1. Hari Prasad

Task 1

ECMAScript is the standard that JavaScript programming language uses. ECMAScript provides the specification on how JavaScript programming language should work.

NODE JS Version-v20.5.0

Task 1

Node.js is an open-source, cross-platform JavaScript runtime environment that allows developers to run JavaScript code on the server side.

Task 3

In Node.js, you have the option to employ the `console.log()` function to exhibit output within the console. This functionality closely resembles the `console.log()` function found in JavaScript for web browsers.

console.log("Hello, Node.js!");

Task 4

A REPL, which stands for "Read-Eval-Print Loop," refers to an interactive programming environment. It permits you to input commands or fragments of code, have them promptly assessed, and instantly view the outcomes. Normally, a REPL environment offers a prompt where you can enter your code. Once you press Enter, the environment evaluates the code, displays the outcome, and subsequently awaits your next input.REPL provides an interactive and immediate way to work with JavaScript code, which can be very useful for quick testing and experimentation.

Task 5

Npm stands for "Node Package Manager." It is a package manager for Node.js, which means it's a tool that allows developers to easily manage and install third-party libraries, frameworks, and tools that are needed to build and run Node.js applications.

npm install express

This command will download and install the latest version of Express.js and its dependencies into your project directory. After the installation is complete, you can check the node\_modules directory in your project folder. This directory contains the installed packages, including Express.

Task 6

List of commonly used command-line (CL) options available in Node.js:

1. `--help` or `-h`: Display help information and available command-line options.

2. `--version` or `-v`: Display the Node.js version.

3. `--inspect` or `--inspect-brk`: Enable the Node.js Debugger for debugging purposes.

4. `--max-old-space-size=<size>`: Set the maximum heap size for the V8 JavaScript engine.

5. `--max-new-space-size=<size>`: Set the maximum size for the "new space" memory in V8.

6. `--experimental-modules`: Enable support for ECMAScript modules in the experimental stage.

7. `--expose-gc`: Make the `gc()` function available in the global scope for manual garbage collection.

8. `--no-deprecation`: Suppress deprecation warnings in the console.

9. `--trace-warnings`: Display stack traces for Node.js process warnings.

10. `--trace-event-categories`: Specify trace event categories to include in the tracing output.

11. `--tls-min-v1.0`, `--tls-max-v1.3`, etc.: Specify the minimum or maximum supported version of TLS.

12. `--v8-options`: Display available V8 engine command-line options.

13. `--icu-data-dir=<path>`: Specify the path to the ICU data directory for internationalization.

14. `--use-openssl-ca`: Use OpenSSL's default CA store instead of Node.js's bundled CA store.

15. `--no-warnings`: Suppress all process warnings.

16. `--inspect-port=<port>`: Specify the port to use for the inspector when using `--inspect`.

17. `--inspect-brk-port=<port>`: Specify the port to use for the inspector when using `--inspect-brk`.

18. `--preserve-symlinks`: Preserve symbolic links when resolving modules.

19. `--enable-source-maps`: Enable source map support for stack traces.

20. `--experimental-worker`: Enable support for Worker threads in the experimental stage.

Task 7

Web application architecture refers to the structural design and organization of components, modules, and technologies that collectively form a web application. It encompasses the high-level design decisions and patterns that guide how different parts of the application interact and work together to achieve the desired functionality, performance, and scalability.

Web application architecture decisions impact various aspects of the application, including performance, scalability, security, maintainability, and user experience. The choice of architecture depends on the project's requirements, team expertise, and long-term goals.

Some common architectural patterns and considerations for building web applications in Node.js:

1. Client-Server Architecture

2. Microservices Architecture

3. API-Driven Architecture

4. Real-Time Applications

5. Single Page Applications (SPAs)

6. Server-Side Rendering (SSR)

7. Event-Driven Architecture

8. Serverless Architecture

Task 8

const http = require('http');

const server = http.createServer((req, res) => {

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

res.write('<h1>Hello, Node.js Web Server!</h1>');

res.end();

});

const PORT = 3000;

server.listen(PORT, () => {

console.log(`Server is listening on port ${PORT}`);

});

