to react elements.

· It is faster than regular JavaScript.

· JSX allows us to put HTML elements in DOM without using [appendChild()](https://www.geeksforgeeks.org/html-dom-appendchild-method/) or [createElement()](https://www.geeksforgeeks.org/html-dom-createelement-method/) method.

· As JSX is an expression, we can use it inside of if statements and for loops, assign it to variables, accept it as arguments, or return it from functions.

· JSX prevents XSS (cross-site-scripting) attacks popularly known as injection attacks.

· It is type-safe, and most of the errors can be found at compilation time.

**Disadvantages of JSX:**

* JSX throws an error if the HTML is not correct.
* In JSX HTML code must be wrapped in one top-level element otherwise it will give an error.
* If HTML elements are not properly closed JSX will give an error.

**Syntax:**

const element = <h1>Welcome to GeeksforGeeks.</h1>;

**Index.js**

import React from 'react';

import ReactDOM from 'react-dom';

const name = "Learner";

const element = <h1>Hello,

{ name }.Welcome to GeeksforGeeks.< /h1>;

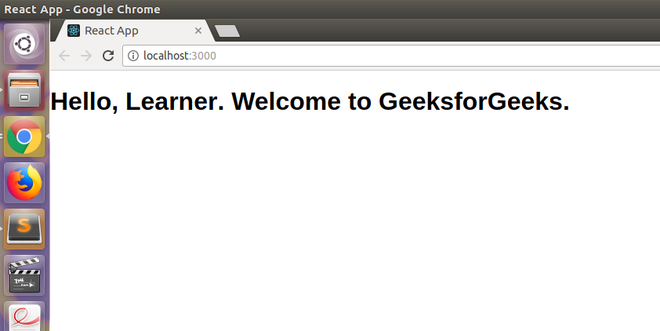
ReactDOM.render(

element,

document.getElementById("root")

);

**Output:**



**Sample: Write an React Application with JSX to display List of Flower Names.**

JSX stands for JavaScript XML.

JSX allows us to write HTML in React.

JSX makes it easier to write and add HTML in React.

## Coding JSX

JSX allows us to write HTML elements in JavaScript and place them in the DOM without any createElement()  and/or appendChild() methods.

JSX converts HTML tags into react elements.

**Source code**:

* use JSX

import React from 'react';

import ReactDOM from 'react-dom/client';

const myElement = <h1>I Love JSX!</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);

**OUTPUT**: I Love JSX!

* Without JSX:

import React from 'react';

import ReactDOM from 'react-dom/client';

const myElement = React.createElement('h1', {}, 'I do not use JSX!');

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);

# OUTPUT: I do not use JSX!

## Expressions in JSX

With JSX you can write expressions inside curly braces { }.

The expression can be a React variable, or property, or any other valid JavaScript expression. JSX will execute the expression and return the result:

Execute the expression 5 + 5:

import React from 'react';

import ReactDOM from 'react-dom/client';

const myElement = <h1>React is {5 + 5} times better with JSX</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);

**OUTPUT:**

# React is 10 times better with JSX

**Experiment 6: Write a React JSX application to print number table of 12.**

**Solution:**

**Step 1:** create a new react app using following command

npx reate-react-app myapp

**Step 2:** open file /myapp/src/App.js and add the following code

import './App.css';

import React from 'react';

function number\_table(number){

var num\_table=[]

var i=1;

for(i=1;i<=10;i++){

var m=number\*i

num\_table.push(<tr><td>{ number }\*{i}</td><td>{m}</td></tr>)

}

return num\_table;

}

function App() {

return (

<div>

<h1> Number Table of:12</h1>

<table border="1">

<tr><td>Number</td><td>Result</td></tr>

{number\_table(12)}

</table>

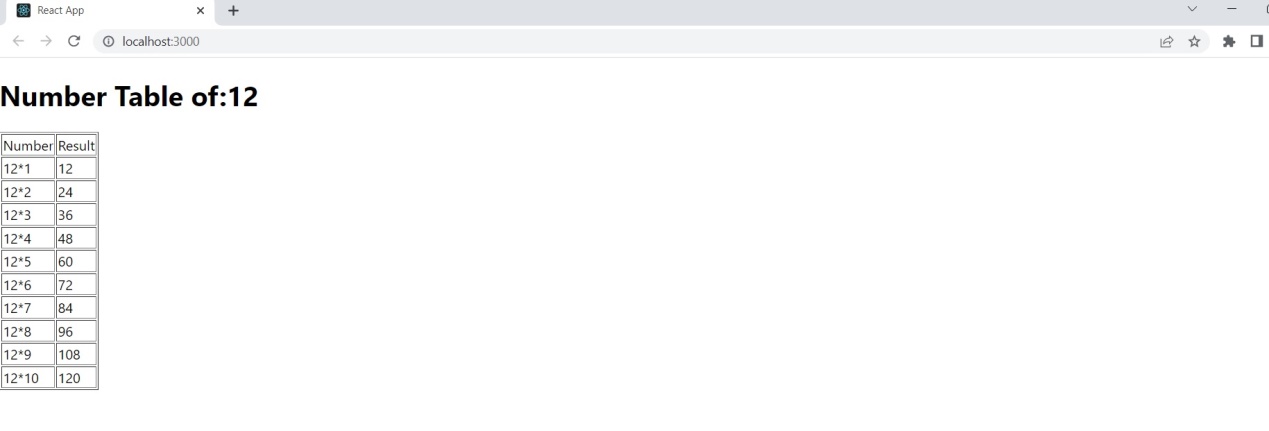
</div>

);

}

export default App;

**Output:**

****

**EXCERSIE:**

* 1. **Write HTML on multiple lines, put the HTML inside parentheses:**

import React from 'react';

import ReactDOM from 'react-dom/client';

const myElement = (

<ul>

<li>Apples</li>

<li>Bananas</li>

<li>Cherries</li>

</ul>);

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);

**OUTPUT:**

* Apples
* Bananas
* Cherries

**6.2 Write "Hello" if** x **is less than 10, otherwise "Goodbye":**

import React from 'react';

import ReactDOM from 'react-dom/client';

const x = 5;

let text = "Goodbye";

if (x < 10)

{

text = "Hello";

}

const myElement = <h1>{text}</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);

**OUTPUT:**

# Hello

# Week 7

**Applications building using reactJS and redux.**

# Expreiment 7:Write an redux based react JS application to concept of state using implement increment and decrement counter.

# React.js Introduction and Working

React is a declarative, efficient, and flexible JavaScript library for building user interfaces. ReactJS is an open-source, component-based front-end library responsible only for the view layer of the application. React is used to create modular user interfaces. It promotes the development of reusable UI components that display dynamic data.

React uses a declarative paradigm, react application aims to be both efficient and flexible. It designs simple views for each state in your application, and React will efficiently update and render just the right component when your data changes. The declarative view makes your code more predictable and easier to debug. A React application is made of multiple components, each responsible for rendering a small, reusable piece of HTML. Components can be nested within other components to allow complex applications to be built out of simple building blocks. A component may also maintain an internal state – for example, a Tab List component may store a variable corresponding to the currently open tab.

**Example:** Create a new React project by using the command below.

npx create-react-app myapp

**Filename App.js:** Now change the **App.js** file with the given below code.

import React, { Component } from "react";

class App extends Component {

render() {

return ( <div>

<h1>Hello, Learner.Welcome to GeeksforGeeks.</h1>

</div> );}

export default App;

1. **Example and solution**

##### **Write a program to create a simple calculator Application using React JS**

class App extends Component { constructor() {

super()

this.state = { operations: [] }

}

.....

}

render() {

return (

<div className="App">

<Display data={this.state.operations} />

<Buttons>

<Button onClick={this.handleClick} label="C" value="clear" />

<Button onClick={this.handleClick} label="7" value="7" />

<Button onClick={this.handleClick} label="4" value="4" />

<Button onClick={this.handleClick} label="1" value="1" />

<Button onClick={this.handleClick} label="0" value="0" />

<Button onClick={this.handleClick} label="/" value="/" />

<Button onClick={this.handleClick} label="8" value="8" />

<Button onClick={this.handleClick} label="5" value="5" />

<Button onClick={this.handleClick} label="2" value="2" />

<Button onClick={this.handleClick} label="." value="." />

<Button onClick={this.handleClick} label="x" value="\*" />

<Button onClick={this.handleClick} label="9" value="9" />

<Button onClick={this.handleClick} label="6" value="6" />

<Button onClick={this.handleClick} label="3" value="3" />

<Button label="" value="null" /><Button onClick={this.handleClick} label="-" value="-" />

<Button onClick={this.handleClick} label="+" size="2" value="+" />

<Button onClick={this.handleClick} label="=" size="2" value="equal" />

</Buttons>

</div>

)

class Buttons extends Component { render() {

return <div className="Buttons"> {this.props.children} </div>

}

} class Button extends Component { render() {

return (

<div

onClick={this.props.onClick} className="Button"

data-size={this.props.size} data-value={this.props.value}

>

{this.props.label}

</div>

)

}

}

class Display extends Component { render() {

const string = this.props.data.join('')

return <div className="Display"> {string} </div>

}

}

handleClick = e => {

const value = e.target.getAttribute('data-value') switch (value) {

case 'clear': this.setState({

operations: [],

})

break

case 'equal': this.calculateOperations() break

default:

const newOperations = update(this.state.operations, {

$push: [value],

})

this.setState({

operations: newOperations,

})

break

}

calculateOperations = () => {

let result = this.state.operations.join('') if (result) {

result = math.eval(result)

result = math.format(result, { precision: 14 })

result = String(result) this.setState({

operations: [result],

})

