**MEAN STACK TECHNOLOGIES**

**LAB PROGRAMS---2024**

1. **Design a Node.js application to add the Node Backend in Mean Stack.(or) Creating a**

**Web-based Node Application:**

**Procedure:**

**Node** is an open source, cross-platform server environment which executes JavaScript.

Node helps to write front-end and back-end code in the same language. It helps to write efficient

code for real-time applications.

A web-based **Node** application consists of the following three important components:

* Import required module
* Create server
* Read Request and return response

### ****Step 1: Import required modules****

### Load Node modules using the require directive. Load http module and

### store returned HTTP instance into a variable. Now, we turn this file into a server. First, we will need

### to import a package, i.e., http package provided by the nodejs. we need to create a variable and

### import the package with the required keyword in the following way:

**Syntax:**

var http = require("http");

### ****Step 2: Creating a server in Node:****

### Create a server to listen the client’s requests. Create server

### instance using createServer() method. Bind server to port 8080 using listen method associated

### with server instance. Now, we will use the http package to create a new server. The http package

### has a createServer() method, which takes a request listener as an argument. It is a function that will

### be executed for every incoming request without worrying about the path which this request targets.

**Syntax:**

http.createServer().listen(8080);

### ****Step 3: Read request and return response in Node:****

Read the client request made using browser or console and return the response.

A function with request and response parameters is used to read client request and return response.

In createServer function, this function will receive two arguments, i.e., request and response object,

which will be passed in by nodejs. These objects offer data and utility methods that allow us to

work with requests and responses.

**Syntax:**

http.createServer(function (request, response) {...}).listen(8080);

**Program:**

**Create a firstprogram.js file containing the following code.**

// Require http header

var http = require('http');

// Create server

http.createServer(function (req, res) {

    // HTTP Status: 200 : OK

    // Content Type: text/html

    res.writeHead(200, {'Content-Type': 'text/html'});

    // Send the response body as "Hello World!"

    res.end('Hello World!');

}).listen(8080);

**Steps to run the application:**

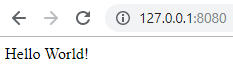
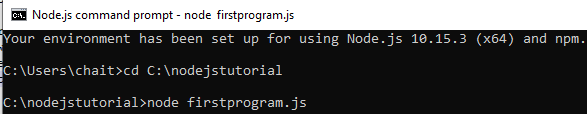
**Step 1:**To run the program type the following command in terminal

node firstprogram.js

**Step 2:**Then type the following URL in browser

http://127.0.0.1:8080/

**Output:**



1. **Design an node.js application to fetch data from Node backend in mean stack.**

We will use the following steps to fetch posts:

# Step 1:

In the first step, we have to register a path because we can fetch posts when we send a get request to that path. For this purpose, we need to get rid of the redundant middleware.

# Step 2:

We will use the middleware in our **use()** function. So, we will add another argument here. We can add as many arguments in it. Previously, we added a function as an argument that handles the request. Now, we will pass a path for which we want to filter-down, as an argument.

app.use('/api/posts', (req, res, next) =>{ res.send('Hello from express');

});

If we add /post, it means only requests targeting localhost 3000/posts will reach this middleware, and all other requests will go into the void because we have no default error handler right now. We name the path /api/posts to make it clear that it is a rest API.

# Step 3:

Now, we have to target this path to reach the code. We will return the JSON data here, so we will use the json() function. This function will return the data in JSON format. We will set the dummy data using a javascript array with the same format of posts that we have used in the front-end.

app.use('/api/posts', (req, res, next) =>{ const posts = [

];

res.json(); });

# Step 4:

We will also introduce an ID because if we had fetched these posts from a database, they would have an ID.

We might store some extra information that we don't want to pass on to the client on the backend. We will not define the post as a model in the posts array because we are not using typescript here. We define the posts like this:

const posts = [

{

id: 'snkfkjkf',

title: 'First server-side post',

content: 'This is comming from the server'

},

{

id: 'gyrrshjhk',

title: 'Second server-side post',

content: 'This is commiing from the server'

},

{

id: 'mhsetghj',

title: 'Third server-side post',

content: 'This is comming from the server'

}

# Step 5:

Now, we need to return them with our response. We can simply pass the object array to the function like this:

res.json(posts);

Now, we save it and go to the localhost:3000/api/posts, we will be able to see our JSON data.

We can also send back more complex object where we may have a message property that holds the message and then our posts property, which holds the posts array like this:

res.status(200).json({message: 'Posts Fetched Successfully', posts: posts

});

In the above code, we have added another method status and passed the value 200 as an argument for success.

Now, we save it and go to the localhost:3000/api/posts, we will be able to see our JSON data with the message and posts property like as:

**Program:**

const express=require('express');

const app=express();

app.use('/api/posts',(req,res,next)->{

const posts=[

{

id: 'snkfkjf',

title:'First server-side post',

content:'This is coming from the server'

},

{

id: 'snkfkjf',

title:'second server-side post',

content:'This is coming from the server'

},

{

id: 'snkfkjf',

title:'third server-side post',

content:'This is coming from the server'

];

res.json(200).json({

message:posts fetched successfukky',

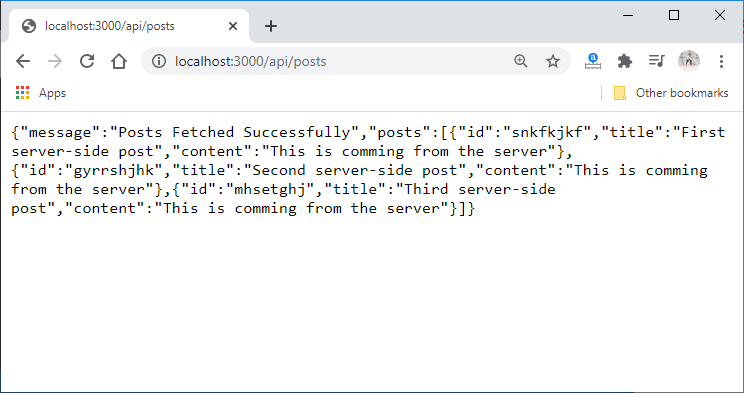
posts:posts

});

});

module.exports=app;

**Output:**



1. **Design an Angular.js application to design and using forms in Mean stack.**

**AngularJS**performs **form validation** on the client side. AngularJS monitors the state of the form and input fields (input, text-area, select), and notify the user about the current state. AngularJS also holds information about whether the input fields have been touched, modified, or not.

**Procedure:**

Step1: start

Step2: use HTML tag, with in that take Head tag.

Step3: with in the head tag use CSS styles for the elements like body, input, error , button etc.

Step3:Use body tag.

Step4: with in the body tag use form tag and the angular directives like ng-app, ng-controller , ng-

model, ng-showetc.

Step5: use <form> tag .Inside the form tag use the form elements like

textbox, password, radio buttons, Action buttons, combobox and textarea

fields properly.

Step6: Use <input> container tag and take the attributes are Name, value ,

Type and Size

Step7: We need a space without a break, use an entity called &nbsp. i.e., Non

breakable space entity.

Step8: close the form tag.

Step9: use script tag for adding javascript in html document.

Step10: with in the script tag, use angular service like $scope.

Step11: close script tag and body tags properly.

Step12: Finally close the HTML tag.

Step13: Stop.

* **Program**

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Angular Form Validation</title>

<style>

/\* CSS styles \*/

body {

font-family: Arial, sans-serif;

background-color: #f1f1f1;

display: flex;

justify-content: center;

align-items: center;

height: 100vh;

margin: 0;

}

#app {

background-color: #fff;

padding: 20px;

border-radius: 5px;

box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);

text-align: center;

max-width: 400px;

margin: auto;

}

input[type="text"], input[type="email"], input[type="password"], textarea {

padding: 10px;

margin-bottom: 10px;

border-radius: 5px;

border: 1px solid #ccc;

width: 100%;

box-sizing: border-box;

}

.error {

color: red;

margin-top: -10px;

margin-bottom: 10px;

font-size: 12px;

text-align: left;

}

button {

padding: 10px 20px;

margin-top: 10px;

border: none;

border-radius: 5px;

background-color: #007bff;

color: #fff;

cursor: pointer;

width: 100%;

}

button:disabled {

background-color: #ccc;

cursor: not-allowed;

}

</style>

</head>

<body>

<div id="app" ng-app="validationApp" ng-controller="ValidationController">

<h2>Angular Form Validation</h2>

<form name="myForm" novalidate>

<input type="text" name="name" placeholder="Name" ng-model="formData.name"

required>

<div class="error" ng-show="myForm.name.$dirty &&

myForm.name.$error.required">Name is required</div>

<input type="email" name="email" placeholder="Email" ng-model="formData.email"

required>

<div class="error" ng-show="myForm.email.$dirty &&

myForm.email.$error.required">Email is required</div>

<div class="error" ng-show="myForm.email.$dirty &&

myForm.email.$error.email">Invalid email format</div>

<input type="password" name="password" placeholder="Password"

ng-model="formData.password"

ng-minlength="8" ng-maxlength="20" required>

<div class="error" ng-show="myForm.password.$dirty &&

myForm.password.$error.required">Password is required</div>

<div class="error" ng-show="myForm.password.$dirty &&

myForm.password.$error.minlength">Password must be at least 8 characters</div>

<div class="error" ng-show="myForm.password.$dirty &&

myForm.password.$error.maxlength">Password cannot exceed 20 characters</div>

<textarea name="message" placeholder="Message" ng-model="formData.message"

rows="4" required></textarea>

<div class="error" ng-show="myForm.message.$dirty &&

myForm.message.$error.required">Message is required</div>

<input type="checkbox" name="agree" id="agree" ng-model="formData.agree" required>

<label for="agree">I agree to the terms and conditions</label>

<div class="error" ng-show="myForm.agree.$dirty &&

myForm.agree.$error.required">You must agree to the terms and conditions</div>

<button ng-click="submitForm()" ng-disabled="myForm.$invalid">Submit</button>

</form>

</div>

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>

<script>

angular.module('validationApp', [])

.controller('ValidationController', ['$scope', function($scope) {

$scope.formData = {};

$scope.submitForm = function() {

// Form submission logic goes here

alert('Form submitted successfully!');

};

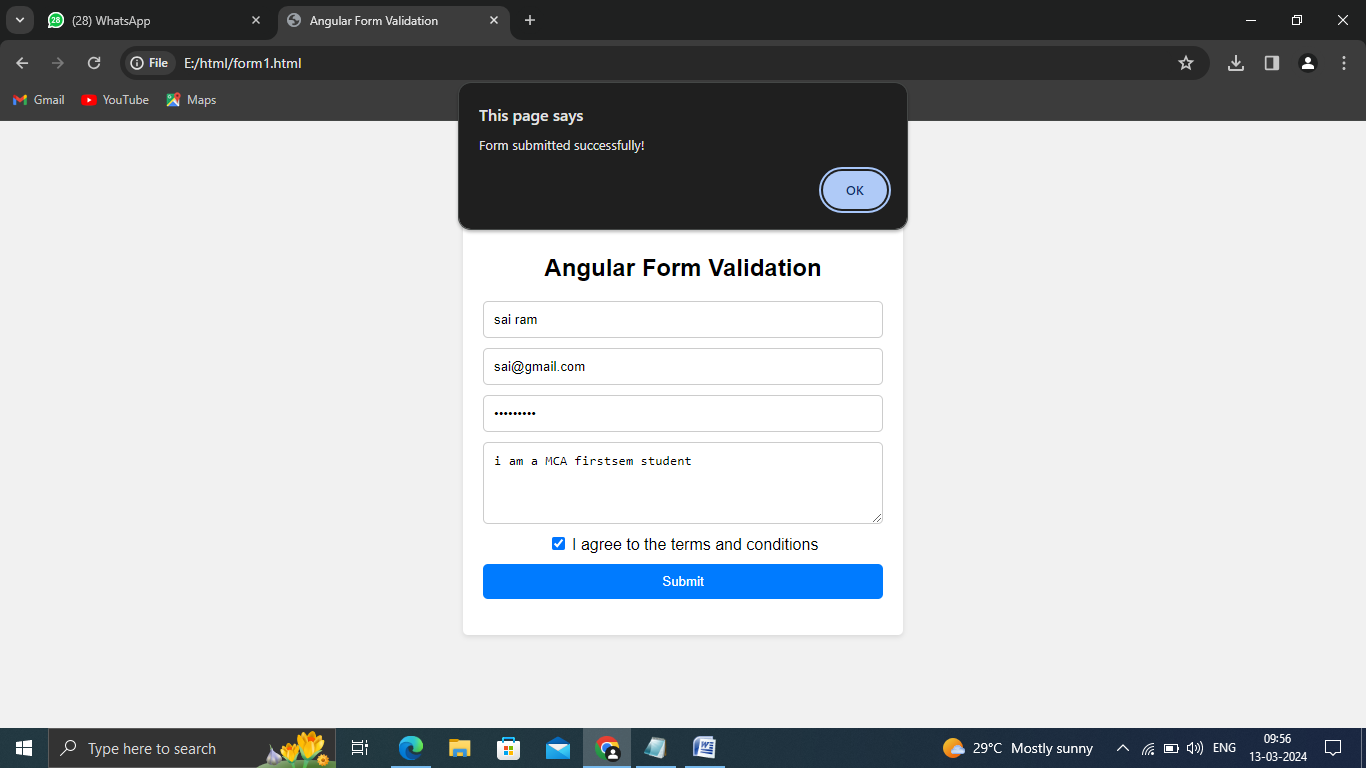
}]);

</script>

</body>

</html>

**Output:**



1. **Design an Angular.js application to implement structural directives in Mean stack.**

**Procedure:**

Structural directives are responsible for the Structure and Layout of the DOM Element. It is used to hide or display the things on the DOM. Structural Directives can be easily identified using the ‘\*’. Every Structural Directive is preceded by a ‘\*’ symbol.  
Some of the Build in Structural Directives with Examples are as follows:

**1. \*ngIf:**

ngIf is used to display or hide the DOM Element based on the expression value assigned to it. The expression value may be either true or false.

**Syntax:**

<div \*ngIf="boolean"> </div>

In the above Syntax, boolean stands for either true or false value. Hence, it leads to 2 valid syntaxes as below :

<div \*ngIf="true"> </div>

<div \*ngIf="false"> </div>

**Program:**

|  |
| --- |
| <div \*ngIf="false">    This text will be hidden    <h1 [ngStyle]="{'color':'#FF0000'}">      GFG Structural Directive Example    </h1>  </div>  <div \*ngIf="true">    This text will be displayed    <h1 [ngStyle]="{'color':'#00FF00'}">      GFG Structural Directive Example    </h1>  </div> |

**Output:**

https://media.geeksforgeeks.org/wp-content/uploads/20200712141115/structuraldirective1-300x62.png

**2. \*ngIf-else:**

ngIf-else works like a simple If-else statement, wherein if the condition is true then ‘If’ DOM element is rendered, else the other DOM Element is rendered. Angular uses ng-template with element selector in order to display the else section on DOM.

**Syntax:**

<div \*ngIf="boolean; else id\_selector"> </div>

<ng-template #id\_selector> </ng-template>

In the above Syntax, boolean stands for either true or false value. If the boolean value is true then Element in If is rendered on the DOM, else another element is rendered on the DOM .

**Program:**

|  |
| --- |
| <div \*ngIf="false;else id\_selector">    This text will be hidden    <h1 [ngStyle]="{'color':'#FF0000'}">      GFG Structural Directive      If Part    </h1>  </div>  <ng-template #id\_selector>    This text will be displayed   <h1 [ngStyle]="{'color':'#00FF00'}">      GFG Structural Directive  Else Part    </h1>  </ng-template> |

**Output:**



**3.\*ngFor:**  
\*ngFor is used to loop through the dynamic lists in the DOM. Simply, it is used to build data presentation lists and tables in HTML DOM.

**Syntax:**

<div \*ngFor="let item of item-list"> </div>

**Program:**

Consider that you are having a list as shown below:

items = ["GfG 1", "GfG 2", "GfG 3", "GfG 4"];

|  |
| --- |
| <div \*ngFor="let item of items">   <p >  {{item}} </p>  </div> |

**Output:**



**4.\*ngSwitch:**  
ngSwitch is used to choose between multiple case statements defined by the expressions inside the \*ngSwitchCase and display on the DOM Element according to that. If no expression is matched, the default case DOM Element is displayed.   
**Syntax:**

<div [ngSwitch]="expression">

<div \*ngSwitchCase="expression\_1"></div>

<div \*ngSwitchCase="expression\_2"></div>

<div \*ngSwitchDefault></div>

</div>

In the above syntax, the expression is checked with each case and then the case matching with the expression is rendered on DOM else the Default case is rendered on the DOM.

**Program:**

|  |
| --- |
| <div [ngSwitch]="'one'">    <div \*ngSwitchCase="'one'">One is Displayed</div>    <div \*ngSwitchCase="'two'">Two is Displayed</div>    <div \*ngSwitchDefault>Default Option is Displayed</div>  </div> |

In the above example, the expression ‘one’ in ngSwitch is matched to the expression in ngSwitchCase. Hence, the Element displayed on DOM is ” One is Displayed “. 

**Output:**

https://media.geeksforgeeks.org/wp-content/uploads/20200712141114/StructuralDirective4-300x63.png

1. **Design an Angular.js application for creating a tool bar using Mean stack.**

**Procedure:**

Step1: start

Step2: use HTML tag, with in that take Head tag.

Step3: with in the Head tag use title tag , for the web page title. And close title tag.

Step3:Use body tag.

Step4: with in the body tag use a property called ” bgcolor” for applying the background color the

web page.

Step5: With in the body tag use a container tag called fir <h1> for assigning the size for the

content.

Step6: use md-toolbar tag, with in that use a property called color.

Step7: with in the md-toolbar tag use an element button.

Step8: close md-toolbar tag and close body tag.

Step9: Finally close the HTML tag.

Step10: Stop.

**Program:**

<html>

<head>

<title> Angular tools</title>

</head>

<body bgcolor="yellow">

<h1 align="center"> Angular Toolbar</h1>

<md-toolbar color="#11aa33">

<button md-button class="md-primary" [routerLink]="['/basic']">

<md-icon class = "icon-20">home</md-icon> &nbsp;Angular Concepts</button>

<button md-button [mdMenuTriggerFor]="bMenu">Basic Concepts</button>

<md-menu #bMenu="mdMenu">

<button md-menu-item [routerLink]="['/a4']">Angular Component</button>

<button md-menu-item [routerLink]="['/cli ']">Angular CLI</button>

<button md-menu-item [routerLink]="['/inout']">Event Emitters</button>

<button md-menu-item [routerLink]="['/template']">Template Driven Forms</button><br>

<button md-menu-item [routerLink]="['/reactive']">Data Driven Forms</button><br>

<button md-menu-item [routerLink]="['/directives']">Angular Custom Directives</button><br>

<button md-menu-item [routerLink]="['/pipes']">Custom Pipes</button>

<button md-menu-item [routerLink]="['/viewchild']">View Child</button><br>

<button md-menu-item [routerLink]="['/view']">View Encapsulation</button><br>

</md-menu>

<button md-button [mdMenuTriggerFor]="aMenu">Advanced Concepts</button><br>

<md-menu #aMenu="mdMenu">

<button md-menu-item [routerLink]="['/ngrx']">Angular Redux using ngrx/store</button><br>

<button md-menu-item [routerLink]="['/guard']">Angular Guards</button><br>

<button md-menu-item [routerLink]="['/host']">Host & Host-Context</button><br>

</md-menu>

<button md-button (click)="openDialog()">&nbsp;Contact Card</button>

</md-toolbar>

</body>

</html>

**Output:**

****

1. **Design an Express.js application to add the express framework in mean stack.**

**Procedure:**

# Step 1: We will install express with npm by running the following command.

npm install --save express

# Step 2: We will now add the express app and all the files that belong to it in the back-end

# folder. In the backend folder, we will create an app.js file.

# Step 3: This app.js file will hold the express app, which is still a nodejs server-side app, to take advantage of these express features. We will import the express in the same way as we have imported the http package in the server.js file.

const express = require ( 'express' );

# Step 4: Now, we will use the express and one way of using it to quickly add one such route that handles a request for a single path only. We will execute express as a function, and it will return us an express app.

const app = express();

**Step5:** The middleware function takes a function that is executed for an incoming request, and that function takes three arguments, i.e., request, response, and next function. The request and response arguments are the same as nodejs, and the next function is used for one special purpose. If we execute this function, then the request will actually continue its journey like this:

app.get('/', (req, res) => {

res.send('Hello Express World!')

})

**Step 6:** One other thing is important before passing app to our createServer function, we need to tell express on which port we are working. For this, we need to set the port key using the set function in the following way:

app.listen(port, () => {

console.log(`Example app listening on port ${port}`)

})

**Program:**

**const express = require('express')**

**const app = express()**

**const port = 3000**

**app.get('/', (req, res) => {**

**res.send('Hello Express World!')**

**})**

**app.listen(port, () => {**

**console.log(`Example app listening on port ${port}`)**

**})**

### Running Locally

First create a directory named myapp, change to it and run npm init. Then install express as a dependency, as per the [installation guide](https://expressjs.com/en/starter/installing.html).

In the myapp directory, create a file named app.js and copy in the code from the example above.

The req (request) and res (response) are the exact same objects that Node provides, so you can invoke req.pipe(), req.on('data', callback), and anything else you would do without Express involved.

**Run the app with the following command:**

$ node app.js

Then, load http://localhost:3000/ in a browser to see the output.

**Output:**

**'Hello Express World!**

1. **Design an application using angular Http client in Mean stack.**

**Procedure:**

Step1: start

Step2: create service file for Http client, with in the service file use import statements

Step3: create

export class UserService {

private baseUrl = 'https://api.example.com/users';

constructor(private http: HttpClient)

Step4: use getUser(0 methods for getting the details of userid and user data.

Step5: Next create HTML file

Step6: with in the HTML file use div tag.

Step7: with in the div tag use user profile.

Step8:Close div tag.

Step9: Again starts a div tag with in that use a directive called ng-if

Step10: with in the div tag use label tag and input tag and enter the user input fields.

Step11: close div tag

Step12: create script file, with in the script file, use import statements

Step13: write output statement like console.log for displaying the results.

Step14:Stop

**Program:**

# Service ﬁle:

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

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

@Injectable({ providedIn: 'root'

})

export class UserService {

private baseUrl = 'https://api.example.com/users'; constructor(private http: HttpClient) { }

getUser(userId: string): Observable<any> {

return this.http.get<any>(`${this.baseUrl}/${userId}`);

}

updateUser(userId: string, userData: any): Observable<any> { return this.http.put<any>(`${this.baseUrl}/${userId}`, userData);

}

}

## Html File:

<div>

<h2>User Profile</h2>

<p>Name: {{ user.name }}</p>

<p>Email: {{ user.email }}</p>

<button (click)="editProfile()">Edit Profile</button>

</div>

<div \*ngIf="editing">

<h2>Edit Profile</h2>

<label>Name:</label>

<input type="text" [(ngModel)]="updatedUser.name">

<label>Email:</label>

<input type="email" [(ngModel)]="updatedUser.email">

<button (click)="saveChanges()">Save Changes</button>

</div>

## Typescript file:

import { Component, OnInit } from '@angular/core'; import { UserService } from './user.service';

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

templateUrl: './profile.component.html'

})

export class ProfileComponent implements OnInit { user: any;

editing: boolean = false; updatedUser: any = {};

constructor(private userService: UserService) { } ngOnInit(): void {

this.fetchUser();

}

fetchUser(): void { this.userService.getUser('user-id').subscribe( user => {

this.user = user;

// Clone user object for editing this.updatedUser = { ...user };

},

error => {

console.error('Error fetching user data:', error);

});}

editProfile(): void { this.editing = true;

}

saveChanges(): void {

this.userService.updateUser('user-id', this.updatedUser).subscribe( updatedUser => {

console.log('Profile updated successfully:', updatedUser); this.user = updatedUser; // Update displayed user information this.editing = false; // Exit editing mode

},

error => {

console.error('Error updating profile:', error);

});}}

## Output:

User Profile

Name: John Doe

Email: j[ohn@example.com](mailto:ohn@example.com)

Edit Profile

1. **Design an application which connects Node Express app to MongoDB.**

**Procedure:**

## Step 1: Node.js Database Interaction

## Step 2: MongoDB Node.js Driver Installation

Step: 2.1:To use MongoDB with Node.js, you will need to install the mongodb package in

your Node.js project.

Step 2.2: Use the following command in your terminal to install the mongodb package:

npm install mongdb

Step 2.3: We can now use this package to connect to a MongoDB database.

Step 2.4: Create an index.js file in your project directory.

const MongoClient = require('mongodb').MongoClient;

## Step 3: Connection String

Step 3.1: In order to connect to our MongoDB database,

Step 3.2: Go to **Database** then click the **CONNECT** button on your Cluster.

Step 3.3: Choose **Connect your application** then copy your connection string.

Step 3.4 : You will need to replace the <username>, <password>, and <cluster.string> with your MongoDBusername, password, and cluster string.

**Step 4**: To display the results like successfully connected use a statement called

Console.log.

**Step 5:** stop.

**Program:**

const MongoClient = require('mongodb').MongoClient;

// Connection URI

const uri = 'mongodb://localhost:27017';

// Database Name

const dbName = 'testdb';

// Connect to the MongoDB server

MongoClient.connect(uri, { useNewUrlParser: true, useUnifiedTopology: true }, (err, client) => { if (err) {

console.error(err); return;

}

console.log('Connected successfully to server');

const db = client.db(dbName);

// Insert a document into the users collection

db.collection('users').insertOne({ name: 'John Doe', email: 'john@example.com' }, (err, result)

=> {

if (err) {

console.error('Error inserting document:', err); return;

}

console.log('Document inserted successfully');

// Fetch all documents from the users collection db.collection('users').find({}).toArray((err, docs) => {

if (err) {

console.error('Error fetching documents:', err); return;

});

});

});

}

console.log('Fetched documents:', docs); client.close();

**output:**

Connected successfully to server Document inserted successfully

Fetched documents: [ { \_id: ObjectId("..."), name: 'John Doe', email: 'john@example.com' } ]

1. **Design an application which is used to update a document in MongoDb using node.js and express.js.**

**Procedure:**

### Step 1: Connect to your MongoDB instance.

### Step 2: Get the database and collection.

Step 2.1: Switch to the database and collection you wish to work with. In this case you will

be using the sample\_guides database and comets collection.

crud-update.js

|  |
| --- |
| Step 2.2:// *database and collection code goes here* |
| const db = client.db(**"sample\_guides"**); |
| const coll = db.collection(**"comets"**); Step3: Update all documents in the comets collection. Now, you have a requirement to update those documents. All fields are in metric units except for the radius, which is in imperial. The below example uses an update document to convert the radius field from the metric system to the imperial system in all documents. |

### Conversion

1 mile = 1.60934 kilometers

Step 4: **update code goes here**

const filter = { };

const updateDoc = {

$mul: {

radius: 1.60934

}};

const result = await coll.updateMany(filter, updateDoc);

### Step 5: Display your results.

Many write operations in MongoDB return a result object that contains information about the operation.For update operations, the result object will contain the modified count of documents the driver successfully updated. Here, you access this property and print them.

console.log(**"Number of documents updated: "** + result.modifiedCount);

**Step6:** Stop

**Program:**

const { MongoClient } = require("mongodb");

// Replace the uri string with your MongoDB deployment's connection string.

const uri =

"mongodb+srv://<user>:<password>@<cluster-url>?retryWrites=true&writeConcern=majority";

const client = new MongoClient(uri);

async function run() {

try {

await client.connect();

// database and collection code goes here

const db = client.db("sample\_guides");

const coll = db.collection("comets");

// update code goes here

const filter = {};

const updateDoc = {

$mul: {

radius: 1.60934,

},

};

const result = await coll.updateMany(filter, updateDoc);

// display the results of your operation

console.log("Number of documents updated: " + result.modifiedCount);

} finally {

// Ensures that the client will close when you finish/error

await client.close();

}

}

run().catch(console.dir);

**Output:**

Number of documents updated: 3

1. **Design an application which is used to delete document from MongoDb using Node.js and Express.js**

**Procedure:**

### Step 1: Connect to your MongoDB instance.

### Step 2: Get the database and collection.

Step 2.1: Switch to the database name and uri.

// Connection URI

const uri = 'mongodb://localhost:27017';

// Database Name

const dbName = 'testdb';

**Step 3:** // Delete the user document with email "[john@example.com](mailto:john@example.com)" db.collection('users').deleteOne({ email: 'john@example.com' },

(err, result) => {

if (err) {

console.error('Error deleting document:', err); return;}

### Step 4: Display your results.

Many write operations in MongoDB return a result object that contains information about the operation.For delete operations, the result object will contain the modified count of documents the driver successfully updated. Here, you access this property and print them.

console.log('Document deleted successfully');

**Step5 :** Stop

**Program:**

const MongoClient = require('mongodb').MongoClient;

// Connection URI

const uri = 'mongodb://localhost:27017';

// Database Name

const dbName = 'testdb';

// Connect to the MongoDB server

MongoClient.connect(uri, { useNewUrlParser: true, useUnifiedTopology: true }, (err, client) => { if (err) {

console.error(err); return;

}

console.log('Connected successfully to server'); const db = client.db(dbName);

// Delete the user document with email "[john@example.com](mailto:john@example.com)" db.collection('users').deleteOne({ email: 'john@example.com' }, (err, result) => {

if (err) {

console.error('Error deleting document:', err); return;

});

});

}

console.log('Document deleted successfully'); client.close();

**Output:**

Connected successfully to server Document deleted successfully

1. **Describe the structure and functionality of the Angular Calculator application, including its HTML, CSS, and AngularJS components.**

**Procedure:**

Step1: start

Step2: use HTML tag, with in that take Head tag.

Step3: with in the head tag use CSS styles for the elements like body, h1, calculator, expression

button etc.

Step4:Use body tag.

Step5: with in the body tag use the angular directives like ng-app, ng-controller , ng-model.

Step6: with in the body tag use div tag, with in the div tag contains a class called buttons.

Step7: We need a space without a break, use an entity called &nbsp. i.e., Non

breakable space entity.

Step 8: with in the button class contain a directive called ng-click.

Step 9: close div tag’

Step10: use script tag for adding javascript in html document.

Step11: with in the script tag, use angular service like $scope.

Step12: close script tag and body tags properly.

Step13: Finally close the HTML tag.

Step14: Stop.

**Program:**

<html>

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Angular Calculator</title>

<style>

/\* CSS styles \*/ body {

font-family: Arial, sans-serif; background-color: #f1f1f1; margin: 0;

padding: 0; display: flex;

flex-direction: column; justify-content: center; align-items: center;

}

h1 {

margin-bottom: 20px;

}

#calculator {

background-color: #fff; border-radius: 10px;

box-shadow: 0 4px 8px rgba(0, 0, 0,

0.1); padding: 20px; text-align: center;

}

#expression {

width: calc(100% - 40px); height: 40px;

font-size: 24px; margin-bottom: 20px; padding: 10px;

border: 1px solid #ccc; border-radius: 5px;

outline: none;

}

.buttons { display: grid;

grid-template-columns: repeat(4, 1fr); gap: 10px;

margin-top: 10px;

}

button {

width: 50px; height: 50px; font-size: 20px; border: none;

background-color: #007bff; color: #fff;

border-radius: 5px; cursor: pointer;

transition: background-color 0.3s;

}

button:hover {

background-color: #0056b3;

}

</style>

</head>

<body>

<h1>Calculator App Using Angular</h1>

<div id="calculator" ng-app="calculatorApp"

ng-controller="CalculatorController"> <input type="text" id="expression" ng-model="expression" disabled>

<div class="buttons">

<button ng-click="appendToExpression('7')">7</button>

<button ng-click="appendToExpression('8')">8</button>

<button ng-click="appendToExpression('9')">9</button>

<button ng-click="appendToExpression('+')">+</button>

<button ng-click="appendToExpression('4')">4</button>

<button ng-click="appendToExpression('5')">5</button>

<button ng-click="appendToExpression('6')">6</button>

<button ng-click="appendToExpression('-')">-</button>

<button ng-click="appendToExpression('1')">1</button>

<button ng-click="appendToExpression('2')">2</button>

<button ng-click="appendToExpression('3')">3</button>

<button ng-click="appendToExpression('\*')">\*</button>

<button ng-click="appendToExpression('0')">0</button>

<button ng-click="appendToExpression('.')">.</button>

<button ng-click="calculateResult()">=</button>

<button ng-click="appendToExpression('/')">/</button>

<button ng-click="clearExpression()">C</button>

</div>

</div>

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>

<script>

// Angular JavaScript angular.module('calculatorApp', [])

.controller('CalculatorController', function($scope) {

$scope.expression = '';

$scope.appendToExpression = function(char) {

$scope.expression += char;

};

$scope.calculateResult = function() { try {

$scope.expression = eval($scope.expression);

} catch (error) {

$scope.expression = 'Error';

}

};

$scope.clearExpression = function() {

$scope.expression = '';

};

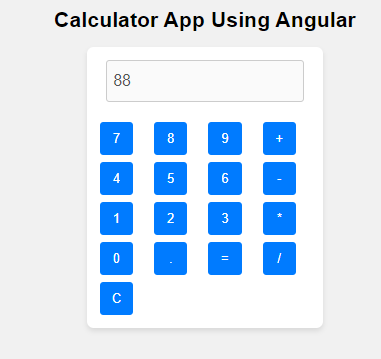
});

</script>

</body>

</html>

**Output:**

****

1. **Describe the structure and functionality of the Angular ToDo List application, including its HTML, CSS, and AngularJS components.**

**Procedure:**

Step 1: start

Step 2: use HTML tag, with in that take Head tag.

Step 3: with in the head tag use CSS styles for the elements like body, input, error , button etc.

Step 4: close head tag.

Step 5:Use body tag.

Step 6: with in the body tag use the angular directives like ng-app, ng-controller , ng-

model, ng-cick etc.

Step 7: with in the body tag use a div tag.

Step 8: use a list tag called unordered list tag. The ul tag contains a sub tag called list item tag

called “ li”.

Step 9: We need a space without a break, use an entity called &nbsp. i.e., Non

breakable space entity.

Step 10: close the ul tag and div tag.

Step 11: use script tag for adding javascript in html document.

Step12: with in the script tag, use angular service like $scope.

Step 13 close script tag and body tags properly.

Step 14: Finally close the HTML tag.

Step 15: Stop.

**Program:**

<html>

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0"> <title>Angular ToDo List</title>

<style>

/\* CSS styles \*/ body {

font-family: Arial, sans-serif; background-color: #f1f1f1; display: flex;

justify-content: center; align-items: center; height: 100vh; margin: 0;

}

#app {

background-color: #fff; padding: 20px;

border-radius: 5px;

box-shadow: 0 2px 5px rgba(0, 0, 0,

0.1); text-align: center;

}

input[type="text"] { padding: 10px; margin-bottom: 10px; border-radius: 5px;

border: 1px solid #ccc; width: 200px;

}

ul {

list-style-type: none; padding: 0;

text-align: left;

}

li {

background-color: #f9f9f9; padding: 10px;

margin-bottom: 5px; border-radius: 3px;

display: flex;

justify-content: space-between;

}

button {

padding: 5px;

background-color: #007bff; color: #fff;

border: none; border-radius: 3px; cursor: pointer;

}

button:hover {

background-color: #0056b3;

}

</style>

</head>

<body>

<div id="app" ng-app="todoApp" ng-controller="TodoController">

<h2>Angular ToDo List</h2>

<input type="text" placeholder="Add new task" ng-model="newTask" ng-keyup="$event.keyCode === 13 && addTask()">

<button ng-click="addTask()">Add Task</button>

<ul>

<li ng-repeat="task in tasks track by $index">

<span>{{ task }}</span>

<button ng-click="deleteTask($index)">Delete</button>

</li>

</ul>

</div>

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>

<script>

// TypeScript (Angular JavaScript) angular.module('todoApp', [])

.controller('TodoController', ['$scope', function($scope) {

$scope.tasks = [];

$scope.addTask = function() {

if ($scope.newTask.trim() !== "") {

$scope.tasks.push($scope.newTask);

$scope.newTask = "";

}

};

$scope.deleteTask = function(index) {

$scope.tasks.splice(index, 1);

};

}]);

</script>

</body>

</html>

**Output:**