Task 10

var http = require("http");

http.createServer(function (request, response) {

// Send the HTTP header

// HTTP Status: 200 : OK

// Content Type: text/plain

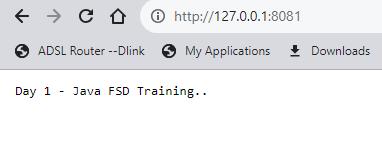
response.writeHead(200, {'Content-Type': 'text/plain'});

response.end( ' Danode Hari.jy 1 - Java FSD Training..\n');

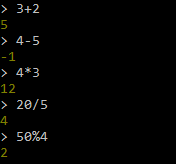
}).listen(8081);

// Console will print the message

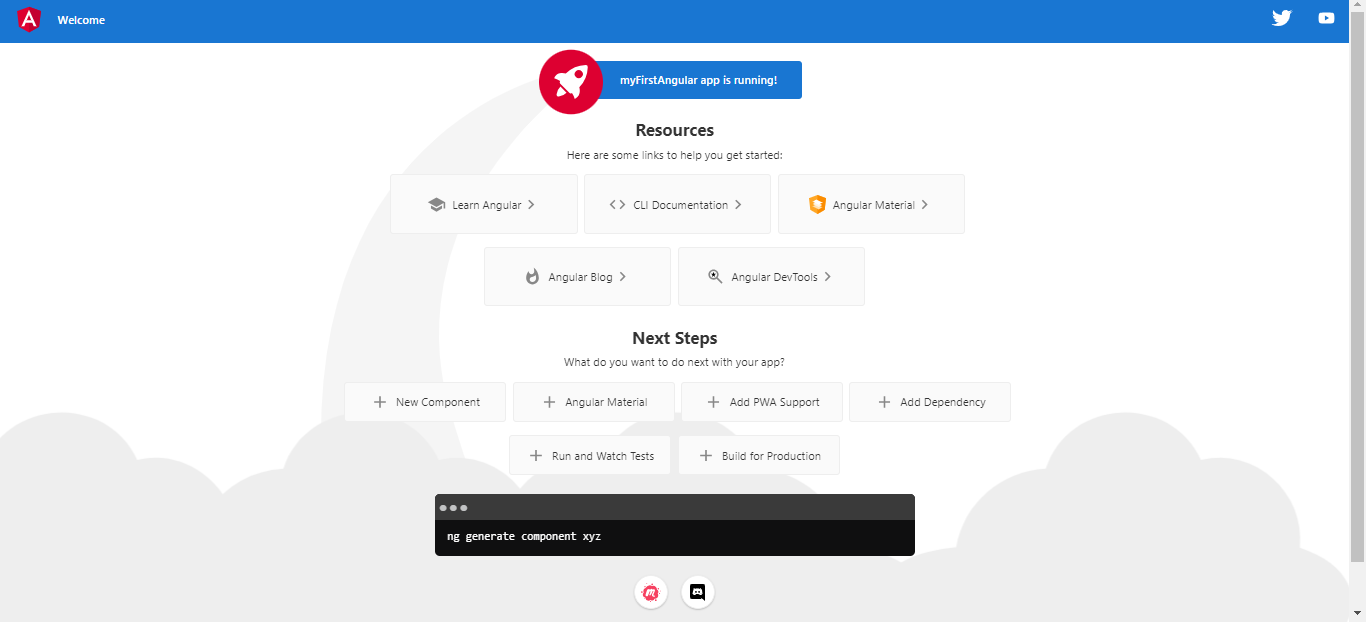
console.log('Server running at http://127.0.0.1:8081/');