}

}

1. **Exercise and solution**

##### **Write a program to create a voting application using React JS**

CREATE

OR REPLACE VIEW "public"."poll\_results" AS SELECT

poll.id AS poll\_id, o.option\_id,

count(\*) AS votes

FROM (

(

SELECT

vote.option\_id, option.poll\_id, option.text

FROM (

vote

LEFT JOIN option ON ((option.id = vote.option\_id))

)

) o

LEFT JOIN poll ON ((poll.id = o.poll\_id))

)

GROUP BY

poll.question, o.option\_id, poll.id;

CREATE

OR REPLACE VIEW "public"."online\_users" AS SELECT

count(\*) AS count

FROM

"user" WHERE

(

"user".last\_seen\_at > (now() - '00:00:15' :: interval)

);

import { ApolloClient, HttpLink, InMemoryCache, split } from "@apollo/client"; import { GraphQLWsLink } from '@apollo/client/link/subscriptions';

import { createClient } from "graphql-ws";

import { getMainDefinition } from "@apollo/client/utilities"; const GRAPHQL\_ENDPOINT = "realtime-poll-example.hasura.app";

const scheme = (proto) =>

window.location.protocol === "https:" ? `${proto}s` : proto;

const wsURI = `${scheme("ws")}://${GRAPHQL\_ENDPOINT}/v1/graphql`; const httpURL = `${scheme("https")}://${GRAPHQL\_ENDPOINT}/v1/graphql`; const splitter = ({ query }) => {

const { kind, operation } = getMainDefinition(query) || {}; const isSubscription =

kind === "OperationDefinition" && operation === "subscription"; return isSubscription;

};

const cache = new InMemoryCache(); const options = { reconnect: true };

const wsLink = new GraphQLWsLink(createClient({ url: wsURI, connectionParams: { options }

}));

const httpLink = new HttpLink({ uri: httpURL }); const link = split(splitter, wsLink, httpLink); const client = new ApolloClient({ link, cache });

# React Redux:

# Redux is an open-source JavaScript library used to manage application state. React uses Redux for building the user interface. It was first introduced by **Dan Abramov** and **Andrew Clark** in **2015**.

React Redux is the official React binding for Redux. It allows React components to read data from a Redux Store, and dispatch **Actions** to the **Store** to update data. Redux helps apps to scale by providing a sensible way to manage state through a unidirectional data flow model. React Redux is conceptually simple. It subscribes to the Redux store, checks to see if the data which your component wants have changed, and re-renders your component.

Redux was inspired by Flux. Redux studied the Flux architecture and omitted unnecessary complexity.

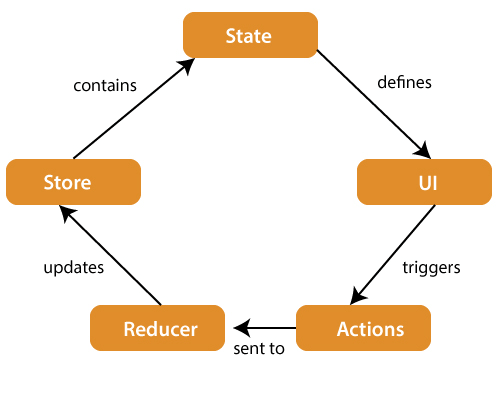
* Redux does not have Dispatcher concept.
* Redux has an only Store whereas Flux has many Stores.
* The Action objects will be received and handled directly by Store.

## Why use React Redux?

The main reason to use React Redux are:

* React Redux is the official **UI bindings** for react Application. It is kept up-to-date with any API changes to ensure that your React components behave as expected.
* It encourages good 'React' architecture.
* It implements many performance optimizations internally, which allows to components re-render only when it actually needs.

## Redux Architecture



The components of Redux architecture are explained below.

**STORE:** A Store is a place where the entire state of your application lists. It manages the status of the application and has a dispatch(action) function. It is like a brain responsible for all moving parts in Redux.

**ACTION:** Action is sent or dispatched from the view which are payloads that can be read by Reducers. It is a pure object created to store the information of the user's event. It includes information such as type of action, time of occurrence, location of occurrence, its coordinates, and which state it aims to change.

**REDUCER:** Reducer read the payloads from the actions and then updates the store via the state accordingly. It is a pure function to return a new state from the initial state.

Redux Installation

**Requirements:** React Redux requires React 16.8.3 or later version.

To use React Redux with React application, you need to install the below command.

$ npm install redux react-redux --save

##### React Redux

# Expreiment 7:

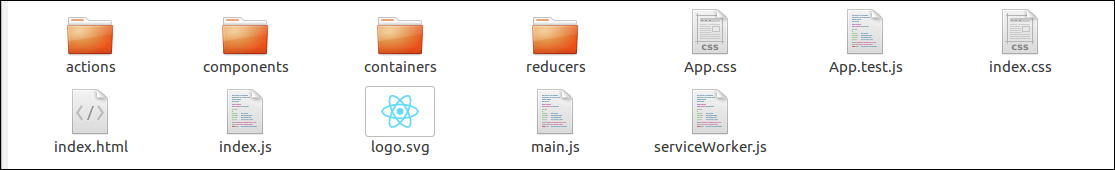
**Write an redux based react JS application to concept of state using implement increment and decrement counter.**

**Step-1** Create a new react project using **create-react-app** command. I choose the project name: "**reactproject**." Now, install **Redux** and **React-Redux**.

1. npx create-react-app reactproject
2. npm install redux react-redux --save
3. Npm install @reduxjs/toolkit

**Step-2 Create Files and Folders**

In this step, we need to create folders and files for actions, reducers, components, and containers. After creating folders and files, our project looks like as below image.



**Step-3 Create a folder “store” inside src folder and create a file Index.js as**

import {configureStore, createSlice} from '@reduxjs/toolkit';

const counterSlice=createSlice({

name: 'counter',

initialState: { counter:0 },

reducers:{

increment(state,action){

state.counter++;

},

decrement(state,action){

state.counter--;

}

}

})

export const actions=counterSlice.actions;

const store=configureStore({

reducer: counterSlice.reducer

})

export default store;

**Step 4:** Inside the src folder modify the app.js file as:

import './App.css';

import { useSelector,useDispatch } from 'react-redux';

import {store,actions} from './store/Index.js';

function App() {

const islogin=useSelector((state)=>state.counter);

const dispatch=useDispatch();

const increment=()=>{

dispatch(actions.increment());

};

const decrement=()=>{

dispatch(actions.decrement());

};

return (

<div class="container">

<h1>The counter test</h1>

<h2>Counter:{counter}</h2>

<button onClick={increment}>Increment</button>

<button onClick={decrement}>Decrement</button>

</div>

);

}

export default App;

Step 5: Again modify the Idex.js file in src folder as :

import React from 'react';

import ReactDOM from 'react-dom/client';

import './index.css';

import App from './App';

import reportWebVitals from './reportWebVitals';

import { Provider } from 'react-redux';

import store from './store/Index.js';

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(

<Provider store={store}>

<React.StrictMode>

<App />

</React.StrictMode>

</Provider>

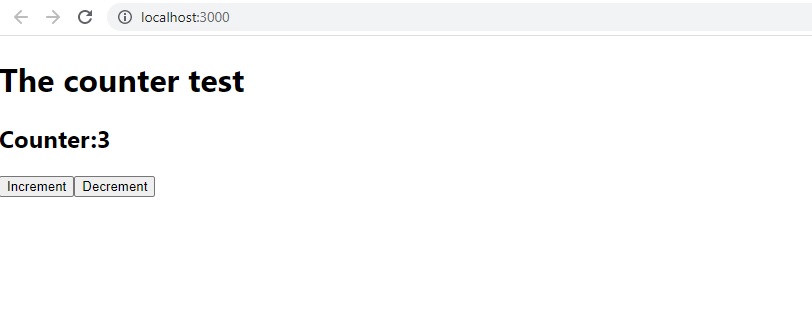
);

reportWebVitals();

Step 6: run the as:

npm start

**OUTPUT:**

****

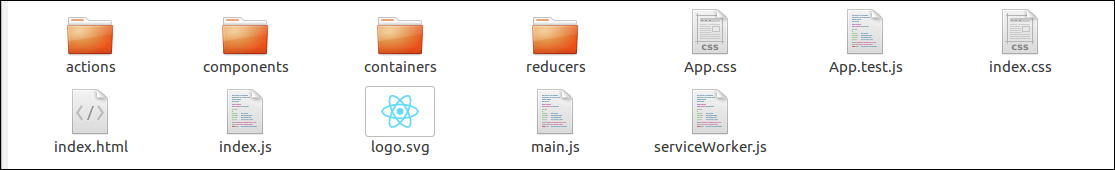
**Exercise 7.1: Implement redux to check whether user is logged in or not.**

**Step-1** Create a new react project using **create-react-app** command. I choose the project name: "**reactproject**." Now, install **Redux** and **React-Redux**.

1. npx create-react-app reactproject
2. npm install redux react-redux --save
3. Npm install @reduxjs/toolkit

**Step-2 Create Files and Folders**

In this step, we need to create folders and files for actions, reducers, components, and containers. After creating folders and files, our project looks like as below image.



**Step-3 Create a folder “store” inside src folder and create a file Index.js as**

import {configureStore, createSlice} from '@reduxjs/toolkit';

const loginSlice=createSlice({

name: 'login',

initialState: { isLoggedIn:false },

reducers:{

sub\_login(state,action){

state.isLoggedIn=true;

},

sub\_logout(state,action){

state.isLoggedIn=false;

}

}

})

export const actions=loginSlice.actions;

const store=configureStore({

reducer: loginSlice.reducer

})

export default store;

**Step 4:** Inside the src folder modify the app.js file as:

import { useSelector,useDispatch } from 'react-redux';

import {store,actions} from './store/Index.js';

import './App.css';

function checkLogin(islogin){

if(islogin)

return "LoggedIn";

else

return "LoggedOut";

}

function App() {

const islogin=useSelector((state)=>state.isLoggedIn);

const dispatch=useDispatch();

const login\_call=()=>{

dispatch(actions.sub\_login());

};

const logout\_call=()=>{

dispatch(actions.sub\_logout());

};

console.log(islogin);

return (

<div>

<h1>Log In form</h1>

User Name:<input type="text" id="txtName"/>

<button onClick={login\_call}>Login</button>

<button onClick={logout\_call}>Logout</button>

<h2>User is {checkLogin(islogin)} </h2>

</div>

);

}

export default App;

Step 5: Again modify the Idex.js file in src folder as :

import React from 'react';

import ReactDOM from 'react-dom/client';

import './index.css';

import App from './App';

import reportWebVitals from './reportWebVitals';

import {Provider} from 'react-redux';

import store from './store/Index';

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(

<Provider store={store}>

<React.StrictMode>

<App />

</React.StrictMode>

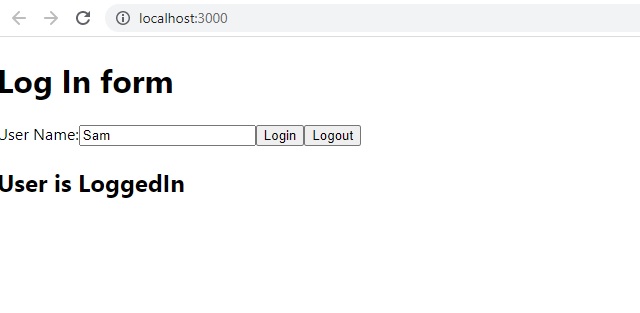
</Provider>

);

reportWebVitals();

Step 6: run the as:

npm start

**OUTPUT:**



**Week 8**

**Experiment 8.1: Creating and implementing modules**

What is Module?

**Modules** are the blocks of encapsulated code that communicates with an external application on the basis of their related functionality. Modules can be a single file or a collection of multiples files/folders. The reason programmers are heavily reliant on modules is because of their re-usability as well as the ability to break down a complex piece of code into manageable chunks.

**Modules are of three types:**

* Core Modules
* local Modules
* Third-party Modules

**Core Modules:** Node.js has many built-in modules that are part of the platform and comes with Node.js installation. These modules can be loaded into the program by using the **require** function.

**Syntax:**

**var module = require('module\_name');**

The require() function will return a JavaScript type depending on what the particular module returns. The following example demonstrates how to use the Node.js Http module to create a web server.

|  |
| --- |
| var http = require('http');  http.createServer(function (req, res) {    res.writeHead(200, {'Content-Type': 'text/html'});    res.write('Welcome to this page!');    res.end();  }).listen(3000); |

In the above example, the require() function returns an object because the Http module returns its functionality as an object. The function http.createServer() method will be executed when someone tries to access the computer on port 3000. The res.writeHead() method is the status code where 200 means it is OK, while the second argument is an object containing the response headers. Following are some examples of core modules.

|  |  |
| --- | --- |
| Core Modules | Description |
| http | creates an HTTP server in Node.js. |
| assert | set of assertion functions useful for testing. |
| fs | used to handle file system. |
| path | includes methods to deal with file paths. |
| process | provides information and control about the current Node.js process. |
| os | provides information about the operating system. |
| querystring | utility used for parsing and formatting URL query strings. |
| url | module provides utilities for URL resolution and parsing. |

**Local Modules:** Unlike built-in and external modules, local modules are created locally in your Node.js application. Let’s create a simple calculating module that calculates various operations. Create a calc.js file that has the following code:

**Filename: calc.js**

|  |
| --- |
| exports.add = function (x, y) {      return x + y;  };  exports.sub = function (x, y) {      return x - y;  };    exports.mult = function (x, y) {      return x \* y;  };  exports.div = function (x, y) {      return x / y;  }; |

Since this file provides attributes to the outer world via exports, another file can use its exported functionality using the require() function.

**Filename: index.js**

|  |
| --- |
| var calculator = require('./calc');  var x = 50, y = 10;  console.log("Addition of 50 and 10 is " + calculator.add(x, y));  console.log("Subtraction of 50 and 10 is " + calculator.sub(x, y));  console.log("Multiplication of 50 and 10 is " + calculator.mult(x, y));  console.log("Division of 50 and 10 is "  + calculator.div(x, y)); |

**Step to run this program:** Run **index.js** file using the following command:

node index.js

Since this file provides attributes to the outer world via exports, another file can use its exported functionality using the require() function.

**Output:**

Addition of 50 and 10 is 60

Subtraction of 50 and 10 is 40

Multiplication of 50 and 10 is 500

Division of 50 and 10 is 5

**Third-party modules:** Third-party modules are modules that are available online using the Node Package Manager(NPM). These modules can be installed in the project folder or globally. Some of the popular third-party modules are mongoose, express, angular, and react.

**Example:**

* npm install express
* npm install mongoose
* npm install -g @angular/cli

**Experiment:**

**Create a USER Module. Use this USER module to check logging credential of a user.**

Filename: user.js

var user1 = {

first\_name: "John",

last\_name: "Smith",

age: "38",

department: "Software",

user:"john123",

psw:"12jh23"

};

var user2 = {

first\_name: "Rohan",

last\_name: "Roy",

age: "40",

department: "Software",

user:"rohan456",

psw:"12rh78"

};

var users=new Array(user1,user2)

exports.log\_check=function(user,psw){

for (let value of users) {

if(user==value["user"] && psw==value["psw"])

return true;

}

return false;

}

Filename: login\_check.js

const prompt = require('prompt-sync')();

const name = prompt('User name?');

const psw=prompt("Password?")

var log\_mod=require("./user")

if(log\_mod.log\_check(name,psw)){

console.log("Valid User");

}

else{

console.log("Error:Invalid user.")

}

**Output:**

node login\_check.js

User name?john123

Password?12jh23

Valid User

node login\_check.js

User name?johnm45

Password?456

Error:Invalid user.

**Exercise 8.1: create date module which should return current date in “”YYYY-MM-DD” format.**

Filename: date\_mod.js

exports.today=function(){

// current timestamp in milliseconds

let ts = Date.now();

let date\_ob = new Date(ts);

let date = date\_ob.getDate();

let month = date\_ob.getMonth() + 1;

let year = date\_ob.getFullYear();

var dt=year + "-" + month + "-" + date

//return date in YYYY-MM-DD format

return dt;

}

Filename: testdate.js

var current\_date=require('./date\_mod.js')

console.log("Today is :"+current\_date.today());

OUTPUT:

node testdate.js

Today is :2023-1-29

**Experiment 8.2: File systems, Buffers, Stream and Events**

Introduction to File System, Buffers, Stream and Events:

**File System:**

To handle file operations like creating, reading, deleting, etc., Node.js provides an inbuilt module called FS (File System). Node.js gives the functionality of file I/O by providing wrappers around the standard POSIX functions. All file system operations can have synchronous and asynchronous forms depending upon user requirements. To use this File System module, use the require() method:

var fs = require('fs');

**Common use for File System module:**

* Read Files
* Write Files
* Append Files
* Close Files
* Delete Files

**What is Synchronous and Asynchronous approach?**

* **Synchronous approach:** They are called **blocking functions** as it waits for each operation to complete, only after that, it executes the next operation, hence blocking the next command from execution i.e. a command will not be executed until & unless the query has finished executing to get all the result from previous commands.
* **Asynchronous approach:** They are called **non-blocking functions** as it never waits for each operation to complete, rather it executes all operations in the first go itself. The result of each operation will be handled once the result is available i.e. each command will be executed soon after the execution of the previous command. While the previous command runs in the background and loads the result once it is finished processing the data.

Asynchronous read example

var fs = require("fs");

// Asynchronous read

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

if (err) {

return console.error(err);

}

console.log("Asynchronous read: " + data.toString());

});

Synchronous read example:

var fs = require("fs");

// Synchronous read

var data = fs.readFileSync('input.txt');

console.log("Synchronous read: " + data.toString());

**Buffers**

The Buffer class in Node.js is used to perform operations on raw binary data. Generally, Buffer refers to the particular memory location in memory. Buffer and array have some similarities, but the difference is array can be any type, and it can be resizable. Buffers only deal with binary data, and it can not be resizable. Each integer in a buffer represents a byte. console.log() function is used to print the Buffer instance.

**Methods to perform the** **operations on Buffer:**

|  |  |  |
| --- | --- | --- |
| No | Method | Description |
| 1 | Buffer.alloc(size) | It creates a buffer and allocates size to it. |
| 2 | Buffer.from(initialization) | It initializes the buffer with given data. |
| 3 | Buffer.write(data) | It writes the data on the buffer. |
| 4 | toString() | It read data from the buffer and returned it. |
| 5 | Buffer.isBuffer(object) | It checks whether the object is a buffer or not. |
| 6 | Buffer.length | It returns the length of the buffer. |
| 7 | Buffer.copy(buffer,subsection size) | It copies data from one buffer to another. |
| 8 | Buffer.slice(start, end=buffer.length) | It returns the subsection of data stored in a buffer. |
| 9 | Buffer.concat([buffer,buffer]) | It concatenates two buffers. |

**Stream:**

**Streams** are one of the fundamental concepts of Node.js. Streams are a type of data-handling methods and are used to read or write input into output sequentially. Streams are used to handle reading/writing files or exchanging information in an efficient way.

The stream module provides an API for implementing the stream interface. Examples of the stream object in Node.js can be a request to an HTTP server and process.stdout are both stream instances. In short, Streams are objects in Node.js that lets the user read data from a source or write data to a destination in a continuous manner.

**Accessing Streams:**

const stream = require('stream');

**Advantages of Streams over other data handling methods:**

* **Time Efficient:** We don’t have to wait until entire file has been transmitted. We can start processing data as soon as we have it.
* **Memory Efficient:** We don’t have to load huge amount of data in memory before we start processing.

**Types of Streams in Node.js:** There are namely four types of streams in Node.js.

1. **Writable:** We can write data to these streams. **Example:** fs.createWriteStream().
2. **Readable:** We can read data from these streams. **Example:** fs.createReadStream().
3. **Duplex:** Streams that are both, Writable as well as Readable. **Example:** net.socket.
4. **Transform:** Streams that can modify or transform the data as it is written and read. **Example:** zlib.createDeflate.

**Some Node APIs that uses streams are:**

* net.Socket()
* process.stdin()
* process.stdout()
* process.stderr()
* fs.createReadStream()
* fs.createWriteStream()
* net.connect()
* http.request()
* zlib.createGzip()
* zlib.createGunzip()
* zlib.createDeflate()
* zlib.createInflate()

**Event**

Node.js has an **event-driven architecture** which can perform asynchronous tasks. Node.js has **‘events’** module which emits named events that can cause corresponding functions or callbacks to be called. Functions(Callbacks) listen or subscribe to a particular event to occur and when that event triggers, all the callbacks subscribed to that event are fired one by one in order to which they were registered. **The EventEmmitter class:** All objects that emit events are instances of the EventEmitter class. The event can be emitted or listen to an event with the help of EventEmitter. **Syntax:**

const EventEmitter=require('events');

var eventEmitter=new EventEmitter();

**Listening events:** Before emits any event, it must register functions(callbacks) to listen to the events. **Syntax:**

eventEmitter.addListener(event, listener)

eventEmitter.on(event, listener)

eventEmitter.once(event, listener)

**eventEmmitter.on(event, listener)** and **eventEmitter.addListener(event, listener)** are pretty much similar. It adds the listener at the end of the listener’s array for the specified event. Multiple calls to the same event and listener will add the listener multiple times and correspondingly fire multiple times. Both functions return emitter, so calls can be chained. **eventEmitter.once(event, listener)** fires at most once for a particular event and will be removed from listeners array after it has listened once. Returns emitter, so calls can be chained. **Emitting events:** Every event is named event in nodejs. We can trigger an event by emit(event, [arg1], [arg2], […]) function. We can pass an arbitrary set of arguments to the listener functions. **Syntax:**

eventEmitter.emit(event, [arg1], [arg2], [...])

**Experiment:**

**a) Create a file, write few data in the file, append data, read the file and close the file.**

var fs = require("fs");

console.log("writing into existing file");

fs.writeFile('input.txt', 'Geeks For Geeks', function(err) {

if (err) {

return console.error(err);

}

console.log("Data written successfully!");

});

var data = "\nLearn Node.js";

// Append data to file

fs.appendFile('input.txt', data, 'utf8',

// Callback function

function(err) {

if (err) throw err;

// If no error

console.log("Data is appended to file successfully.")

});

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

if (err) {

return console.error(err);

}

console.log("Asynchronous read: " + data.toString());

});

// Get the file descriptor of the given path

file\_descriptor = fs.openSync("input.txt");

console.log("The file descriptor is:", file\_descriptor);

// Close the file descriptor

fs.close(file\_descriptor, (err) => {

if (err)

console.error('Failed to close file', err);

else {

console.log("\n> File Closed successfully");

}

});

**b) Allocate buffer for 1024 buytes, write data string into it and print length**

// Different Method to create Buffer

var buffer1 = Buffer.alloc(1024);

// Writing data to Buffer

buffer1.write("Happy Learning");

// Reading data from Buffer

var a = buffer1.toString('utf-8');

console.log(a);

// Check length of Buffer

console.log(buffer1.length);

**c) Write and Read streams**

// Sample JavaScript Code for

// Writable Stream

// Accessing Streams

const { Writable } = require('stream');

// Whatever is passed in standard

// input is out streamed here.

const outStream = new Writable({

// The Write function takes three

// arguments

// Chunk is for Buffer

// Encoding is used in case we want

// to configure the stream differently

// In this sample code, Encoding is ignored

// callback is used to indicate

// successful execution

write(chunk, encoding, callback) {

console.log(chunk.toString());

callback();

}

});

// Echo the data to the standard output

process.stdin.pipe(outStream);

// Sample JavaScript Code for creating

// a Readable Stream

// Accessing streams

const { Readable } = require('stream');

// Reading the data

const inStream = new Readable({

read() { }

});

// Pushing the data to the stream

inStream.push('GeeksForGeeks : ');

inStream.push(

'A Computer Science portal for Geeks');

// Indicates that no more data is

// left in the stream

inStream.push(null);

// Echoing data to the standard output

inStream.pipe(process.stdout);

**d) Write two functions “fun1” and “fun2” and register it to event listener.**

// Importing events

const EventEmitter = require('events');

// Initializing event emitter instances

var eventEmitter = new EventEmitter();

var fun1 = (msg) => {

console.log("Message from fun1: " + msg);

};

var fun2 = (msg) => {

console.log("Message from fun2: " + msg);

};

// Registering fun1 and fun2

eventEmitter.on('myEvent', fun1);

eventEmitter.on('myEvent', fun1);

eventEmitter.on('myEvent', fun2);

// Removing listener fun1 that was

// registered on the line 13

eventEmitter.removeListener('myEvent', fun1);

// Triggering myEvent

eventEmitter.emit('myEvent', "Event occurred");

// Removing all the listeners to myEvent

eventEmitter.removeAllListeners('myEvent');

// Triggering myEvent

eventEmitter.emit('myEvent', "Event occurred");

**Excercises 8.2:**

**Create a file and write “Mallareddy University” and append ” AIML Department” to the same file**

var fs = require("fs");

fs.writeFile('input.txt', Mallareddy University’, function(err) {

if (err) {

return console.error(err);

}

console.log("Data written successfully!");

});

// Append data to file

fs.appendFile('input.txt', ‘ AIML Department’, 'utf8',

// Callback function

function(err) {

if (err) throw err;

// If no error

console.log("Data is appended to file successfully.")

});

**Exercise 8.3:**

**Allocate a buffer of 100 bytes and write “Mallareddy University” into the buffer. Print the length of the buffer.**

var buffer1 = Buffer.alloc(100);

// Writing data to Buffer

buffer1.write("Mallareddy University");

// Reading data from Buffer

var a = buffer1.toString('utf-8');

console.log(a);

// Check length of Buffer

console.log(buffer1.length);

**Week 9**

**Experiment 9: Creating HTTP server and interacting with client using Node JS**

**Introduction to Node Js server with HTTP:**

The Hypertext Transfer Protocol, or HTTP, drives the Web. HTTP is a stateless, text-based protocol that works on top of TCP. An encrypted version of HTTP, named HTTP Secure, or HTTPS, is also commonly used when dealing with sensitive data. Traditionally, a browser is used as the client in an HTTP transaction, but you’ll see that this is not always the case. When a browser navigates to a given URL, an HTTP request is made to the server that hosts the URL. As you learned in Chapter 10, this request is normally made on TCP port 80 (or 443 if HTTPS is in use). The server processes the request and then responds to the client. This is how HTTP works at a *very* high level.

A basic server can be written as

var http = require("http");

var server = http.createServer(function(request, response) {

response.write("Hello <strong>HTTP</strong>!");

response.end();

});

server.listen(8000);

**Experiment:**

**Create a HTTP server which will forward a HTML page in response of a client request.**

***Filename: Indext.html***

<!DOCTYPE html>

<head>

<title>My Website</title>

<style>

\*,

html {

margin: 0;

padding: 0;

border: 0;

}

html {

width: 100%;

height: 100%;

}

body {

width: 100%;

height: 100%;

position: relative;

background-color: rgb(236, 152, 42);

}

.center {

width: 100%;

height: 50%;

margin: 0;

position: absolute;

top: 50%;

left: 50%;

transform: translate(-50%, -50%);

color: white;

font-family: "Trebuchet MS", Helvetica, sans-serif;

text-align: center;

}

h1 {

font-size: 144px;

}

p {

font-size: 64px;

}

</style>

</head>

<body>

<div class="center">

<h1>Hello Again!</h1>

<p>This is served from a file</p>

</div>

</body>

</html>

***Filename: http\_server.js***

const http = require("http");

const fs = require('fs').promises;

const host = 'localhost';

const port = 8000;

const requestListener = function (req, res) {

fs.readFile(\_\_dirname + "/index.html")

.then(contents => {

res.setHeader("Content-Type", "text/html");

res.writeHead(200);

res.end(contents);

})

.catch(err => {

res.writeHead(500);

res.end(err);

return;

});

};

const server = http.createServer(requestListener);

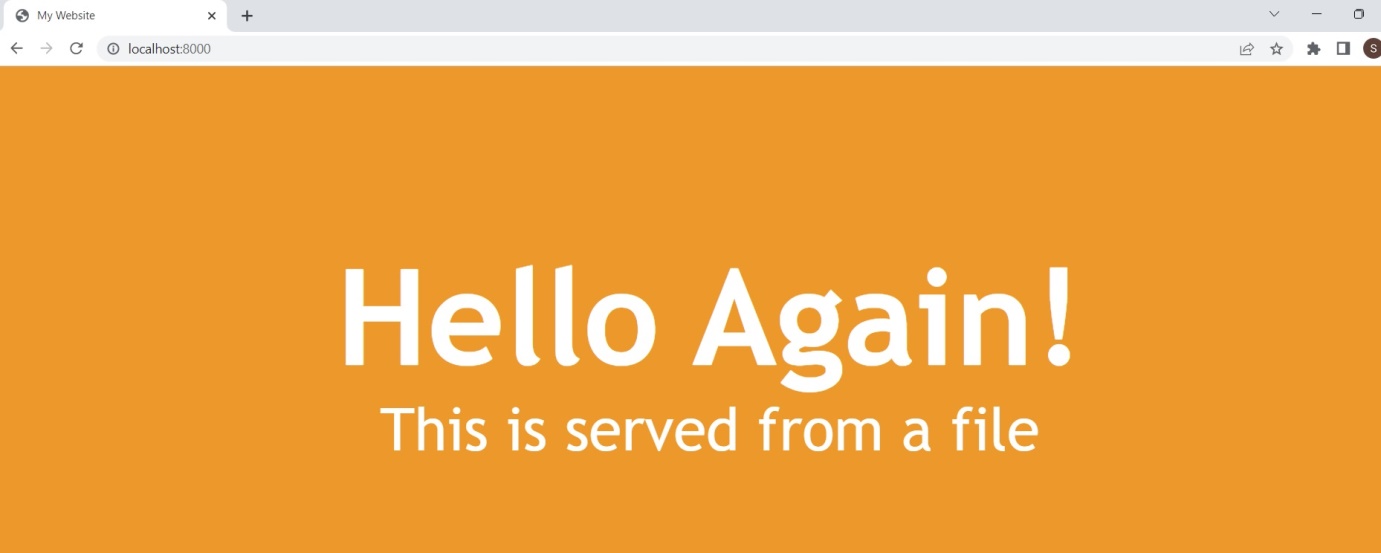
server.listen(port, host, () => {

console.log(`Server is running on http://${host}:${port}`);

});

Output:

node http\_server.js



**Exercise9.1: Create a HTTP server which will respond by sending message “server is active”**

Filename: new\_http\_server.js

const http = require("http");

const fs = require('fs').promises;

const host = 'localhost';

const port = 8000;

const requestListener = function (req, res) {

res.setHeader("Content-Type", "application/json");

res.writeHead(200);

res.end("The server is active");

}

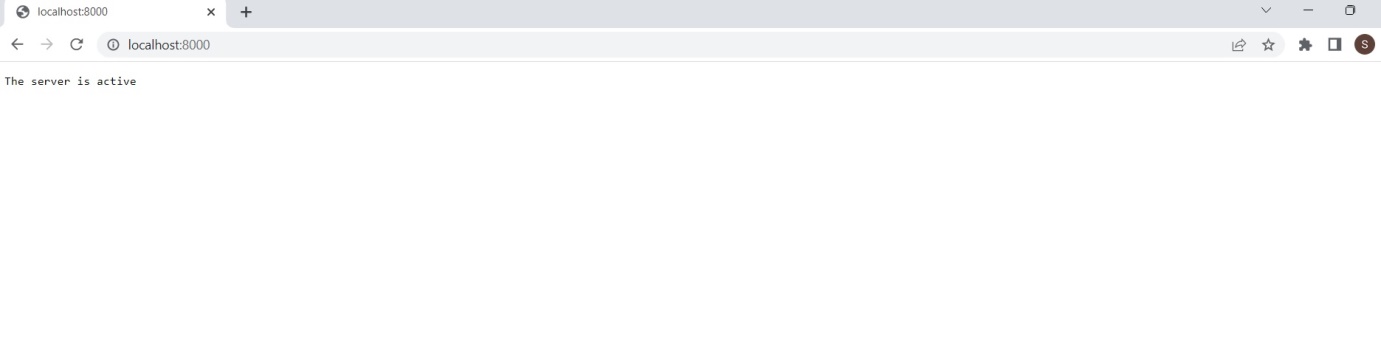
const server = http.createServer(requestListener);

server.listen(port, host, () => {

console.log(`Server is running on http://${host}:${port}`);

});

Output:



**Week 10**

**Experiment 10: Develop Application to store, maintain data using Node JS and database**.

Introduction to database connectivity:

MongoDB is a NoSQL database used to store large amounts of data without any traditional relational database table. Instead of rows & columns, MongoDB used collections & documents to store data. A collections consist of a set of documents & a document consists of key-value pairs which are the basic unit of data in MongoDB.

Make sure that MongoDB installs on your pc.

To connect a Node.js application to MongoDB, we have to use a library called **Mongoose**.

const mongoose = require("mongoose");

After that, we have to call the connect method of Mongoos.

To develop application in node js another framework is needed called express js. Express.js is a small framework that works on top of Node.js web server functionality to simplify its APIs and add helpful new features. It makes it easier to organize your application’s functionality with middleware and routing. It adds helpful utilities to Node.js HTTP objects and facilitates the rendering of dynamic HTTP objects.

**Why Express ?**

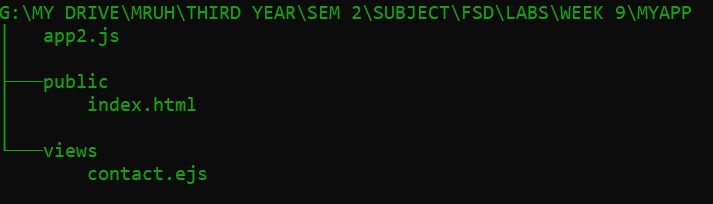
* Develops Node.js web applications quickly and easily.
* It’s simple to set up and personalise.
* Allows you to define application routes using HTTP methods and URLs.
* Includes a number of middleware modules that can be used to execute additional requests and responses activities.
* Simple to interface with a variety of template engines, including Jade, Vash, and EJS.
* Allows you to specify a middleware for handling errors.

**Installing Express:**

npm install express -g

**Experiment: Create contact form to store data email and query using node js express and monodb**.

Create the application directory as:



***Filename: app2.js***

const express = require("express");

const ejs = require("ejs");

const mongoose = require("mongoose");

const bodyParser = require("body-parser");

mongoose.set('strictQuery', true);

mongoose.connect("mongodb://127.0.0.1:27017/sample", {

useNewUrlParser: true,

useUnifiedTopology: true

}).then(db => console.log('DB is connected'))

.catch(err => console.log(err));

const contactSchema = {

email: String,

query: String,

};

const Contact = mongoose.model("contact", contactSchema);

const app = express();

app.set("view engine", "ejs");

app.engine("ejs",ejs.renderFile)

app.use(bodyParser.urlencoded({

extended: true

}));

app.use(express.static(\_\_dirname+'/public'));

app.get("/contact", function(req, res){

res.render("contact");

});

app.post("/contact", function (req, res) {

console.log(req.body.email);

const contact = new Contact({

email: req.body.email,

query: req.body.query,

});

contact.save(function (err) {

if (err) {

throw err;

} else {

res.render("contact");

}

});

});

app.listen(3000, function(){

console.log("App is running on Port 3000");

});

***Filename: index.html***

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content=

"width=device-width, initial-scale=1.0">

<title>Document</title>

</head>

<body>

<form action="/contact" method="post">

<input type="text" placeholder="Email" name="email">

<input type="text" placeholder="Query" name="query">

<button type="submit">Submit</button>

</form>

</body>

</html>

***Filename: contact.ejs***

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content=

"width=device-width, initial-scale=1.0">

<title>Document</title>

</head>

<body>

<form action="/contact" method="post">

<input type="text" placeholder="Email" name="email">

<input type="text" placeholder="Query" name="query">

<button type="submit">Submit</button>

</form>

</body>

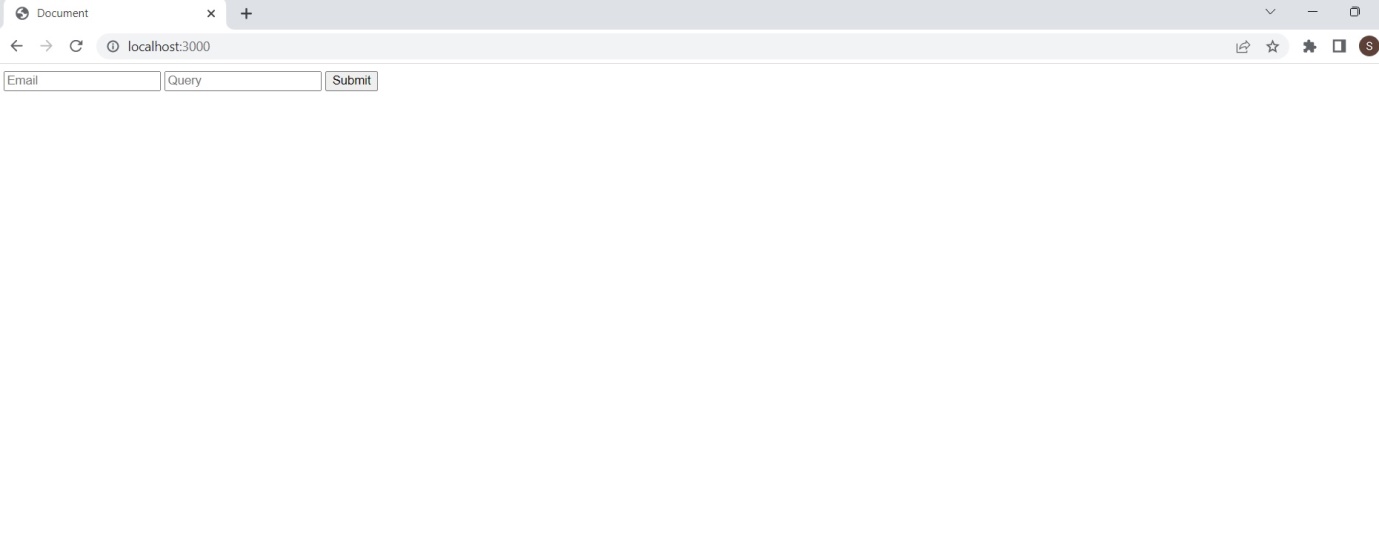
</html>

Output:

node app2.js

App is running on Port 3000

DB is connected



**Exercise 10.1: Create contact form to store data name and address using node js express and monodb.**

***Filename: app.js***

const express = require("express");

const ejs = require("ejs");

const mongoose = require("mongoose");

const bodyParser = require("body-parser");

mongoose.set('strictQuery', true);

mongoose.connect("mongodb://127.0.0.1:27017/sample", {

useNewUrlParser: true,

useUnifiedTopology: true

}).then(db => console.log('DB is connected'))

.catch(err => console.log(err));

const contactSchema = {

name: String,

address: String,

};

const Contact = mongoose.model("contact", contactSchema);

const app = express();

app.set("view engine", "ejs");

app.engine("ejs",ejs.renderFile)

app.use(bodyParser.urlencoded({

extended: true

}));

app.use(express.static(\_\_dirname+'/public'));

app.get("/contact", function(req, res){

res.render("contact");

});

app.post("/contact", function (req, res) {

console.log(req.body.name);

const contact = new Contact({

name: req.body.name,

address: req.body.address,

});

contact.save(function (err) {

if (err) {

throw err;

} else {

res.render("contact");

}

});

});

app.listen(3000, function(){

console.log("App is running on Port 3000");

});

***Filename: index.html***

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content=

"width=device-width, initial-scale=1.0">

<title>Document</title>

</head>

<body>

<form action="/contact" method="post">

<input type="text" placeholder="Name" name="name">

<input type="text" placeholder="Address" name="address">

<button type="submit">Submit</button>

</form>

</body>

</html>

***Filename: contact.ejs***

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content=

"width=device-width, initial-scale=1.0">

<title>Document</title>

</head>

<body>

<form action="/contact" method="post">

<input type="text" placeholder="Name" name="name">

<input type="text" placeholder="Address" name="address">

<button type="submit">Submit</button>

</form>

</body>

</html>

**Week 11**

**Create, delete, update collection of Mongodb documents. Different approaches to migrate data Mongodb.**

**Experiment 11 : Create database employee. Create a collection in employee database and insert following data. Update salary of “BInay”. Delete the record for “Ajay”.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **address** | **Phone\_no** | **salary** |
| **Ajay** | **Hyderabad** | **9000789211** | **10000** |
| **Binay** | **Delhi** | **9825800212** | **8000** |
| **Manis** | **Mumbai** | **9725652582** | **18000** |

**Experiment 11.2: create a Json file containing employee data. And import the data into mongodb database.**

**Introduction:**

MongoDB, the most popular NoSQL database, is an open-source document-oriented database. The term ‘NoSQL’ means ‘non-relational’. It means that MongoDB isn’t based on the table-like relational database structure but provides an altogether different mechanism for storage and retrieval of data. This format of storage is called BSON ( similar to JSON format).

A simple MongoDB document Structure:

{

title: 'Geeksforgeeks',

by: 'Harshit Gupta',

url: 'https://www.geeksforgeeks.org',

type: 'NoSQL'

}

SQL databases store data in tabular format. This data is stored in a predefined data model which is not very much flexible for today’s real-world highly growing applications. Modern applications are more networked, social and interactive than ever. Applications are storing more and more data and are accessing it at higher rates.

Relational Database Management System(RDBMS) is not the correct choice when it comes to handling big data by the virtue of their design since they are not horizontally scalable. If the database runs on a single server, then it will reach a scaling limit. NoSQL databases are more scalable and provide superior performance. MongoDB is such a NoSQL database that scales by adding more and more servers and increases productivity with its flexible document model.

Features of MongoDB:

Document Oriented: MongoDB stores the main subject in the minimal number of documents and not by breaking it up into multiple relational structures like RDBMS. For example, it stores all the information of a computer in a single document called Computer and not in distinct relational structures like CPU, RAM, Hard disk, etc.

Indexing: Without indexing, a database would have to scan every document of a collection to select those that match the query which would be inefficient. So, for efficient searching Indexing is a must and MongoDB uses it to process huge volumes of data in very less time.

Scalability: MongoDB scales horizontally using sharding (partitioning data across various servers). Data is partitioned into data chunks using the shard key, and these data chunks are evenly distributed across shards that reside across many physical servers. Also, new machines can be added to a running database.

Replication and High Availability: MongoDB increases the data availability with multiple copies of data on different servers. By providing redundancy, it protects the database from hardware failures. If one server goes down, the data can be retrieved easily from other active servers which also had the data stored on them.

Aggregation: Aggregation operations process data records and return the computed results. It is similar to the GROUPBY clause in SQL. A few aggregation expressions are sum, avg, min, max, etc.

Where do we use MongoDB?

MongoDB is preferred over RDBMS in the following scenarios:

Big Data: If you have huge amount of data to be stored in tables, think of MongoDB before RDBMS databases. MongoDB has built-in solution for partitioning and sharding your database.

Unstable Schema: Adding a new column in RDBMS is hard whereas MongoDB is schema-less. Adding a new field does not effect old documents and will be very easy.

Distributed data Since multiple copies of data are stored across different servers, recovery of data is instant and safe even if there is a hardware failure.

Language Support by MongoDB:

MongoDB currently provides official driver support for all popular programming languages like C, C++, Rust, C#, Java, Node.js, Perl, PHP, Python, Ruby, Scala, Go, and Erlang.

Installing MongoDB:

Just go to http://www.mongodb.org/downloads and select your operating system out of Windows, Linux, Mac OS X and Solaris. A detailed explanation about the installation of MongoDB is given on their site.

Terminologies:

A MongoDB Database can be called as the container for all the collections.

Collection is a bunch of MongoDB documents. It is similar to tables in RDBMS.

Document is made of fields. It is similar to a tuple in RDBMS, but it has dynamic schema here. Documents of the same collection need not have the same set of fields.

**mongo:** The Command Line Interface to interact with the db.

**mongod**: This is the database. Sets up the server.

**mongodump**: It dumps out the Binary of the Database(BSON)

**mongoexport**: Exports the document to Json, CSV format

**mongoimport**: To import some data into the DB.

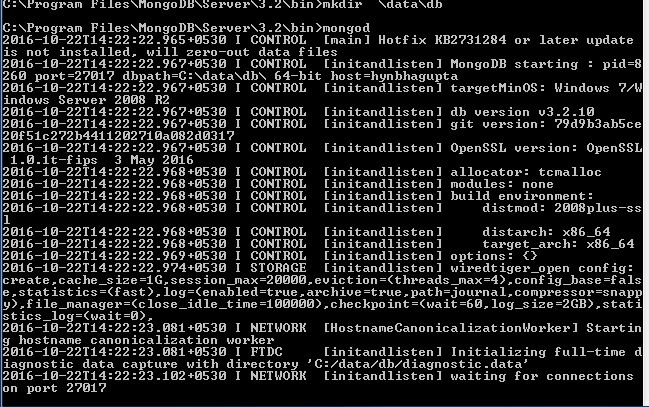
**mongorestore**: to restore anything that you’ve exported.

**mongostat**: Statistics of databases

**How to Start Mongo DB server?**

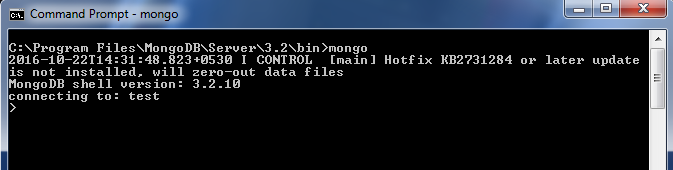
Step 1: Go to default installation path

Step 2: type **mongod** to start the server.



**How to start the client?**

Step 1: go to default location for Mongodb

Step 2: type **mongo** to start the client

**The create database:**

Step 1: use Mydb

Step 2: db.user.insert({name: "Ada Lovelace", age: 205})

So, database “Mydb” is created with collection “user”.

The deletion of database:

**Syntex:**

db.dropDatabase()

Example:

use temp

db.dropDatabase()

**The createCollection() Method**

MongoDB db.createCollection(name, options) is used to create collection.

**Syntax**

Basic syntax of createCollection() command is as follows −

db.createCollection(name, options)

|  |  |  |
| --- | --- | --- |
| Parameter | Type | Description |
| Name | String | Name of the collection to be created |
| Options | Document | (Optional) Specify options about memory size and indexing |

Options parameter is optional, so you need to specify only the name of the collection. Following is the list of options you can use −

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| capped | Boolean | (Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also. |
| autoIndexId | Boolean | (Optional) If true, automatically create index on \_id field.s Default value is false. |
| size | number | (Optional) Specifies a maximum size in bytes for a capped collection. If capped is true, then you need to specify this field also. |
|  |  |  |
| max | number | (Optional) Specifies the maximum number of documents allowed in the capped collection. |
|  |  |  |

Basic syntax of createCollection() method without options is as follows −

>use test

switched to db test

>db.createCollection("mycollection")

{ "ok" : 1 }

You can check the created collection by using the command show collections.

>show collections

mycollection

system.indexes

The following example shows the syntax of createCollection() method with few important options −

> db.createCollection("mycol", { capped : true, autoIndexID : true, size : 6142800, max : 10000 } ){

"ok" : 0,

"errmsg" : "BSON field 'create.autoIndexID' is an unknown field.",

"code" : 40415,

"codeName" : "Location40415"

}

The deletion of Collection:

Syntex:

|  |
| --- |
|  |
| **db.collection.drop(<options>)** |

Example:

db.students.drop()

**The creation of documents:**

To insert data into MongoDB collection, you need to use MongoDB's insert() or save() method.

**Syntax**

The basic syntax of insert() command is as follows −

db.COLLECTION\_NAME.insert(document)

**The update of Documents:**

The update() method updates the values in the existing document.

**Syntax**

The basic syntax of update() method is as follows −

>db.COLLECTION\_NAME.update(SELECTION\_CRITERIA, UPDATED\_DATA)

Example

Consider the mycol collection has the following data.

{ "\_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}

{ "\_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}

{ "\_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}

Following example will set the new title 'New MongoDB Tutorial' of the documents whose title is 'MongoDB Overview'.

>db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New MongoDB Tutorial'}})

WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

>db.mycol.find()

{ "\_id" : ObjectId(5983548781331adf45ec5), "title":"New MongoDB Tutorial"}

{ "\_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}

{ "\_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}

**Experiment 11 : Create database employee. Create a collection in employee database and insert following data. Update salary of “Binay”. Delete the record for “Ajay”.**

**Solution:**

**Create database:**

**use employee**

**Create collection:**

db.createCollection("employee\_details")

Insert Data:

db.employee\_details.insert({name:”Ajay”,Address:”Hyderabad”,Phone\_no: 9000789211,salary: 10000})

db.employee\_details.insert({name:”Binay”, Address:”Delhi”, Phone\_no: 9825800212,salary: 8000})

db.employee\_details.insert({name:”Manis”, Address:”Mumbai”, Phone\_no: 9725652582,salary: 18000})

**Update:**

db. employee\_details.update({'name':'Binay'},{$set:{'salary':12000}})

**Delete Document:**

db.employee\_details.remove( { name: "Ajay" }).

**Exercise:**

**Create database employee. Create a collection in students database and insert following data. Update marks of “Smith”. Delete the record for “Binod”.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **address** | **Phone\_no** | **Marks** |
| **Binod** | **Hyderabad** | **9000789211** | **88** |
| **Smith** | **Delhi** | **9825800212** | **90** |
| **John** | **Mumbai** | **9725652582** | **65** |

**Solution:**

**Create database:**

**use students**

**Create collection:**

db.createCollection("students\_details")

Insert Data:

db. students\_details.insert({name:”Binod”,Address:”Hyderabad”,Phone\_no: 9000789211,Marks: 88})

db. students\_details.insert({name:” Smith”, Address:”Delhi”, Phone\_no: 9825800212, Marks: 90})

db. students\_details.insert({name:” John”, Address:”Mumbai”, Phone\_no: 9725652582, Marks: 65})

**Update:**

db. students\_details.update({'name':'Smith'},{$set:{'Marks':85}})

**Delete Document:**

db. students\_details.remove( { name: "Binod" }).

**Experiment 11.2: create a Json file containing employee data. And import the data into mongodb database.**

**Solution:**

**Step1**: create a file employee.json

[

{"name":"Ajay", "address":"Hyderabad","Phone\_no":9000789211,"salary":10000},

{"name":"Binay", "address":"Hyderabad","Phone\_no":9000789211,"salary":8000},

{"name":"Ajay", "address":"Hyderabad","Phone\_no":9000789211,"salary":18000}

]

**Step 2**: use employee

**Step 3**: mongoimport --jsonArray --db gfg --collection employee\_details --file employee.json

**Exercise 11.2: create a Json file containing student data data. And import the data into mongodb database.**

**Solution:**

**Step1**: create a file student.json

[

{"name":" Binod ", "address":"Hyderabad","Phone\_no":9000789211,"salary":88},

{"name":" Smith ", "address":"Hyderabad","Phone\_no":9000789211,"salary":90},

{"name":" John ", "address":"Hyderabad","Phone\_no":9000789211,"salary":65}

]

**Step 2**: use student

**Step 3**: mongoimport --jsonArray --db gfg --collection student\_details --file student.json

**Week 12**

**Experiment 12: Create a student database and using angular JS & node JS collect, update and delete student data**.

***Introduction:***

The student database management is a complete application in which all kind of operations like addition, deletion an update of records or information is possible. The database schema for of the student database is as follows:

|  |  |
| --- | --- |
| Attributes | Type |
| Name | String |
| Marks | Number |

The complete function of this application depends on three server access:

1. Angular JS server which hosts all the front end designs.
2. Mongodb server which hosts our backend database.
3. Node JS server which the web server for our application.

The front end design steps are as follows:

Step 1: create a new project student\_app as:

ng new student\_app

**Step 2**: Install require packages and copy node\_module/dist/bootstrap/css/bootstrap.css file conetent to angular project style.css file.

npm install express --save

npm install bootstrap –save

**Step 3**: modify app.module.ts file as follows:

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

import { bootstrapApplication, BrowserModule } from '@angular/platform-browser';

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

import { AppComponent } from './app.component';

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

import { HttpClientModule } from '@angular/common/http';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

AppRoutingModule,

FormsModule,

HttpClientModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

**Step 4:** Modify app.component.ts as follows:

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

import { HttpClient } from '@angular/common/http';

import { ColdObservable } from 'rxjs/internal/testing/ColdObservable';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'db\_test';

name: string ="";

marks:Number=0;

\_id:string="";

StudentArray : any[] = [];

currentStudentID = "";

data(id: string) {

this.\_id = id;

console.log("Id selected:"+this.\_id);

}

constructor(private http: HttpClient )

{

this.getAllStudent();

}

getAllStudent() {

this.http.get("http://localhost:8000/getAll")

.subscribe((resultData: any)=>

{

console.log(resultData);

this.StudentArray = resultData;

});

}

setUpdate(id:string)

{

this.name=this.StudentArray.find(x=>x.\_id==id).name;

this.marks=this.StudentArray.find(x=>x.\_id==id).marks;

this.currentStudentID=id;

}

UpdateRecords()

{

if(this.currentStudentID==''){

alert("Select Data for Update");

console.log("Data is not selected");

}

//this.marks=this.StudentArray[id]

else{

let bodyData = {

"name" : this.name,

"marks" : this.marks,

"\_id":this.currentStudentID

};

this.http.patch("http://localhost:8000/update",bodyData).subscribe((resultData: any)=>

{

console.log(resultData);

alert("Student Updateddd")

//this.getAllStudent();

});

this.getAllStudent();

}

}

save()

{

console.log("data:",{"name":this.name,"marks":this.marks})

this.register();

}

register()

{

let bodyData = {

"name" : this.name,

"marks" : this.marks

};

console.log("register data:",bodyData);

this.http.post("http://localhost:8000/save",bodyData).subscribe((bodyData: any)=>

{

console.log(bodyData);

//alert("Student Registered Successfully")

//this.getAllEmployee();

//this.name = '';

//this.marks = 0;

console.log("Result data:",bodyData)

});

this.getAllStudent();

}

delete(id:string){

//alert('Are you sure you want to delete?');

let id\_detail={"\_id":id};

console.log("Delere id is:"+id)

this.http.post("http://localhost:8000/delete/",id\_detail).subscribe(()=>{

});

console.log("Item Deleted");

this.getAllStudent();

}

}

**Step 4:** Modify app.component.html as

<div class="container mt-4" >

<div class="card">

<h1>Student Marks Information</h1>

<form method="post" action="/save">

<div class="form-group">

<label>Name</label>

<input type="text" [(ngModel)]="name" [ngModelOptions]="{standalone: true}" class="form-control" id="name" placeholder="Enter Name">

</div>

<div class="form-group">

<label>Marks</label>

<input type="text" class="form-control" [(ngModel)]="marks" [ngModelOptions]="{standalone: true}" id="marks" placeholder="Enter Address">

</div>

<button type="submit" (click)="save()" class="btn btn-primary" id="sub\_text">Submit</button>

<button type="submit" class="btn btn-primary" id="update\_text" (click)="UpdateRecords()">Update</button>

</form>

</div>

</div>

<div class="container mt-5">

<h2>Studnets Marks List</h2>

<table class="table table-striped table-bordered table-hover">

<thead class="thead-dark">

<tr>

<th>Id</th>

<th>Name</th>

<th>Marks</th>

<th>Action</th>

</tr>

</thead>

<tbody>

<tr \*ngFor="let studnet2 of StudentArray">

<td>{{ studnet2.\_id }}</td>

<td>{{ studnet2.name }}</td>

<td>{{ studnet2.marks }}</td>

<td><a href="#" (click)="delete(studnet2.\_id)">Delete</a><a href="#" (click)="setUpdate(studnet2.\_id)"> Edit</a></td>

</tr>

</tbody>

</table>

</div>

The important steps is to create server.js file. But, before that we need install few packages. We can create a different project directory for server.js, but in our application we create the server.js file inside the angular project directory.

**Step 1:** Install mongoose and mongodb for mongodb connection as

Npm install mongoose

Npm install mongodb

**Step 2**: create a file server.js as:

var express=require('express');

var server=express();

var mongo=require('mongoose');

var bodyParser=require("body-parser")

var cors = require('cors');

server.use(express.json());

//var response=require('http')

mongo.set('strictQuery', false);

mongo.connect("mongodb://127.0.0.1:27017/student",function checkDB(err){

if(err){

console.log("error");

}else{

console.log("connectedd....");

}

});

server.use(bodyParser());

server.use(bodyParser.json({limit:'5mb'}));

server.use(bodyParser.urlencoded({extended:true}));

server.use(cors());

server.use(function(req,res,next){

res.setHeader("Access-Control-Allow-Origin", "\*");

res.setHeader("Access-Control-Allow-Credentials", "true");

res.setHeader("Access-Control-Allow-Methods", "GET,HEAD,OPTIONS,POST,PUT");

res.setHeader("Access-Control-Allow-Headers", "Access-Control-Allow-Headers, Origin,Accept, X-Requested-With, Content-Type, Access-Control-Request-Method, Access-Control-Request-Headers");

next();

});

var studentSchema =new mongo.Schema({

name: {type:String},

marks:{type:Number}

});

var studModel = mongo.model('student\_details', studentSchema);

//var student =studModel.find({});

server.post('/save',function(req,res){

console.log(req.body);

var studDetails = new studModel({

name: req.body.name,

marks: req.body.marks,

});

studDetails.save(function (err) {

if (err) return handleError(err);

// saved!

});

})

server.patch('/update',function(req,res){

console.log(req.body);

const { ObjectId } = require('mongodb');

var studDetails = new studModel({

name: req.body.name,

marks: req.body.marks,

});

studDetails.updateOne({\_id:new ObjectId(req.body.\_id)},{ $set: studDetails },function (err) {

if (err) return handleError(err);

console.log("Data updated");

});

//console.log("Data updated");

})

server.get('/getAll',function(req,res){

studModel.find({},function(err, result) {

if (err) {

console.log(err);

} else {

res.json(result);

}

});

console.log("request has come");

//res.send("Request recieved");

})

server.post("/delete",function(req,res){

console.log("Inside delete")

const { ObjectId } = require('mongodb');

console.log(req.body.\_id)

studModel.remove({

\_id : new ObjectId(req.body.\_id)

}, function(err) {

if(err) { res.send(err);}

console.log("removed id");

})

})

server.listen(8000,function check(error){

if(error){

console.log("error");

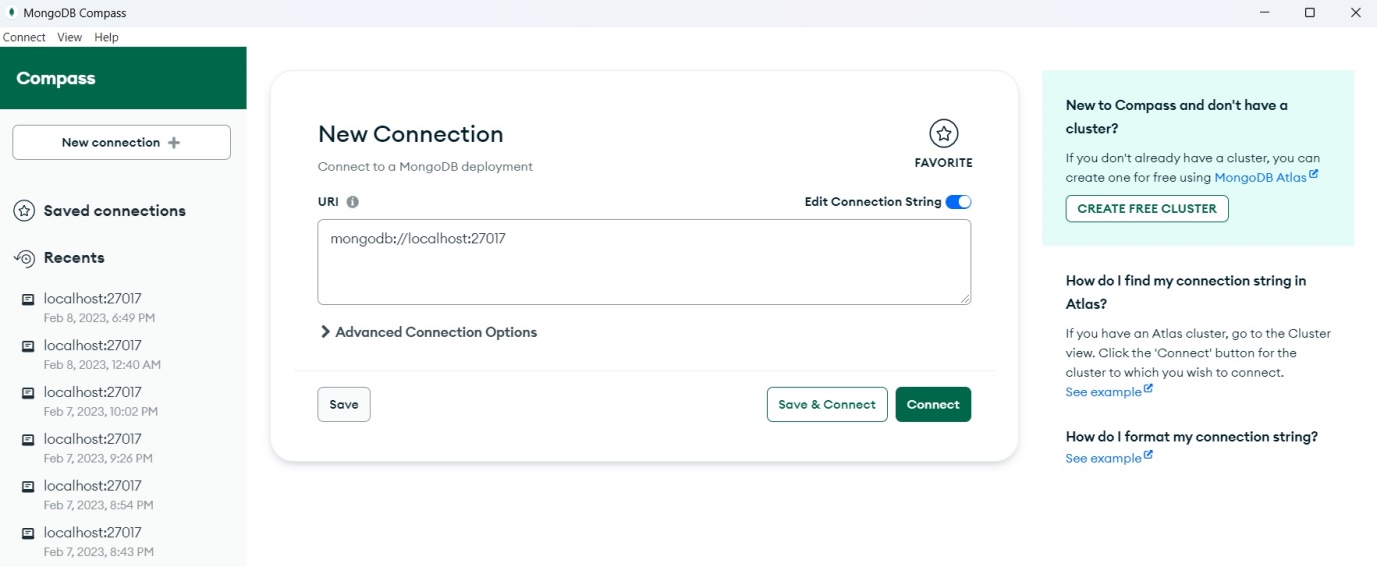
}else{

console.log("starteddddd...");

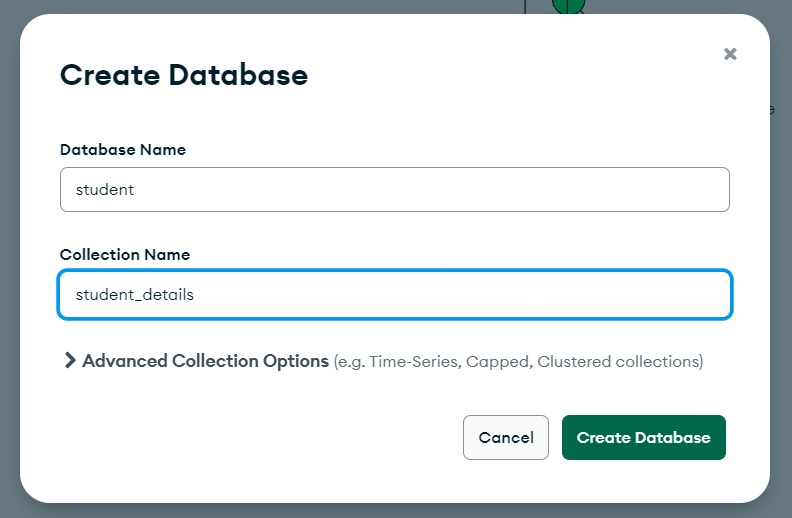
}

});

The mongodb server under windows is installed as a service. Therefore we don’t need to start every time. The database under mongodb we can create using mongoDbCompass tool.

**Step 1**: open MongoDbCompass and connect to the server

**Step 2:** After Connection goto Databases on left panel and click on the plus sign to create new datanase. In our application we will create student database.



Step 3: A dialogue box will appear. In this box fill with database name and collection name and click on create “Create Database” button. Therefore our database is created.

After creation of the database we need to execute our project. But, for that we need start three servers. The steps to execute our application is as follows:

**Step 1:** Start the node JS server as:

Node server.js

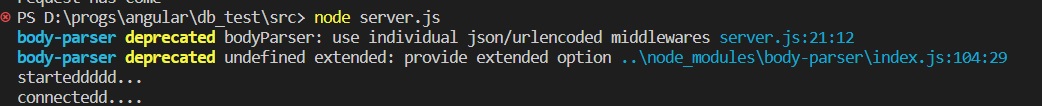
**Step 2:** start Angular server as

Ng server –open

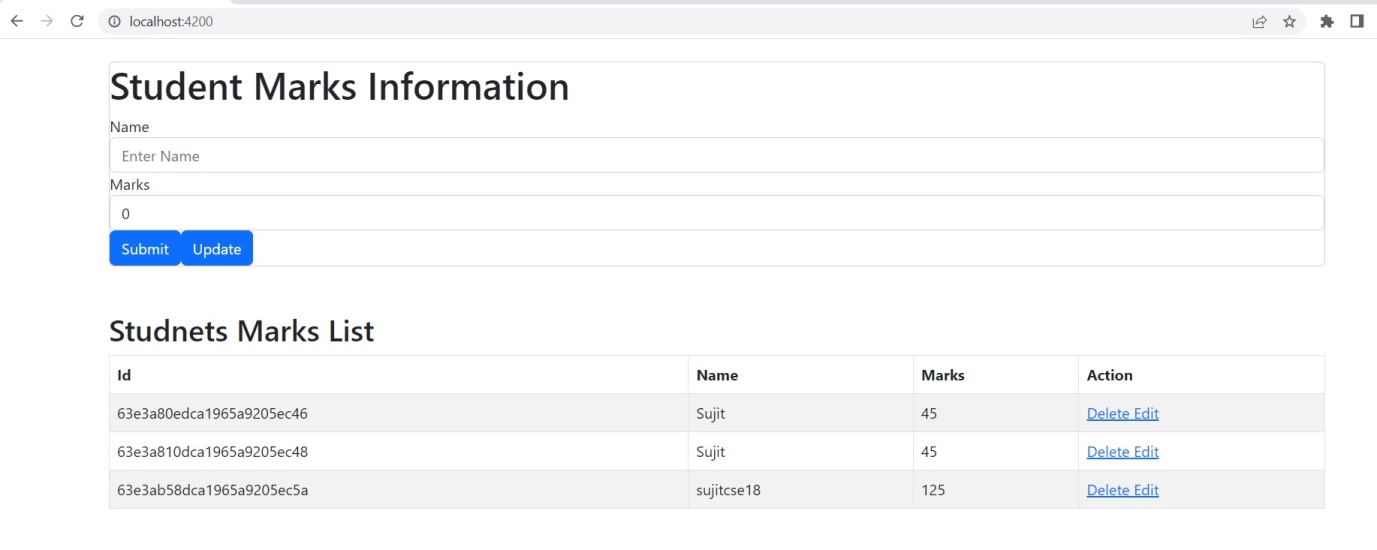
**Step 3:** We don’t need the start our database explicitly since after install with each time windows starts it start automatically.

Output:

The output for when server is started



The angular JS interface is as follows



**Exercise 12: Create a employee web application with Angular JS, node JS and mongodb.**