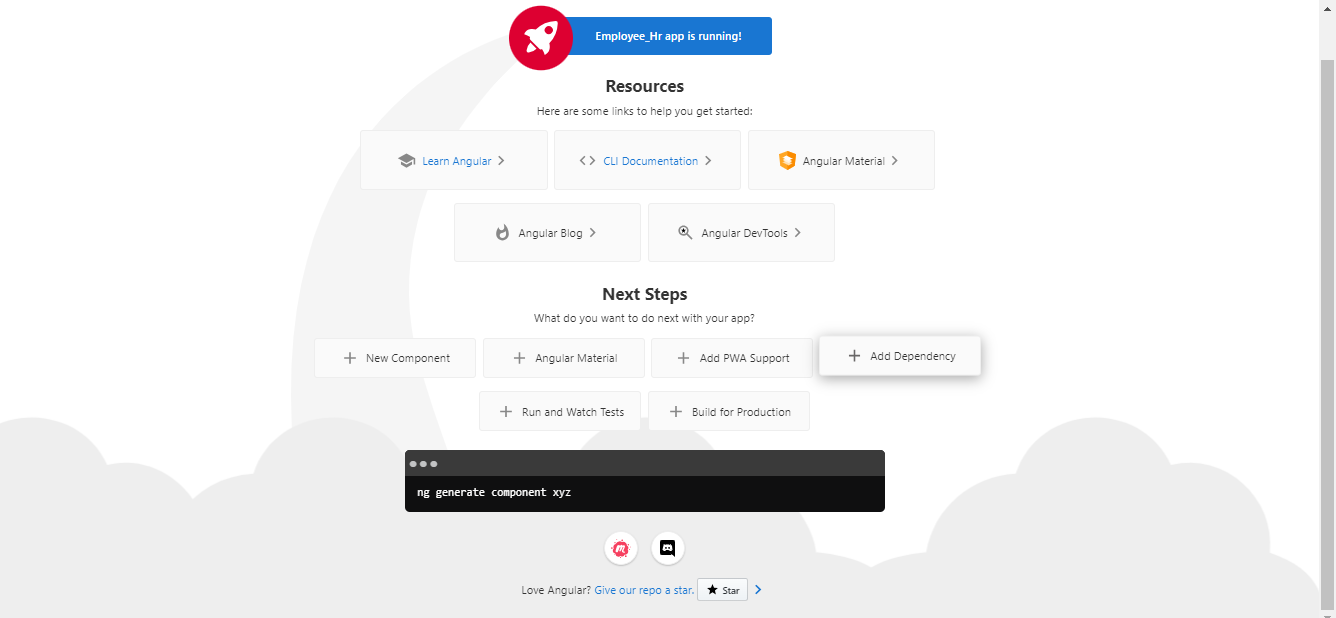


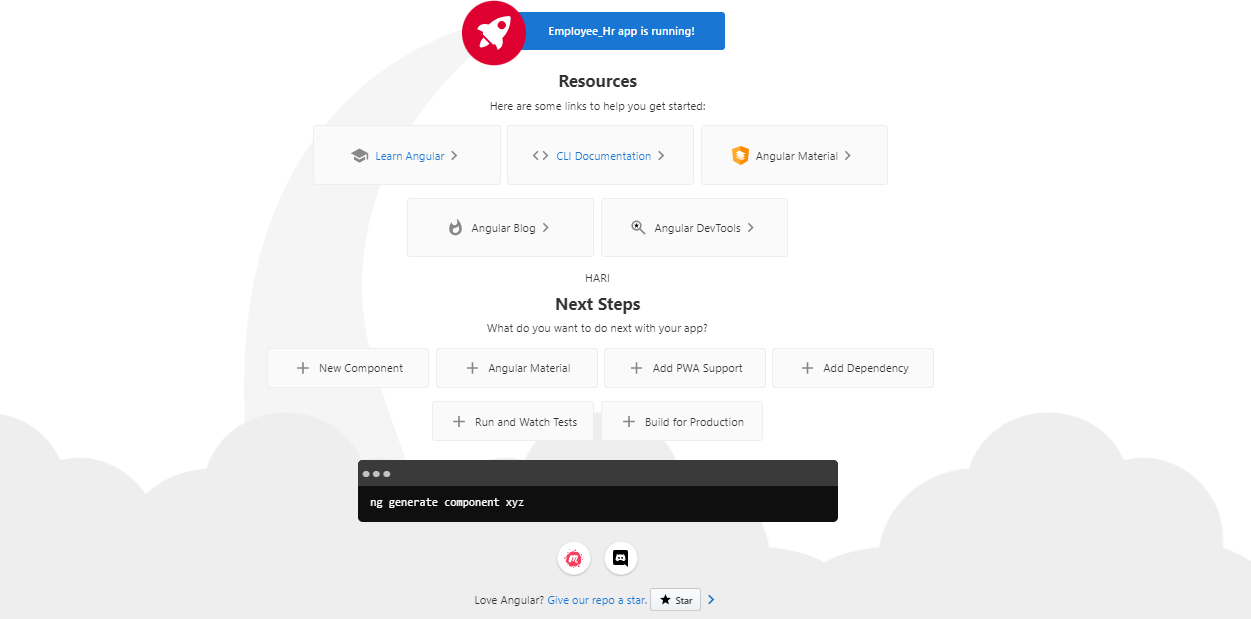
Five calculation on node js

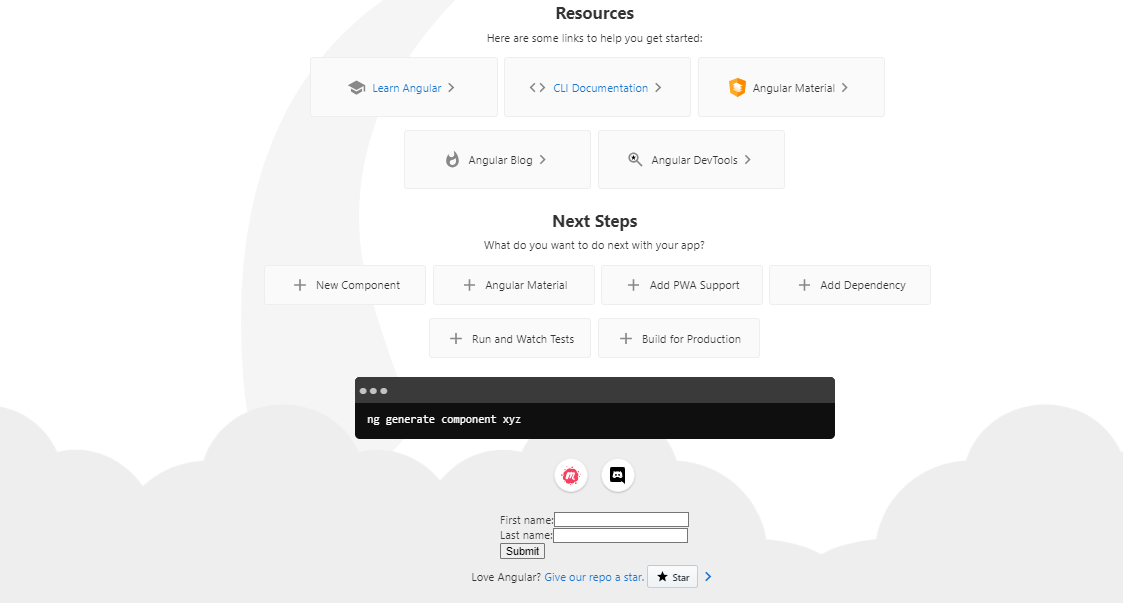


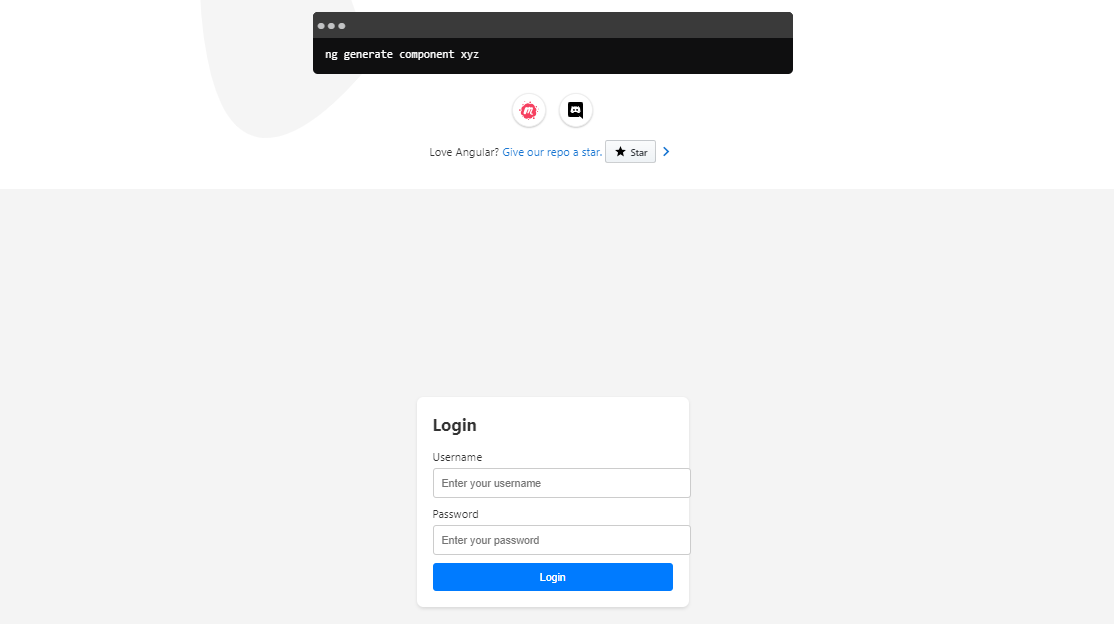
http://localhost:4200/

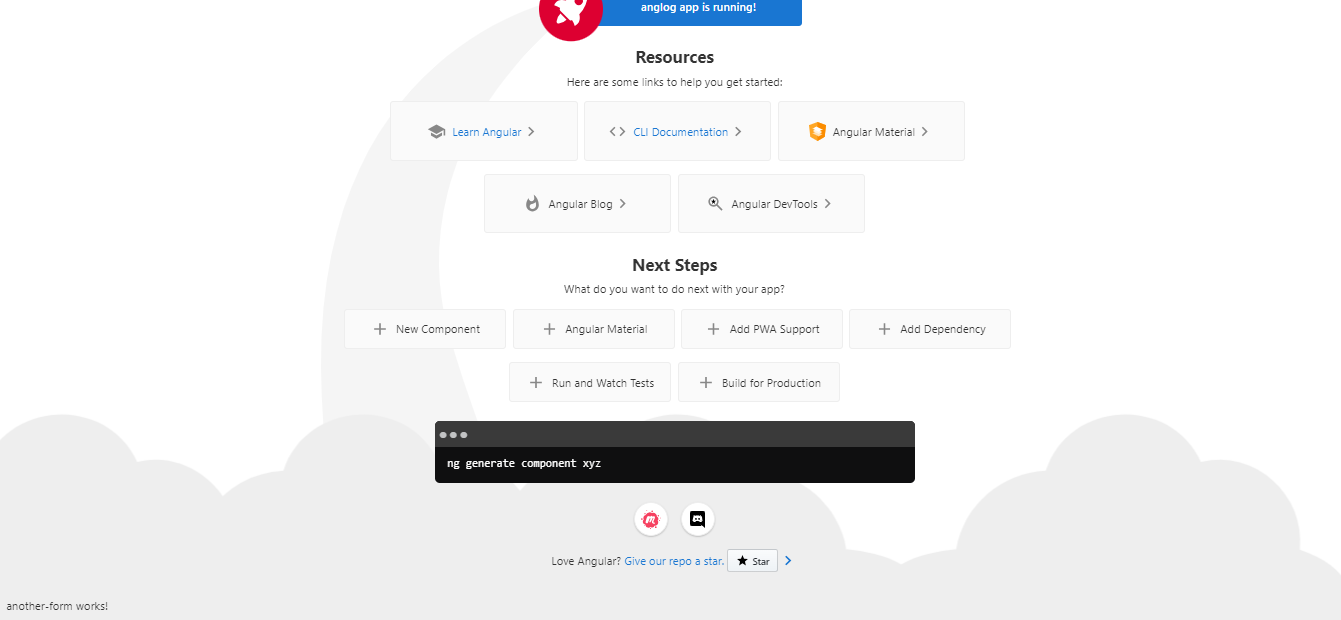


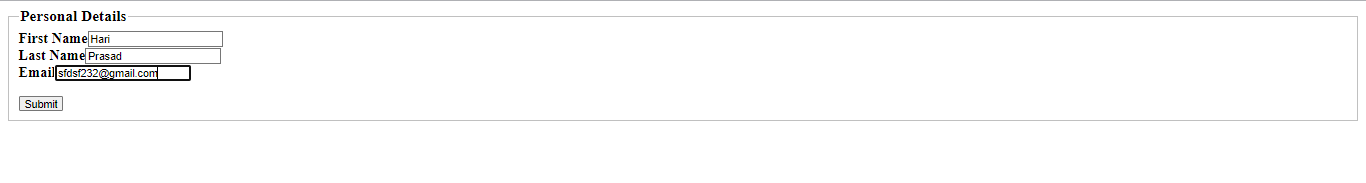


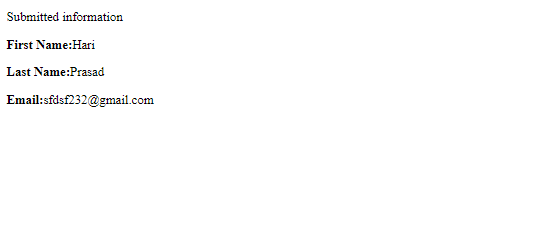












**personal-form.component.html**

<form (ngSubmit)="onSubmit()" #myForm="ngForm">

    <fieldset>

        <legend style="font-size: larger;"><b>Personal Details</b></legend>

        <label for="fname" style="font-size: large;"><b>First Name</b></label>

        <input type="text" id="fname" name="fname" required [(ngModel)]="formData.fname">

  <div \*ngIf="myForm.controls['fname'].invalid && (myForm.controls['fname'].dirty || myForm.controls['fname'].touched)">

  <div \*ngIf="myForm.controls['fname'].errors"> Name is required</div>

  </div>

  <br>

  <label for="lname" style="font-size: large;"><b>Last Name</b></label>

        <input type="text" id="lname" name="lname" required [(ngModel)]="formData.lname">

  <div \*ngIf="myForm.controls['lname'].invalid && (myForm.controls['lname'].dirty || myForm.controls['lname'].touched)">

  <div \*ngIf="myForm.controls['lname'].errors"> Name is required</div>

  </div>

  <br>

        <label for="email" style="font-size: large;"><b>Email</b></label>

        <input type="email" id="email" name="email" required [(ngModel)]="formData.email">

        <div \*ngIf="myForm.controls['email'].invalid && (myForm.controls['email'].dirty || myForm.controls['email'].touched)">

          <div \*ngIf="myForm.controls['email'].errors"> Email is required</div>

          <div \*ngIf="myForm.controls['email'].errors"> Email format wrong</div>

          </div>

        <br>

        <br>

  <button type="submit"[disabled]="!myForm.valid">Submit</button>

    </fieldset>

  </form>

  <router-outlet></router-outlet>

**personal-form.component.ts**

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

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

@Component({

  selector: 'app-personal-form',

  templateUrl: './personal-form.component.html',

  styleUrls: ['./personal-form.component.css']

})

export class PersonalFormComponent {

  formData={

    fname:'',

    lname:'',

    email:''

  };

  title = 'FormValidation';

myForm: any;

constructor(private router: Router){}

onSubmit(){

this.router.navigate(['/submitted'],{

  state:{formData:this.formData}

});

}

}

Submitted Component -(Displays the value submitted in this component)

**submitted.component.html**

<p>Submitted information</p>

<p><strong>First Name:</strong>{{submittedData.fname}}</p>

<p><strong>Last Name:</strong>{{submittedData.lname}}</p>

<p><strong>Email:</strong>{{submittedData.email}}</p>

**submitted.component.ts**

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

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

@Component({

  selector: 'app-submitted',

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

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

})

export class SubmittedComponent implements OnInit{

submittedData:any;

constructor(private router:Router){

  const nav = this.router.getCurrentNavigation();

if(nav && nav.extras.state){

this.submittedData = nav.extras.state['formData'];

}

else{

}

}

  ngOnInit() {

}

}

**app-routing.module.ts**

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

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

import { SubmittedComponent } from './submitted/submitted.component';

import { PersonalFormComponent } from './personal-form/personal-form.component';

const routes: Routes = [

  {path:'',redirectTo:'/form',pathMatch:'full'},

  {path:'submitted',component:SubmittedComponent},

  {path:'form',component:PersonalFormComponent

}

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule]

})

export class AppRoutingModule { }

**app.component.html**

<router-outlet></router-outlet>

**app.module.ts**

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

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

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

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

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

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

import { SubmittedComponent } from './submitted/submitted.component';

import { PersonalFormComponent } from './personal-form/personal-form.component';

@NgModule({

  declarations: [

    AppComponent,

    SubmittedComponent,

    PersonalFormComponent,

  ],

  imports: [

    BrowserModule,

    FormsModule,

    AppRoutingModule,

    ReactiveFormsModule,FormsModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }