

Practical No.1: Setup environment for Angular framework by Installing Node.js, npm package manager using editor like Visual Code.

A. Objective: Setting up environment for first time execution for angular practicals is important task before we start our angular practical's programming. It consist of many installations like visual code editor, Node.js and npm package manager. Once it is installed we are ready to use our setup environment to make project in angular.

B. Expected Program Outcomes (POs)

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the digital electronics engineering problems.
2. **Design/ development of solutions:** Design solutions for digital electronics engineering well-defined technical problems and assist with the design of systems components or processes to meet specified needs.
3. **Engineering Tools, Experimentation and Testing:** Apply modern digital electronics engineering tools and appropriate technique to conduct standard tests and measurements.

C. Expected Skills to be developed based on competency:

1. Install visual code open source software.
2. Setup environment for angular practical execution using node js and npm package manager.

D. Expected Course Outcomes(Cos)

Prepare environment for angular project using Node.js, npm and visual code editor.

E. Practical Outcome(PRo)

Setup environment for angular practical execution with NODE JS and NPM Package manager.

F. Expected Affective domain Outcome(ADos)

1. Follow Coding standards and practices.
2. Maintain tools and equipment.
3. Follow safety practices.
4. Follow ethical practices

G. Prerequisite Theory:

Angular is basically is an open-source, JavaScript-based client-side framework that helps us to develop a web-based application. Actually, Angular is one of the best frameworks for developing any Single Page Application or SPA Applications.

Angular is a UI framework for building mobile and desktop web applications. It is built using javascript framework for front-end development. It uses Typescript language (Superset of javascript) to make angular application. You can build amazing client-side applications using HTML, CSS, and Typescript using Angular. It is maintained by Google and a community of experts acting as a solution for rapid front-end development.

Learner can have following roles after learning Angular in a company.

- Web developer
- Web app developer
- UI developer
- UX developer
- Front-end developer
- JavaScript developer

Now, let's start process of Environment setup to install angular framework. Following tools/packages are required to run angular project.

- Nodejs
- Npm
- Angular CLI
- IDE for writing your code

Here are the steps needs to be followed to Environment setup in angular.

Step1: Install Node.js and npm:

Node.js is tool to run the development server for an Angular application. This allows developers to make changes to the code and see the updates in real-time without having to manually reload. It can be used as a build tool to automate tasks such as compiling TypeScript to JavaScript, bundling the application code, and optimizing the code for production.

Node.js can also be used to create a backend API that an Angular application can consume. Overall, Node.js can greatly enhance the development and deployment process for Angular applications.

Npm stands for Node Package Manager, which is a package manager for the Node.js runtime environment. npm is used to manage packages and dependencies for Node.js applications, including Angular applications. In an Angular application, npm is used to install and manage packages that the application depends on, such as Angular itself, third-party libraries, and development tools. These packages are typically stored in the "node_modules" directory of the application.

npm provides a command-line interface that allows developers to install, update, and remove packages, as well as manage package versions and dependencies.

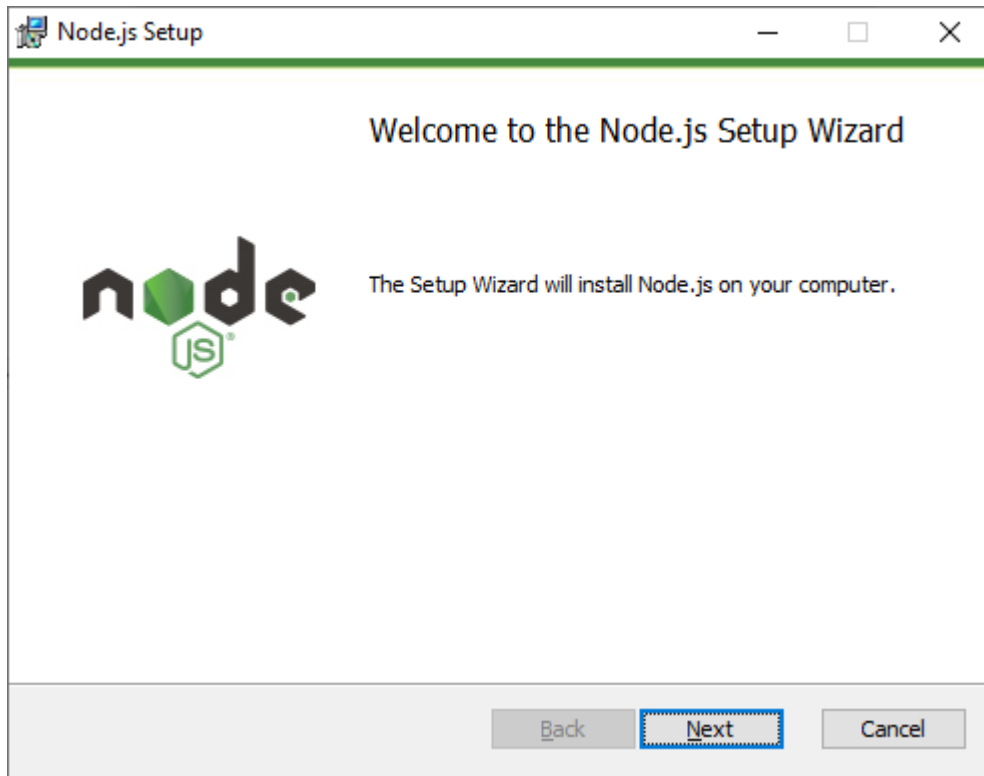
Overall, npm plays a critical role in the development and management of Angular applications, making it easier for developers to build and maintain their applications and collaborate with others in the community.

To install Node.js, Go to <https://nodejs.org/en/> and download the latest version of Node.js that corresponds to your operating system.

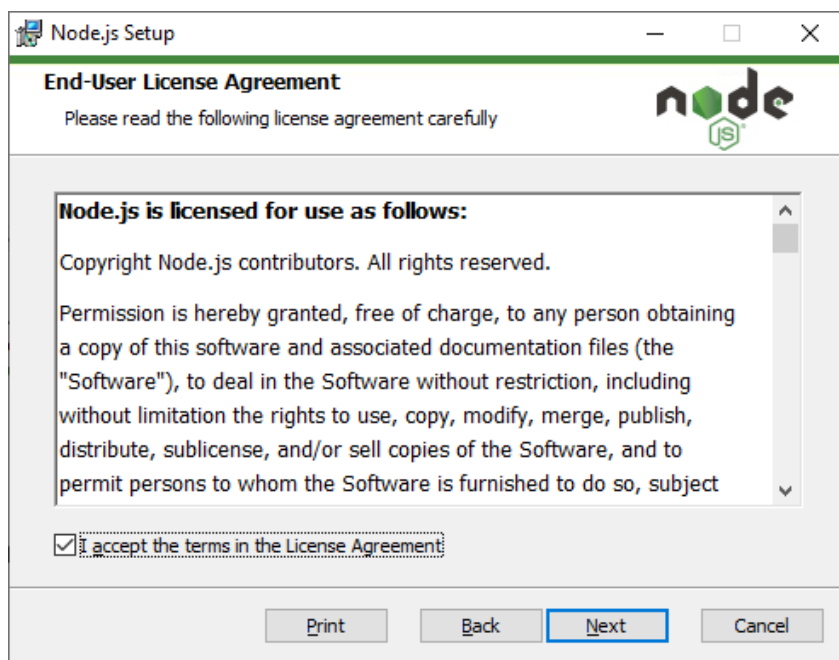
The screenshot shows the Node.js download page. At the top, there's a navigation bar with links: HOME, ABOUT, DOWNLOADS, DOCS, GET INVOLVED, SECURITY, CERTIFICATION, NEWS. Below the navigation bar, the page is titled "Downloads" and states "Latest LTS Version: 18.13.0 (includes npm 8.19.3)". It then says "Download the Node.js source code or a pre-built installer for your platform, and start developing today." There are two main sections: "LTS Recommended For Most Users" and "Current Latest Features". Under "LTS", there are three download options: "Windows Installer" (node-v18.13.0-x64.msi), "macOS Installer" (node-v18.13.0.pkg), and "Source Code" (node-v18.13.0.tar.gz). Under "Current", there are three download options: "Windows Installer" (node-v18.13.0-x64.msi), "macOS Installer" (node-v18.13.0.pkg), and "Source Code" (node-v18.13.0.tar.gz). Below these, there's a table of "Additional Platforms" with columns for "32-bit", "64-bit", and "ARMv7". The table lists various architectures and their corresponding download links.

32-bit	64-bit	ARMv7
32-bit	64-bit	
64-bit / ARM64		
64-bit	ARM64	
64-bit		
ARMv7	ARMv8	
node-v18.13.0.tar.gz		

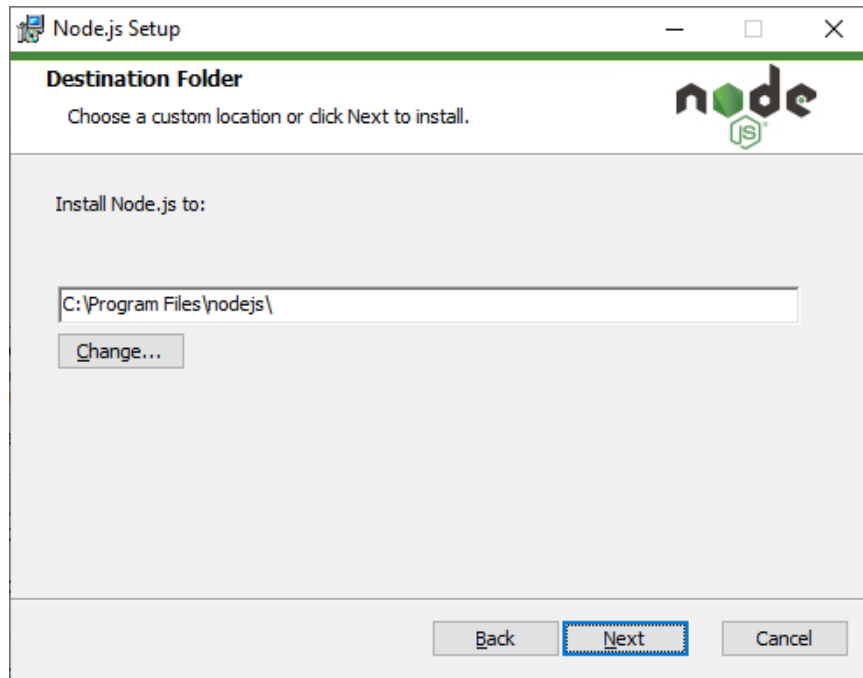
After download nodejs, Run the downloaded installer file and Follow the installation wizard to install Node.js on your computer.



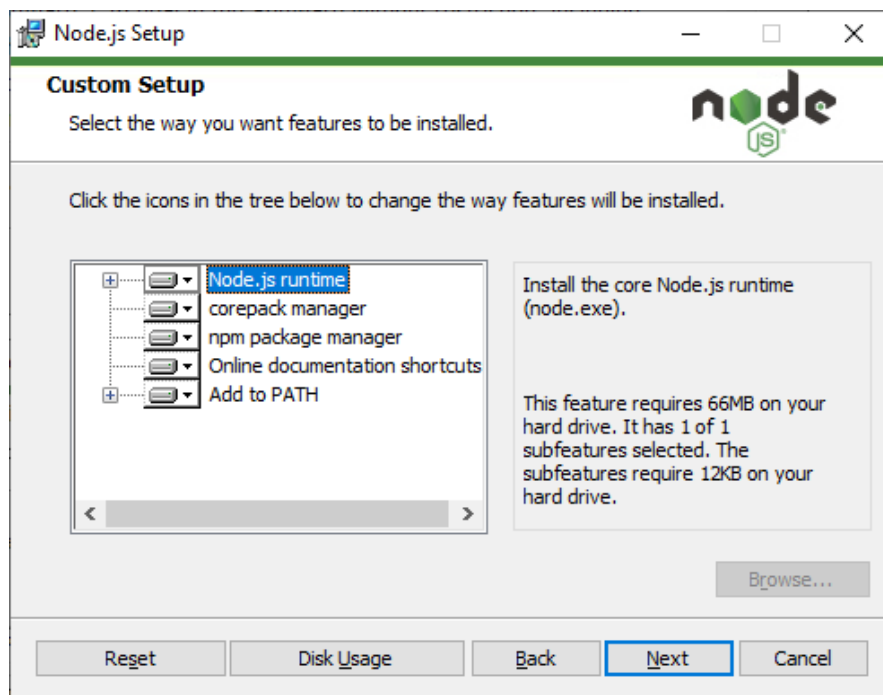
When click on next button, it will ask to accept End-User Licence Agreement as shown below.



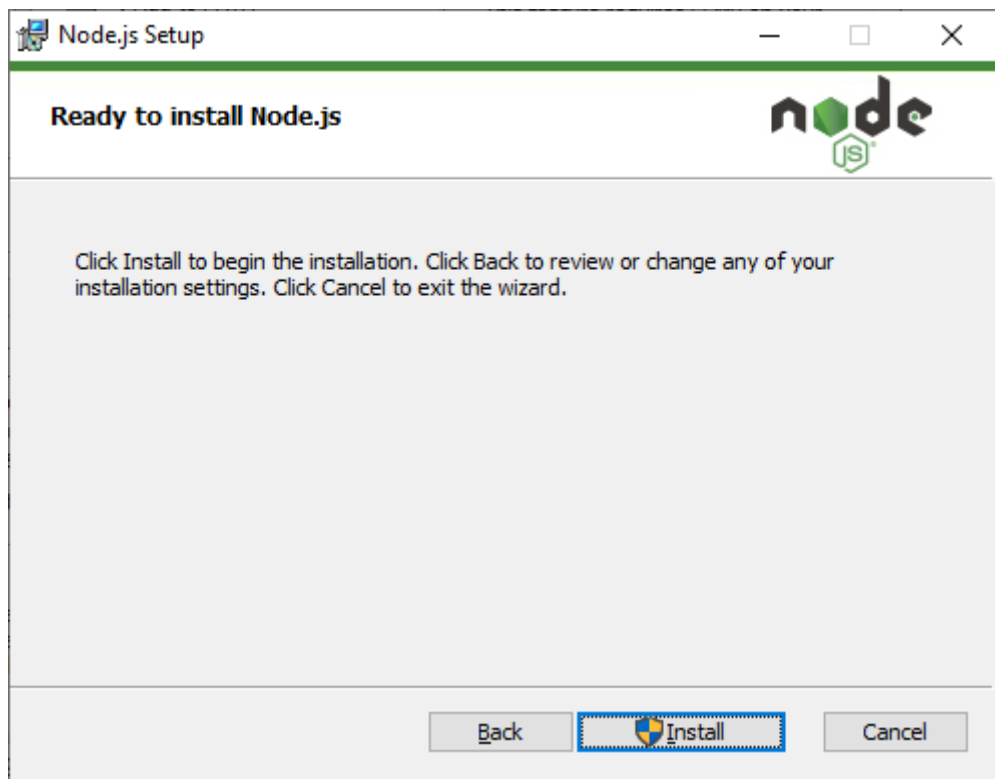
Once you check in checkbox and click on Next Button, you will get prompt box for node.js installation directory. Here you can see default location path as shown in below however you can customize your location by click on change button option.



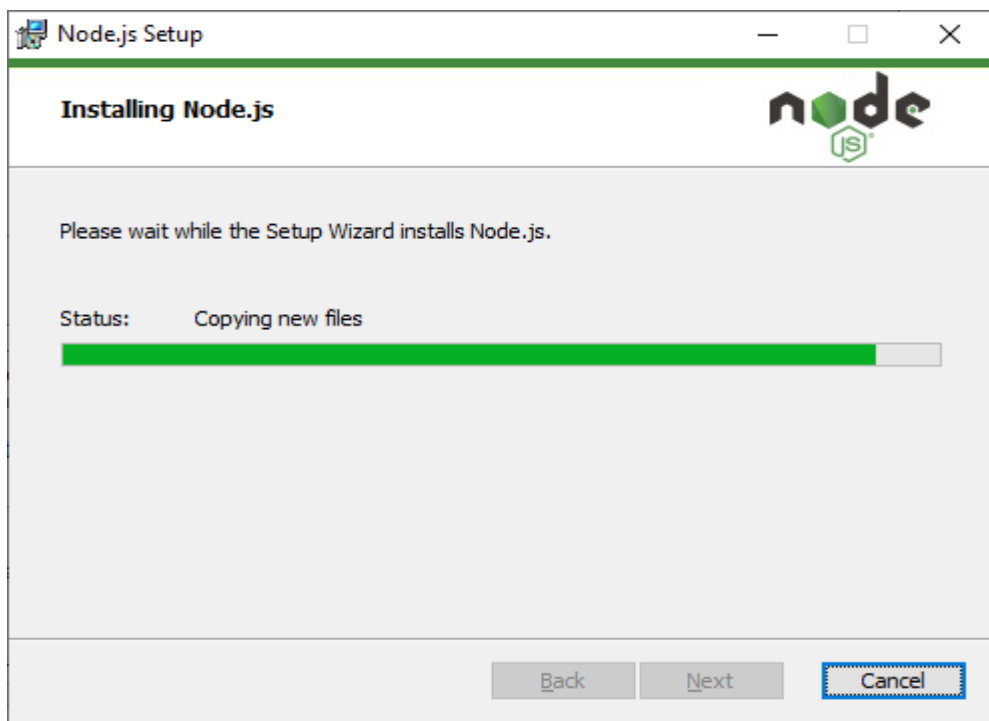
After choose directory, you can select features that you want to customize as shown in below.



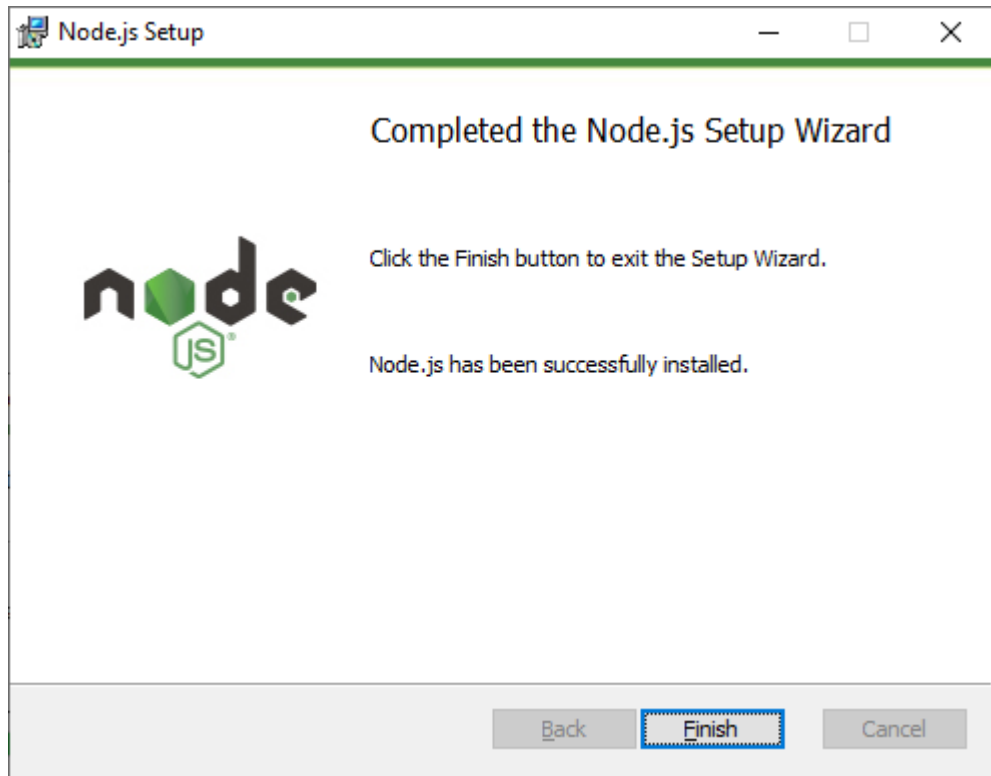
Here npm package manager is also installed automatically as you can see from above figure. After select custom features and click on next button, Node.js Setup wizard ready to install node.js and npm as shown below.



When you click on Install button, wizard will startfor installation as shown in below.Wait for the installation to complete. This may take a few minutes.



Once the installation is complete, click "Finish" to close the installer as shown below.

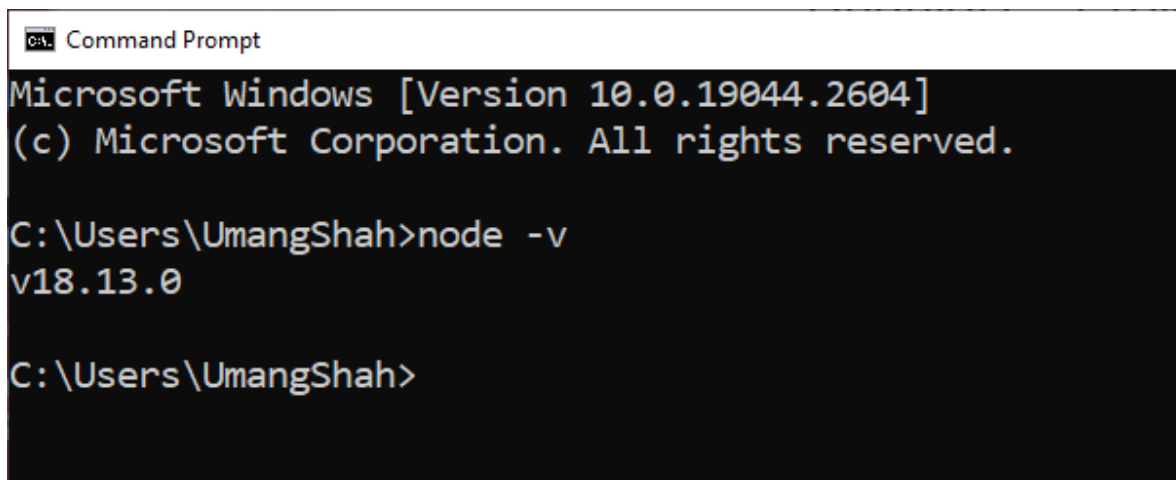


On successful installation, the command prompt will be displayed like this:

```
C:\Windows\system32\cmd.exe - node
Microsoft Windows [Version 10.0.19044.2604]
(c) Microsoft Corporation. All rights reserved.

C:\Users\UmangShah>node
Welcome to Node.js v18.13.0.
Type ".help" for more information.
>
```

You can Verify that Node.js has been installed correctly by opening a command prompt and typing "node -v" This should display the version of Node.js that you have installed as shown below.

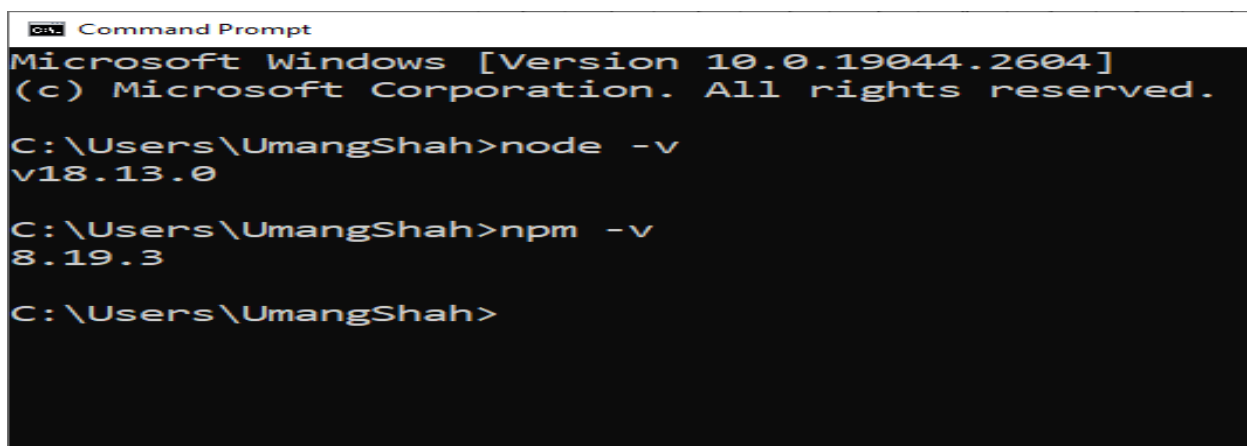
A screenshot of a Windows Command Prompt window. The title bar reads "Command Prompt". The window content shows the Microsoft Windows version (10.0.19044.2604) and copyright notice. The user is at the prompt "C:\Users\UmangShah>" and has entered the command "node -v". The output is "v18.13.0".

```
Command Prompt
Microsoft Windows [Version 10.0.19044.2604]
(c) Microsoft Corporation. All rights reserved.

C:\Users\UmangShah>node -v
v18.13.0

C:\Users\UmangShah>
```

You can verify that npm (Node Package Manager) has been installed correctly by opening a command prompt and typing "npm -v". This should display the version of npm that you have installed as shown below.

A screenshot of a Windows Command Prompt window, continuing from the previous one. The user has entered the command "npm -v" at the prompt "C:\Users\UmangShah>". The output is "8.19.3".

```
Command Prompt
Microsoft Windows [Version 10.0.19044.2604]
(c) Microsoft Corporation. All rights reserved.

C:\Users\UmangShah>node -v
v18.13.0

C:\Users\UmangShah>npm -v
8.19.3

C:\Users\UmangShah>
```

Step2: Install Angular CLI:

Angular CLI is used to create, build, and manage your Angular projects.


Once Node.js is installed, open a command prompt or terminal window and run the following command to install the Angular CLI

```
npm install -g @angular/cli
```


Wait for the installation to complete. This may take a few minutes depending on your internet speed. Once the installation is complete, run the following command to verify that the Angular CLI has been installed correctly.

`ng version`

You should see the version number of the Angular CLI displayed in the console output as given below.



```
C:\Windows\system32\cmd.exe
>
C:\Users\UmangShah>ng version

Angular CLI: 15.1.6
Node: 18.13.0
Package Manager: npm 8.19.3
OS: win32 x64

Angular:
...

Package                                  Version
-----
@angular-devkit/architect                0.1501.6 (cli-only)
@angular-devkit/core                     15.1.6 (cli-only)
@angular-devkit/schematics                15.1.6 (cli-only)
@schematics/angular                      15.1.6 (cli-only)
```

You have successfully installed the Angular CLI on your system.

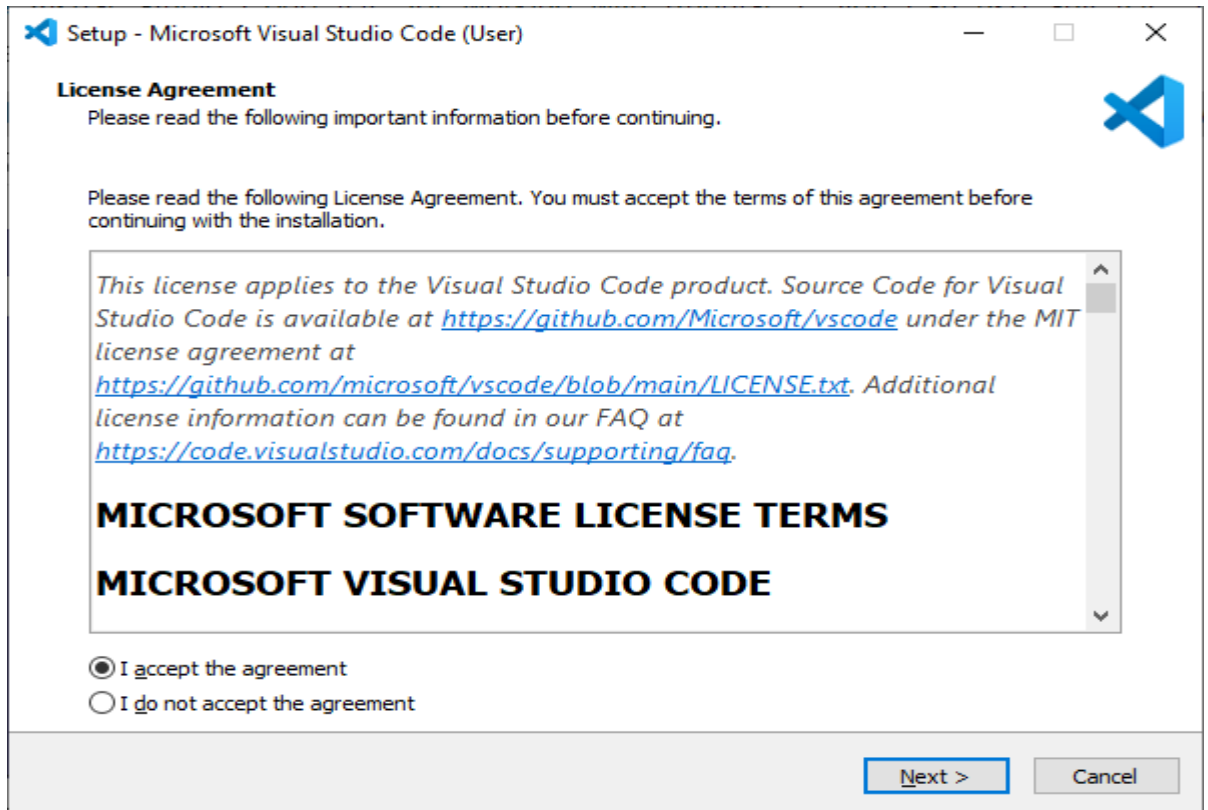
It gives the version for Angular CLI, typescript version and other packages available for Angular.

We are done with the installation of Angular.

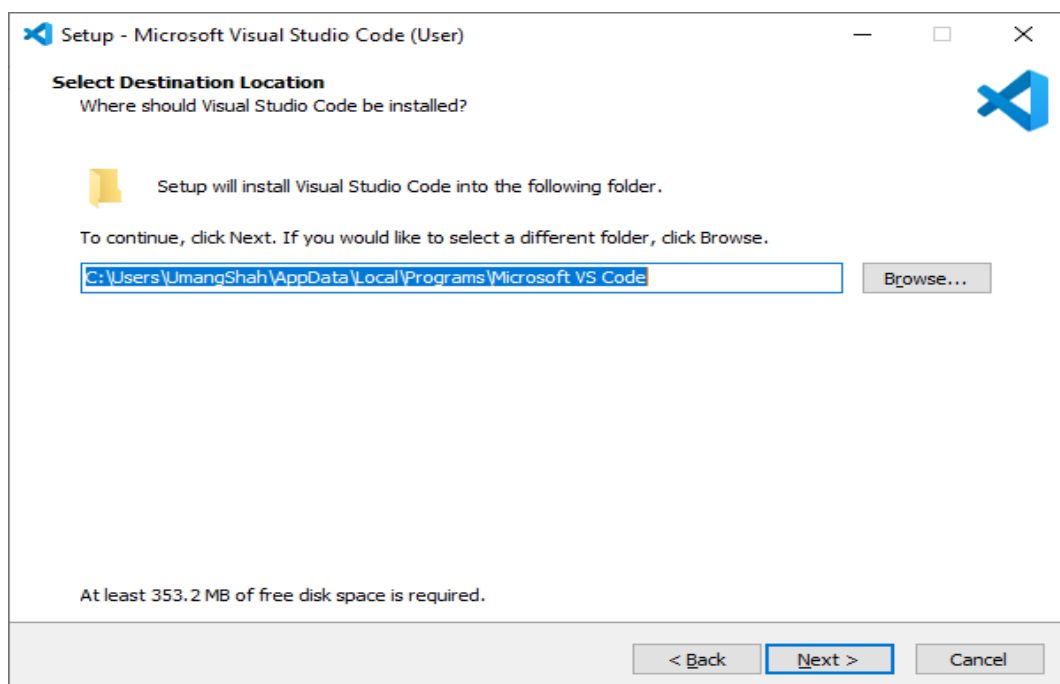
Step3: Install Visual Studio Code Editor:

Following steps are needed to perform for installing Visual Studio Code.

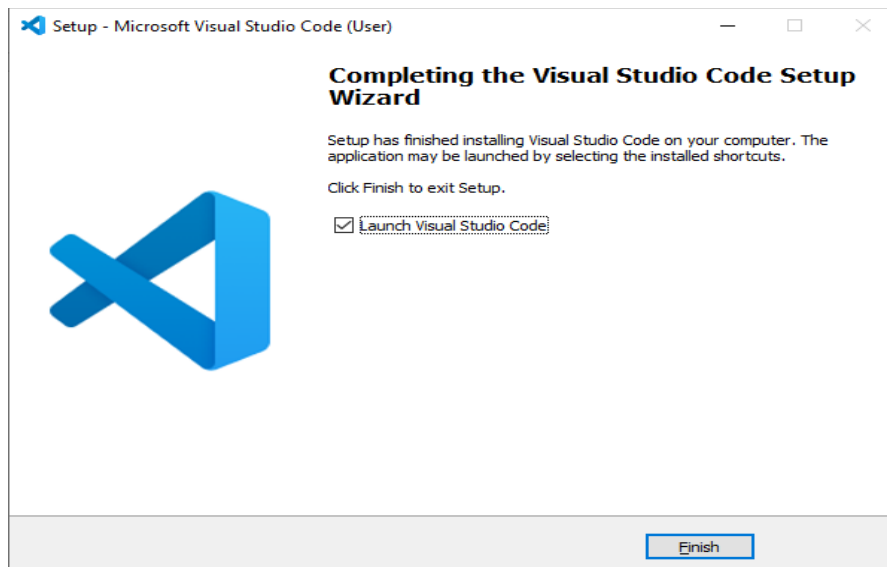
1. Go to the official Visual Studio Code website at <https://code.visualstudio.com/>
2. Click on the "Download for Windows" button to download the Windows version of Visual Studio Code.



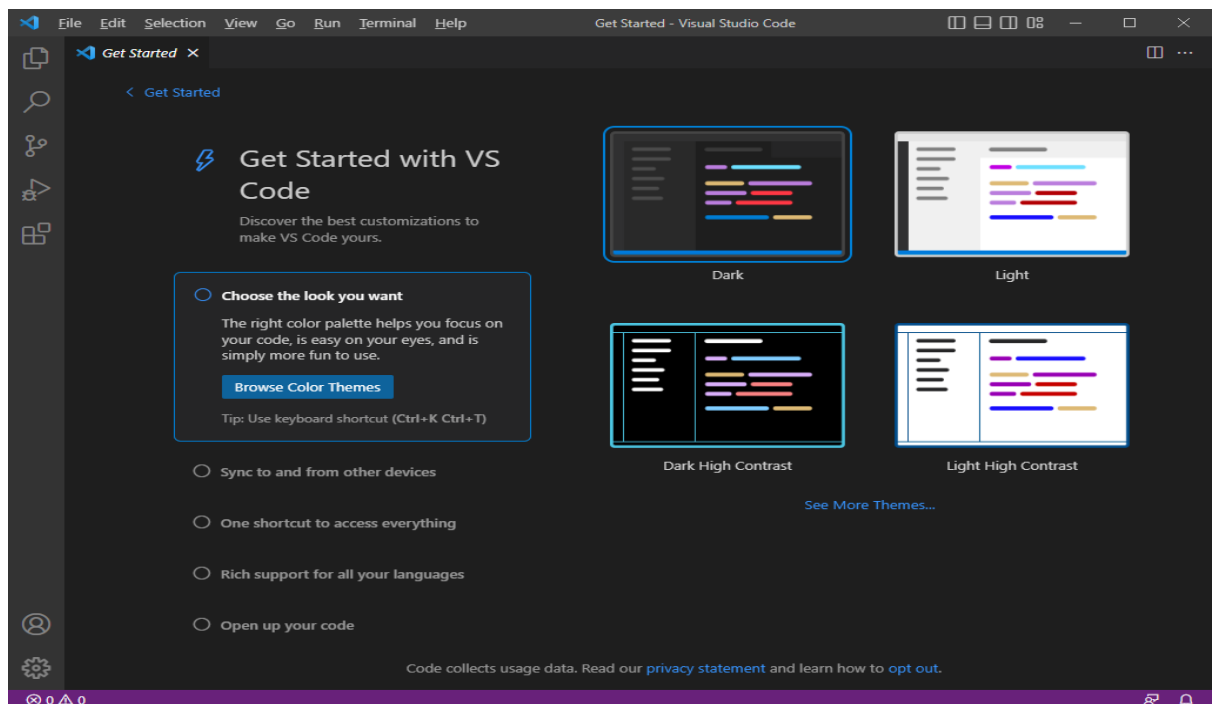
3. Run the downloaded installer.
4. Click on the "Next" button to begin the installation process.
5. Choose the installation location and click on the "Next" button.



6. Choose the start menu folder and click on the "Next" button.
7. Choose the additional tasks you want the installer to perform and click on the "Next" button.
8. Click on the "Install" button to start the installation process.
9. Wait for the installation process to complete.



10. Once the installation is complete, click on the "Finish" button to exit the installer.
11. Launch Visual Studio Code by double-clicking on the desktop icon or by searching for "Visual Studio Code" in the Windows Start menu.



You have successfully installed Visual Studio Code on your Windows computer for manage Angular application.

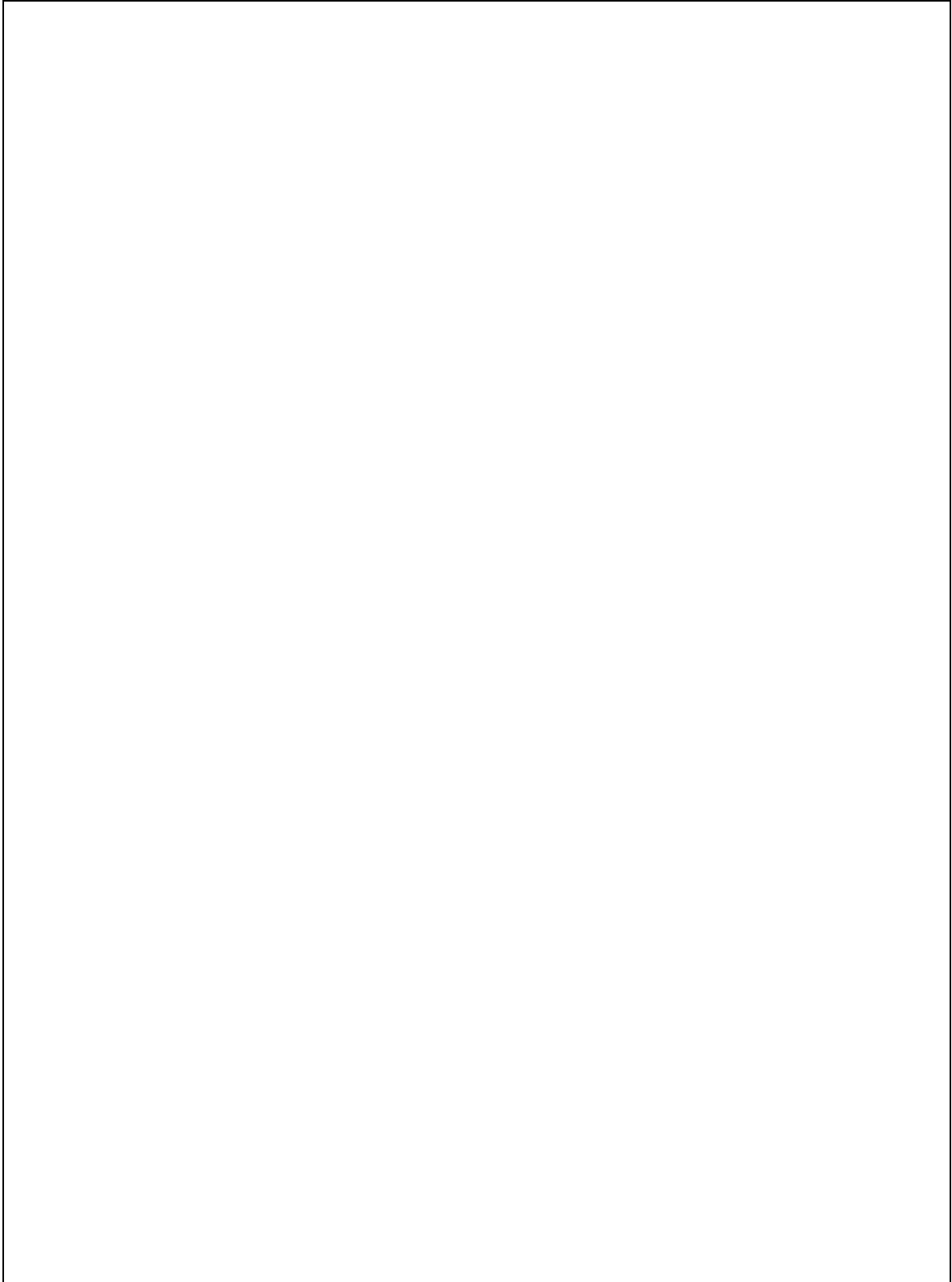
H. Resources/Equipment Required

S r. N o.	Equipment/ Software Resources	Specification
1	Computer System	Intel I3 processor with minimum 4 GB RAM, 40GB HDD, Windows 7 or above Operating system.
2	Visual Code	Open source software from Microsoft
3	Node JS and NPM Package Manager	Open source software
4	Browser	Microsoft Edge, Google Chrome etc

I. Output Source code:

1. Snapshot of Node.Js successfully installed.

OUTPUT



2. Snapshot of Angular/CLI installed successfully.

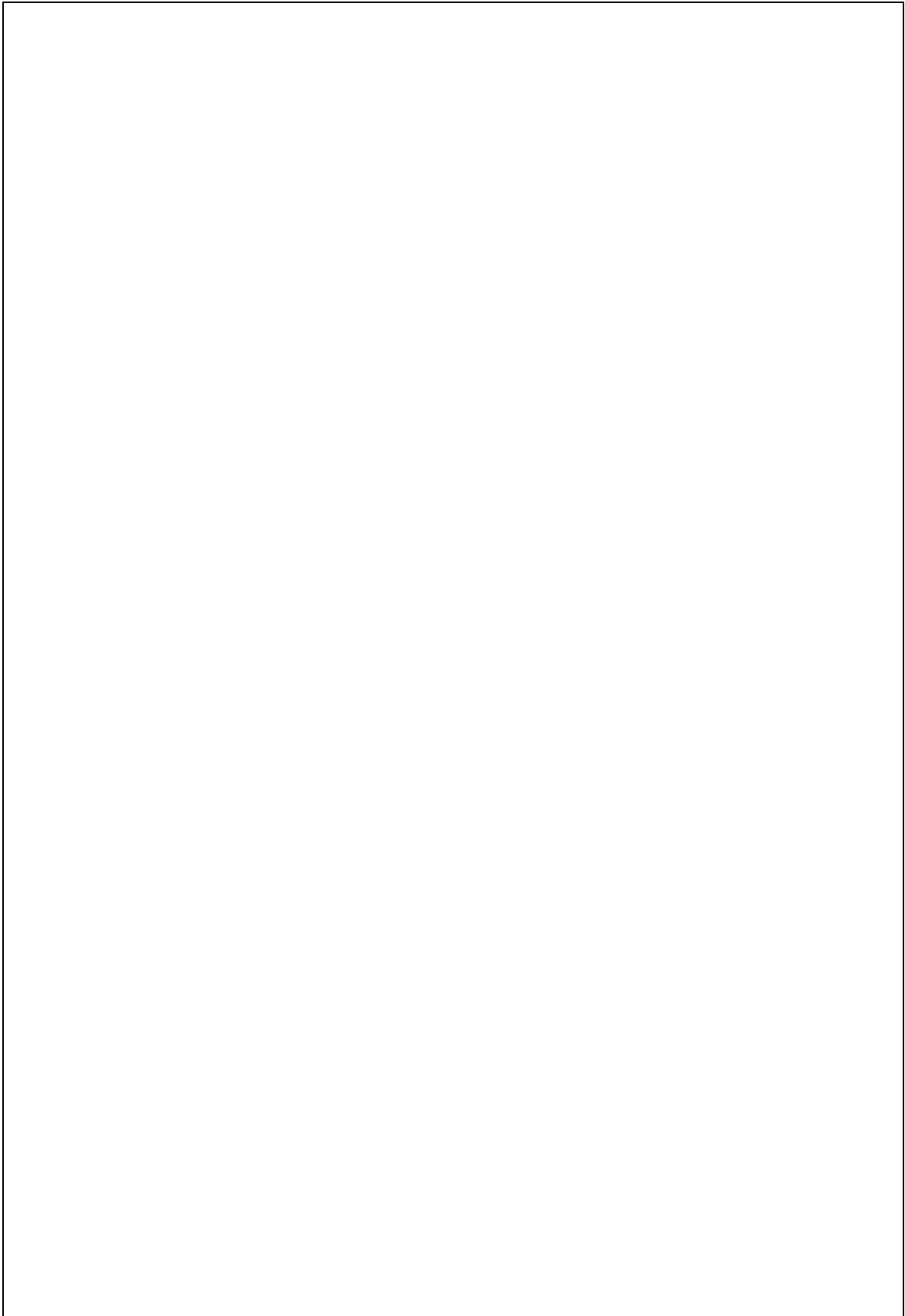
OUTPUT

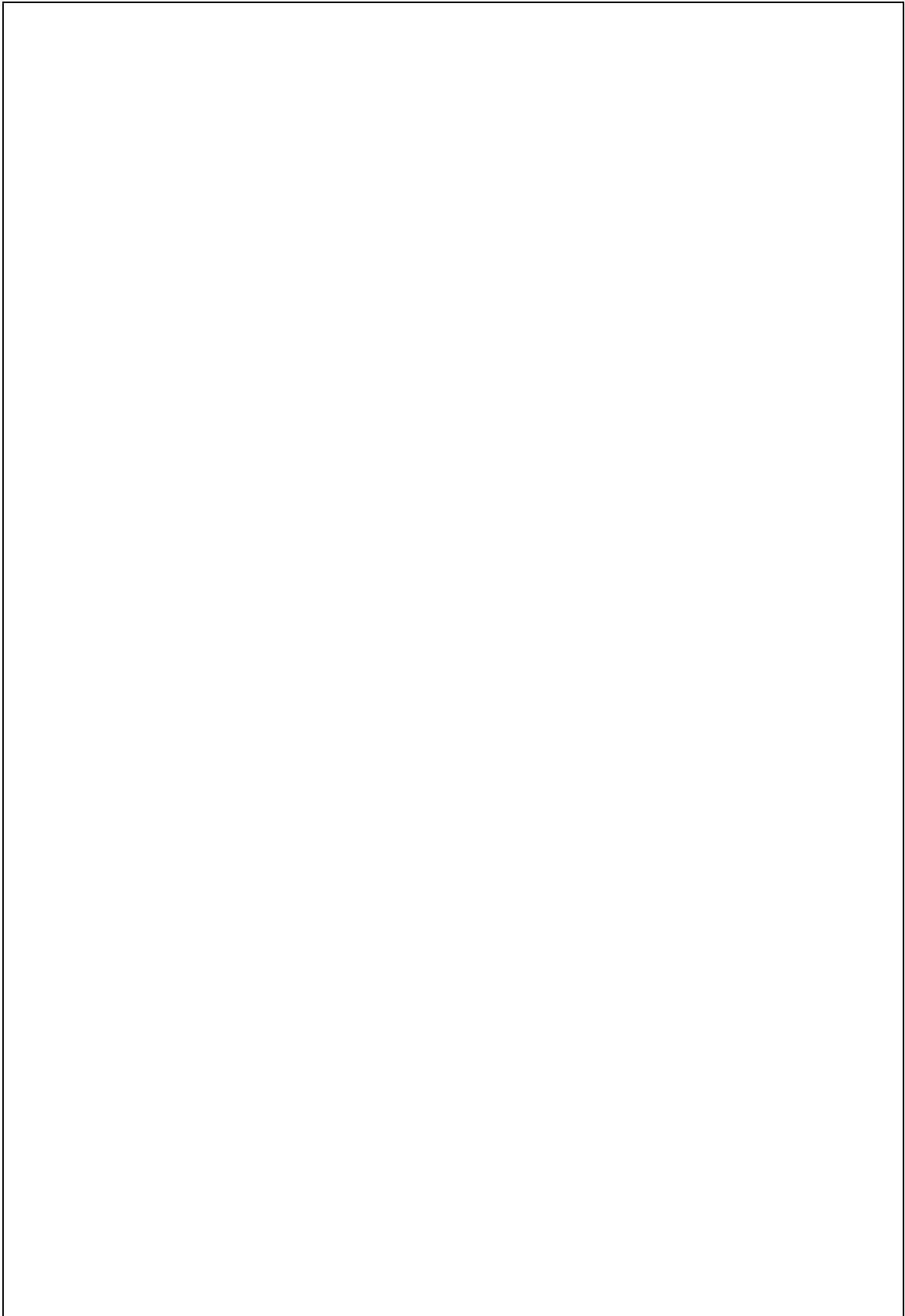
3. Snapshot of Visual Studio Code successfully installed.

OUTPUT

J. Practical related Questions.

1. Why Angular is more popular than other framework?
2. Describe NPM in detail.
3. How do you verify that Node.js is installed correctly on your system?
4. How do you verify that the Angular CLI is installed correctly on your system?





Practical No.2: Create first application to print Hello World message using angular framework.

A. Objective:

The objective of this program is a simple that is starting point for learning a new programming language or framework that is to display a simple message or output on the screen to demonstrate the basic functionality of the framework.

B. Expected Program Outcomes (POs)

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the digital electronics engineering problems.
2. **Design/ development of solutions:** Design solutions for digital electronics engineering well-defined technical problems and assist with the design of systems components or processes to meet specified needs.
3. **Engineering Tools, Experimentation and Testing:** Apply modern digital electronics engineering tools and appropriate technique to conduct standard tests and measurements.

C. Expected Skills to be developed based on competency:

1. Project Environment setup for run application in Angular framework.
2. Display Data on web application using Angular framework.

D. Expected Course Outcomes(Cos)

Setup environment for angular practical execution with NODE JS and NPM Package manager.

E. Practical Outcome(PRo)

Create first application to print Hello World message using angular framework.

F. Expected Affective domain Outcome(ADos)

1. Follow Coding standards and practices.
2. Maintain tools and equipment.
3. Follow safety practices.
4. Follow ethical practices

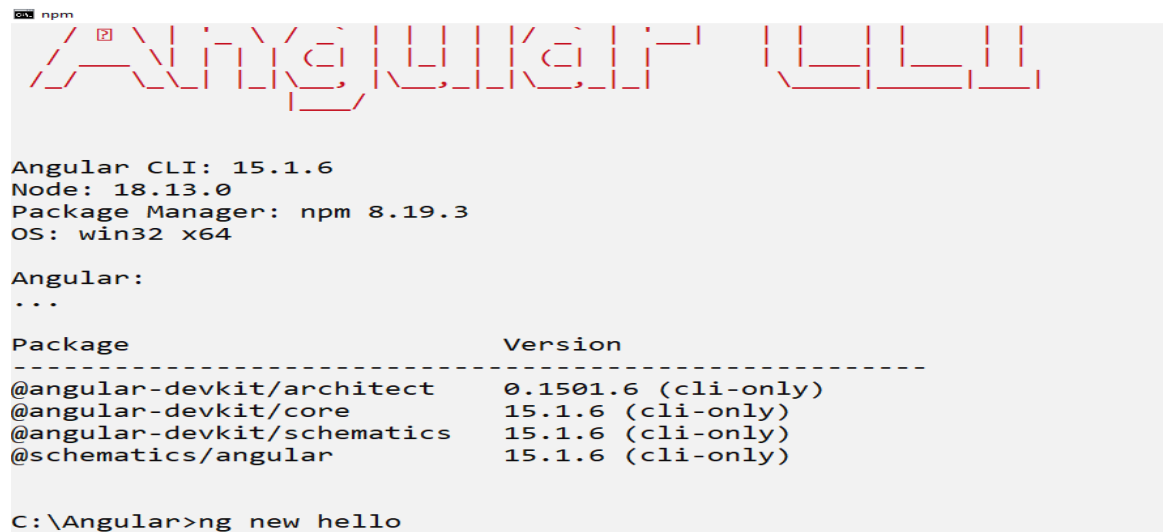
G. Prerequisite Theory:

To create a project in Angular After setup environment, go to directory where you already installed Angular/Cli and type following commands using either terminal of visual studio or command line.

ng new projectname

Here, You can use the projectname of your choice.

Let us now run the above command in the command line.



```

Angular CLI: 15.1.6
Node: 18.13.0
Package Manager: npm 8.19.3
OS: win32 x64

Angular:
...

Package                                Version
-----
@angular-devkit/architect              0.1501.6 (cli-only)
@angular-devkit/core                   15.1.6 (cli-only)
@angular-devkit/schematics             15.1.6 (cli-only)
@schematics/angular                   15.1.6 (cli-only)

C:\Angular>ng new hello
  
```

Once you run the command it will ask you about routing as shown below –



```

Angular CLI: 15.1.6
Node: 18.13.0
Package Manager: npm 8.19.3
OS: win32 x64

Angular:
...

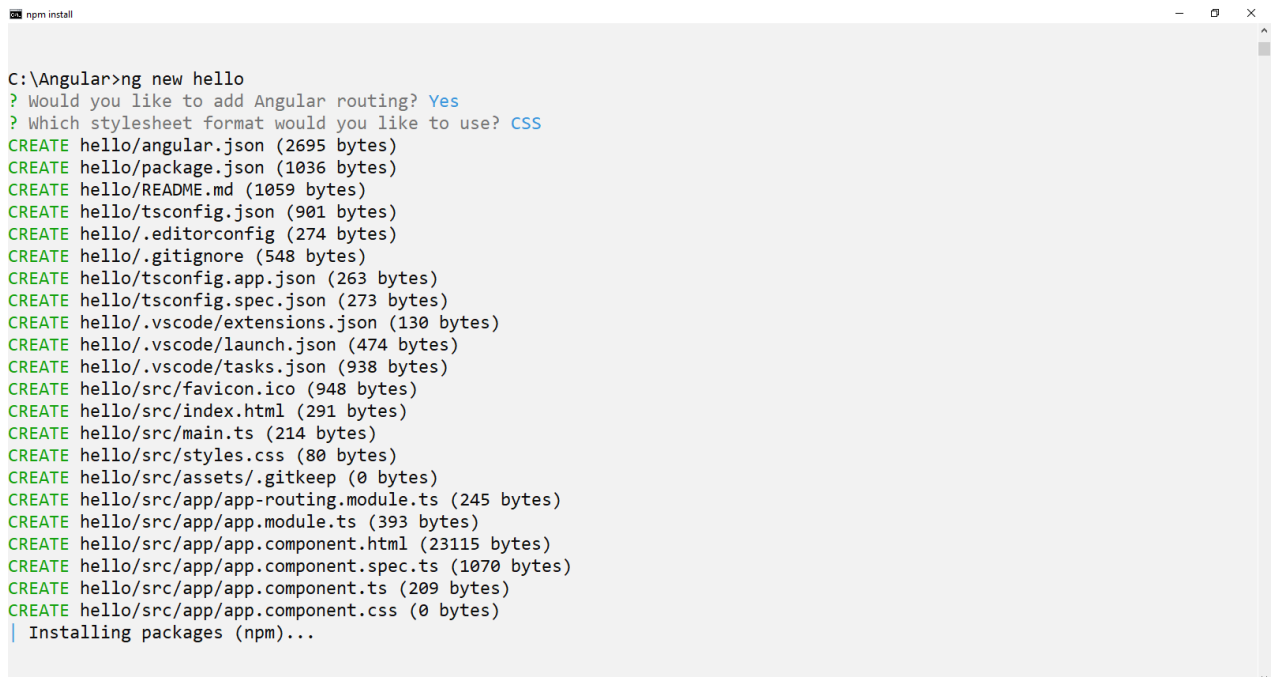
Package                                Version
-----
@angular-devkit/architect              0.1501.6 (cli-only)
@angular-devkit/core                   15.1.6 (cli-only)
@angular-devkit/schematics             15.1.6 (cli-only)
@schematics/angular                   15.1.6 (cli-only)

C:\Angular>ng new hello
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use?
> CSS
SCSS  [ https://sass-lang.com/documentation/syntax#scss ]
Sass  [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
Less  [ http://lesscss.org ]
  
```

Type y to add routing to your project setup and again ask question for the stylesheet as shown below.

The options available are CSS, Sass, Less and Stylus. In the above screenshot, the arrow is on CSS. To change, you can use arrow keys to select the one required for your project setup. At present, we select CSS option for our project-setup.

It installs all the required packages necessary for our project to run in Angular as shown below.

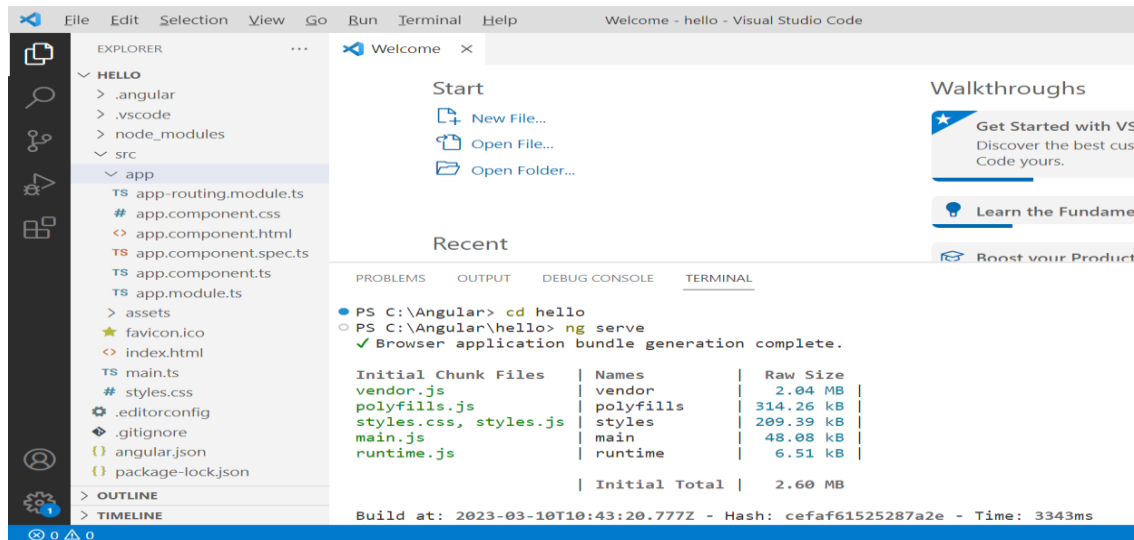


```
npm install
C:\Angular>ng new hello
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? CSS
CREATE hello/angular.json (2695 bytes)
CREATE hello/package.json (1036 bytes)
CREATE hello/README.md (1059 bytes)
CREATE hello/tsconfig.json (901 bytes)
CREATE hello/.editorconfig (274 bytes)
CREATE hello/.gitignore (548 bytes)
CREATE hello/tsconfig.app.json (263 bytes)
CREATE hello/tsconfig.spec.json (273 bytes)
CREATE hello/.vscode/extensions.json (130 bytes)
CREATE hello/.vscode/launch.json (474 bytes)
CREATE hello/.vscode/tasks.json (938 bytes)
CREATE hello/src/favicon.ico (948 bytes)
CREATE hello/src/index.html (291 bytes)
CREATE hello/src/main.ts (214 bytes)
CREATE hello/src/styles.css (80 bytes)
CREATE hello/src/assets/.gitkeep (0 bytes)
CREATE hello/src/app/app-routing.module.ts (245 bytes)
CREATE hello/src/app/app.module.ts (393 bytes)
CREATE hello/src/app/app.component.html (23115 bytes)
CREATE hello/src/app/app.component.spec.ts (1070 bytes)
CREATE hello/src/app/app.component.ts (209 bytes)
CREATE hello/src/app/app.component.css (0 bytes)
| Installing packages (npm)...
```

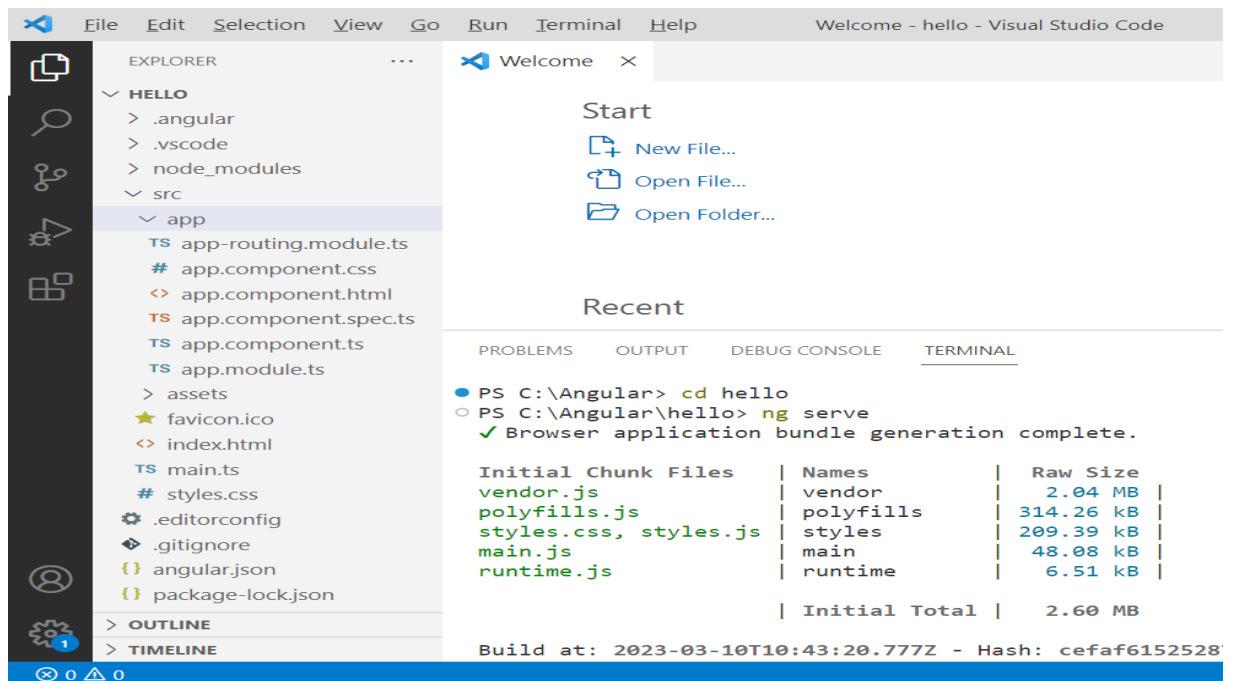
Once all packages installed, Our project is installed successfully. We need to go our project directory using following command

cd projectname

Now I am going to show you using Visual code editor. You can see above command in our example as given below.



If you want to see the file structure of our project where you can manage your application, open Visual studio code editor which is already installed and open our project directory, you will get the folder structure that looks like the one given below.



To compile our project, the following command is used in angular.

ng serve

The ng serve command builds the application and starts the web server.

You will see the below output when the command starts executing.

The screenshot shows the Visual Studio Code interface with the Explorer sidebar on the left, the main editor area, and the Terminal panel at the bottom. The Explorer sidebar shows a project structure with files like `app-routing.module.ts`, `app.component.css`, `app.component.html`, `app.component.spec.ts`, `app.component.ts`, `app.module.ts`, `assets`, `favicon.ico`, `index.html`, `main.ts`, `styles.css`, `.editorconfig`, `.gitignore`, `angular.json`, and `package-lock.json`. The main editor area shows the `Start` page with options like `New File...`, `Open File...`, and `Open Folder...`. The Terminal panel shows the following output:

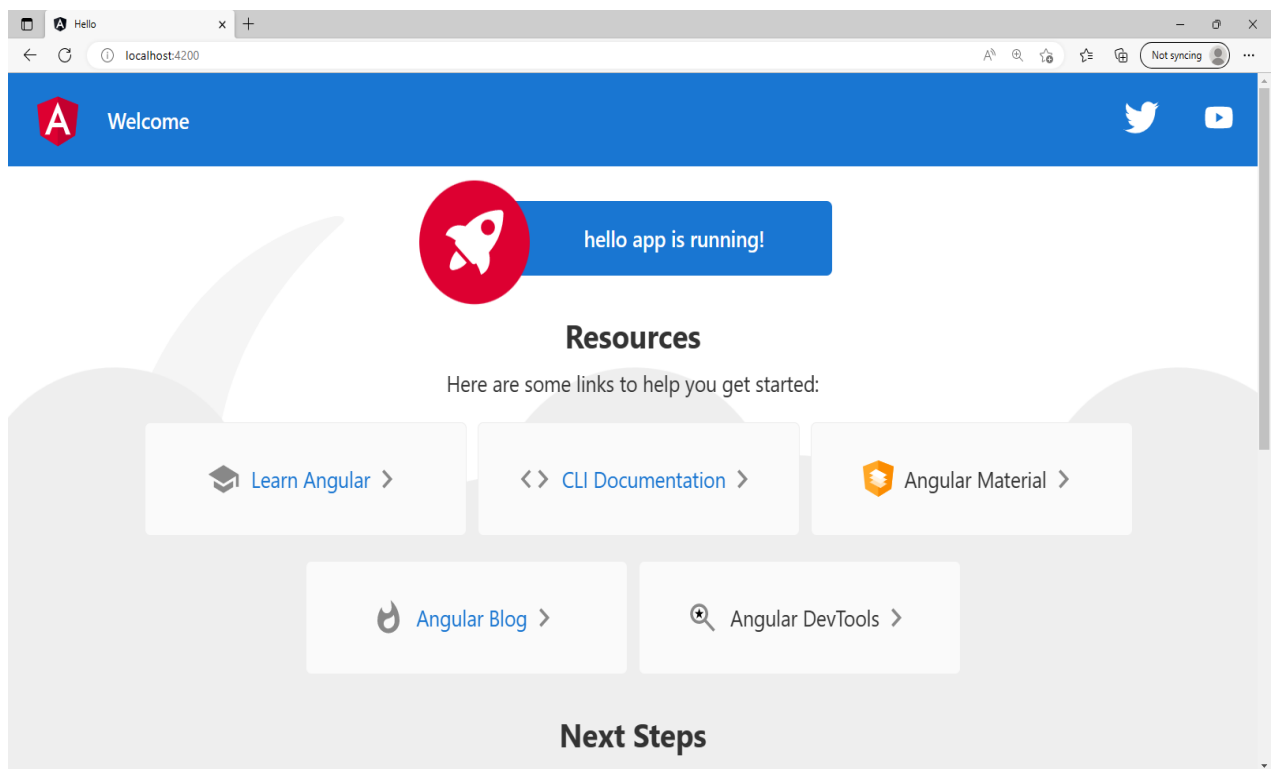
```
PS C:\Angular\hello> cd..
PS C:\Angular> cd hello
PS C:\Angular\hello> ng serve
✓ Browser application bundle generation complete.
```

Initial Chunk Files	Names	Raw Size
<code>vendor.js</code>	<code>vendor</code>	2.04 MB
<code>polyfills.js</code>	<code>polyfills</code>	314.26 kB
<code>styles.css, styles.js</code>	<code>styles</code>	209.39 kB
<code>main.js</code>	<code>main</code>	48.08 kB
<code>runtime.js</code>	<code>runtime</code>	6.51 kB
Initial Total		2.60 MB

```
Build at: 2023-03-10T10:43:20.777Z - Hash: cefaf61525287a2e - Time: 3343ms

** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **
```

Once the project is compiled and type url, `http://localhost:4200/` in the browser, you will see your project home page as shown below.



The necessary files for this app are given below:

1. **app.module.ts**: This is the root module of an Angular application. It is responsible for importing and configuring all the components, services, directives, and other modules used in the application. The module is typically defined in a file named **app.module.ts**.

Here is an example:

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

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

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

2. **app.component.ts**: This is the root component of an Angular application. It defines the structure and behavior of the main view of the application. The component is typically defined in a file named **app.component.ts**.

Here is an example:

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

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

3. **app.component.html**: This is the HTML template for the root component. It defines the layout and content of the main view of the application. The template is typically defined in a file named **app.component.html**. Here is an example:

```
<h1>Hello World!</h1>
```

4. **main.ts:** This is the main entry point of an Angular application. It bootstraps the root module of the application and starts the Angular runtime. The file is typically named main.ts.

Here is an example:

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
enableProdMode();
platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

5. **angular.json:** This is the configuration file for an Angular application. It defines the settings and options for building, serving, and testing the application. The file is typically named angular.json

Now we are going to understand concepts of string interpolation to make our desired output.

String interpolation is a feature in Angular that allows you to embed dynamic values from your component's TypeScript code into your HTML template. This can make your HTML templates more dynamic and interactive. Here's an example of how to use string interpolation in both the HTML and TypeScript files of an Angular component:

HTML Template:

```
<h1>Welcome {{name}}!</h1>
```

TypeScript File:

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

@Component({
  selector: 'app-welcome',
  templateUrl: './welcome.component.html',
  styleUrls: ['./welcome.component.css']
})
export class WelcomeComponent {
  name = 'Admin';
}
```


In the HTML template, we've used double curly braces `{{ }}` to surround the name variable, which is defined in the TypeScript file. This tells Angular to replace the `{{name}}` with the value of the name variable when the component is rendered.

In the TypeScript file, we've imported the Component decorator from `@angular/core` and used it to define our component. We've also defined a name variable and set it to 'John'. When the component is rendered, Angular will replace `{{name}}` with the value of name, which is 'Admin'.

You can also use string interpolation to perform simple operations within the template, such as concatenating strings or performing basic arithmetic:

```
<p>{{firstName + ' ' + lastName}} is {{age + 1}} years old</p>
```

Where firstName,lastName and age property variables's value must be declared in TypeScript file.

H. Resources/Equipment Required

S r. N o.	Equipment/ Software Resources	Specification
1	Computer System	Intel I3 processor with minimum 4 GB RAM, 40GB HDD, Windows 7 or above Operating system.
2	Visual Code	Open source software from Microsoft
3	Node JS and NPM Package Manager	Open source software
4	Browser	Microsoft Edge, Google Chrome etc

I. Program Source code Output

SOURCE CODE

Practical No.3: Design a web page to utilize property binding and event binding concepts using button and textbox controls.

A. Objective:

1. To know how to exchange data between the component and the template
2. Create dynamic, interactive web applications that respond to user input and update dynamically based on changes in the component.

B. Expected Program Outcomes (POs)

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the digital electronics engineering problems.
2. **Design/ development of solutions:** Design solutions for digital electronics engineering well-defined technical problems and assist with the design of systems components or processes to meet specified needs.
3. **Engineering Tools, Experimentation and Testing:** Apply modern digital electronics engineering tools and appropriate technique to conduct standard tests and measurements.

C. Expected Skills to be developed based on competency:

To create dynamic and interactive user interfaces, Manipulating the DOM, Creating responsive UIs, Implementing data-driven applications.

D. Expected Course Outcomes(Cos)

Setup environment for angular practical execution with NODE JS and NPM Package manager.

E. Practical Outcome(PRo)

Design a web page to utilize property binding and event binding concepts using button and textbox controls.

F. Expected Affective domain Outcome(ADos)

1. Follow Coding standards and practices.
2. Maintain tools and equipment.
3. Follow safety practices.
4. Follow ethical practices

G. Prerequisite Theory:

PropertyBinding :

Property **binding** is a feature in Angular that allows you to set the value of an HTML element property to a value from the component. In other words, you can bind a property of an HTML element to a property in your component, and the value of the HTML property will be dynamically updated as the value of the component property changes.

Here is an example of how to use property binding in an Angular component:

HTML Template (app.component.html)

```
<p>Welcome to {{title}}!</p>
<button [disabled]="isDisabled">Click me</button>
```

In this example, we have two different uses of property binding. The first one binds the content of a paragraph (**<p>**) to a property **title** in the component. The curly braces **{{ }}** indicate that this is an example of **one-way data binding**. When the component is rendered, the value of the **title** property is interpolated into the template.

The second use of property binding binds the **disabled** attribute of a button to the **isDisabled** property of the component. Square brackets **[]** are used to indicate that this is an example of property binding. When the value of the **isDisabled** property changes, the **disabled** attribute of the button will be updated accordingly.

TypeScript 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 = 'Property binding Demo';
  isDisabled = false;
}
```

In the TypeScript file for our component, we define two properties: **title** and **isDisabled**. The **title** property is a simple string that is used in the first example of property binding to set the content of a paragraph.

The **isDisabled** property is a boolean value that is used in the second example of property binding to disable or enable a button. By default, the **isDisabled** property is set to **false**, so the button is enabled. If we were to change the value of the **isDisabled** property to **true**, the button would become disabled.

By using property binding, we can make our Angular templates more dynamic and interactive. We can easily bind properties of HTML elements to properties in our component, and update them in real time as our data changes.

Event binding:

Event binding is a feature in Angular that allows you to listen for and respond to events that occur in the HTML template, such as button clicks, key presses, or mouse movements. You can bind an event to a method in your component, and when the event is triggered, the method will be called with any relevant data.

Here is an example of how to use event binding in an Angular component:

HTML Template (app.component.html)

```
<button (click)="onClick()">Click me</button>

<p>{{ message }}</p>
```

In this example, we have a button element that is used to trigger an event, and a paragraph element that is used to display a message based on the event. The (click) syntax is used to bind the click event of the button to a method onClick() in the component. When the button is clicked, the onClick() method will be called.

The message that is displayed in the paragraph element is determined by the message property in the component, which is updated by the onClick() method.

TypeScript File (app.component.ts)

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

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

```
export class AppComponent {  
  message: string = "";  
  onClick() {  
    this.message = 'Button clicked!';  
  }  
}
```

In the TypeScript file for our component, we define a property **message** that will be used to display a message in the paragraph element. We also define a method **onClick()** that will be called when the button is clicked.

In the **onClick()** method, we update the value of the **message** property to **'Button clicked!'**. This will cause the message to be updated in the HTML template, and the new message will be displayed to the user.

By using event binding, we can create more interactive and responsive Angular components. We can listen for events in the HTML template and respond to them by updating the properties and methods in our component. This allows us to create dynamic and engaging user interfaces that respond to user input in real time.

ngModel Directive:

ngModel is a built-in directive in Angular that allows you to perform two-way data binding between a form control element and a property in your component. It's typically used with input and select elements to capture user input and update the component's properties with the entered value.

Here's an example of how to use **ngModel** in an Angular component:

HTML Template (app.component.html)

```
<input [(ngModel)]="name" placeholder="Enter your name">  
<p>Your name is: {{name}}</p>
```

In this example, we have an input element that is used to capture the user's name. We use the **[(ngModel)]** syntax to bind the input's value to the **name** property in the component.

This is a two-way data binding, which means that any changes to the input element's value will be reflected in the **name** property, and any changes to the **name** property will be reflected in the input element's value.

We also have a paragraph element that displays the value of the **name** property.

TypeScript File (app.component.ts)

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

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

In the TypeScript file for our component, we define a property **name** that will be used to store the user's name. By default, the **name** property is initialized to an empty string.

When the user types a value into the input element, the **name** property will be updated with that value. When the **name** property is updated (for example, if it's updated in code), the input element's value will be updated as well.

By using **ngModel**, we can easily capture user input and update our component's properties in real time. We don't have to manually update the properties or listen for events - **ngModel** takes care of all of that for us. This makes it a very useful tool for creating forms and other interactive user interfaces in Angular.

H. Resources/Equipment Required

S r. N o.	Equipment/ Software Resources	Specification
1	Computer System	Intel I3 processor with minimum 4 GB RAM, 40GB HDD, Windows 7 or above Operating system.

Practical related Exercises.

1. Design a webpage to display your name, enrolment number, college name using interpolation and photo using property binding.
2. Design a webpage to display your name, enrolment number, college name using event binding.[hint : name, number and college name values must be declared in ts file]

Practical No.4: Create various components of web page using Attribute Directives.

A. Objective:

1. Create a more modular, scalable, and maintainable web page
2. Manipulate and add functionality to HTML elements by binding custom attributes.

B. Expected Program Outcomes (POs)

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the *engineering* problems.
2. **Problem analysis:** Identify and analyse well-defined *engineering* problems using codified standard methods.
3. **Design/ development of solutions:** Design solutions for *engineering* well-defined technical problems and assist with the design of systems components or processes to meet specified needs.
4. **Engineering Tools, Experimentation and Testing:** Apply modern *engineering* tools and appropriate technique to conduct standard tests and measurements.
5. **Life-long learning:** Ability to analyze individual needs and engage in updating in the context of technological changes *in field of engineering*.

C. Expected Skills to be developed based on competency:

Learner can develop a wide range of technical and soft skills including angular framework knowledge, collaboration etc. making them better equipped to develop high-quality, efficient, and scalable web applications.

D. Expected Course Outcomes(Cos)

Apply angular directives, components and pipes in different web page development.

E. Practical Outcome(PRo)

Create various components of web page using Attribute Directives.

F. Expected Affective domain Outcome(ADos)

1. Follow Coding standards and practices.

2. Maintain tools and equipment.
3. Follow safety practices.
4. Follow ethical practices

G. Prerequisite Theory:

The component is the basic building block of Angular. It has a selector, template, style, and other properties, and it specifies the metadata required to process the component.

Creating a Component in Angular :

To create a component in any angular application, follow the below steps:

- [1] Get to the angular app via your terminal.
- [2] Create a component using the following command:
`ng g c <component_name>`
OR
`ng generate component <component_name>`

Following files will be created after generating the component:

- `component_name.component.html`
- `component_name.component.spec.ts`
- `component_name.component.ts`
- `component_name.component.css`

Now we are going to create a header, sidebar, and footer components in Angular, you can follow these steps:

1. Create a new Angular project:
`ng new my-app`
2. Create a header component using the following command:
`ng generate component header`
3. Create a sidebar component using the following command:
`ng generate component sidebar`
4. Create a footer component using the following command:
`ng generate component footer`

This will create the necessary files for each component, including HTML, CSS, TypeScript, and a spec file as we discussed earlier.

5. Open the `header.component.html` file and add the following code:

```
<header>
<h1>My Website</h1>
</header>
```

6. Open the sidebar.component.html file and add the following code:

```
<aside>
<h2>Menu</h2>
<nav>
<ul>
<li><a href="#">Home</a></li>
<li><a href="#">>About</a></li>
<li><a href="#">Contact</a></li>
</ul>
</nav>
</aside>
```

7. Open the footer.component.html file and add the following code:

```
<footer>
<p>My Website &copy; 2023</p>
</footer>
```

8. Open the app.component.html file and replace the existing code with the following code:

```
<app-header></app-header>
<main>
<app-sidebar></app-sidebar>
<section>
<router-outlet></router-outlet>
</section>
</main>
<app-footer></app-footer>
```

9. Open the app.component.css file and add the following styles:

```
header {
background-color: #333;
color: #fff;
padding: 20px;
}

aside {
background-color: #f2f2f2;
```

```
padding: 10px;
}

footer {
background-color: #333;
color: #fff;
padding: 20px;
text-align: center;
}

main {
display: flex;
}

section {
margin: 20px;
padding: 20px;
border: 1px solid #ccc;
flex-grow: 1;
}
```

You have designed a basic Angular app with a header, sidebar, and footer components.

ngClass directory

ngClass is a built-in Angular directive that allows you to dynamically add or remove CSS classes to an HTML element based on certain conditions. This is particularly useful when you want to apply different styles to an element based on its state or user interaction.

The **ngClass** directive can be used in two ways:

1. Using a string expression:

```
<div [ngClass]="class1 class2 class3">Example</div>
```

In this example, the **ngClass** directive applies three CSS classes to the **div** element: **class1**, **class2**, and **class3**.

2. Using an object expression:

```
<div [ngClass]="{ 'class1': condition1, 'class2': condition2 }">Example</div>
```

In this example, the **ngClass** directive applies the CSS class **class1** to the **div** element if **condition1** is true, and the CSS class **class2** if **condition2** is true. **condition1** and **condition2** value set in ts file.

Here's a more detailed example that demonstrates how to use **ngClass** to dynamically apply CSS classes based on user interaction:

```
<button [ngClass]="{ 'active': isActive }" (click)="toggleActive()">Toggle Active</button>
```

In this example, the **ngClass** directive applies the CSS class **active** to the **button** element if the **isActive** property is true. The **isActive** property is initially set to false in the component:

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

@Component({
  selector: 'app-button',
  template: `
    <button [ngClass]="{ 'active': isActive }" (click)="toggleActive()">Toggle Active</button>
  `,
})
export class ButtonComponent {
  isActive = false;

  toggleActive() {
    this.isActive = !this.isActive;
  }
}
```

When the user clicks the button, the **toggleActive()** method is called, which toggles the value of the **isActive** property. When **isActive** is true, the **active** CSS class is added to the button, and when **isActive** is false, the **active** CSS class is removed.

ngStyledirective :

ngStyle is a built-in Angular directive that allows you to dynamically add or remove inline styles to an HTML element based on certain conditions. This is particularly useful when you want to apply different styles to an element based on its state or user interaction.

The **ngStyle** directive can be used in the following way:

```
<div [ngStyle]="{ 'property1': value1, 'property2': value2 }">Example</div>
```

In this example, the **ngStyle** directive applies the CSS styles **property1** and **property2** to the **div** element with their corresponding values **value1** and **value2**.

Here's a more detailed example that demonstrates how to use **ngStyle** to dynamically apply inline styles based on user interaction:

```
<button [ngStyle]="{ 'background-color': isActive ? 'green' : 'red' }"
(click)="toggleActive()">Toggle Active</button>
```

In this example, the **ngStyle** directive applies the inline style **background-color** to the **button** element based on the value of the **isActive** property. If **isActive** is true, the button's background color will be green, and if it is false, the background color will be red. The **isActive** property is initially set to false in the component:

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

@Component({
  selector: 'app-button',
  template: `
    <button [ngStyle]="{ 'background-color': isActive ? 'green' : 'red' }"
    (click)="toggleActive()">Toggle Active</button>
  `
})
export class ButtonComponent {
  isActive = false;

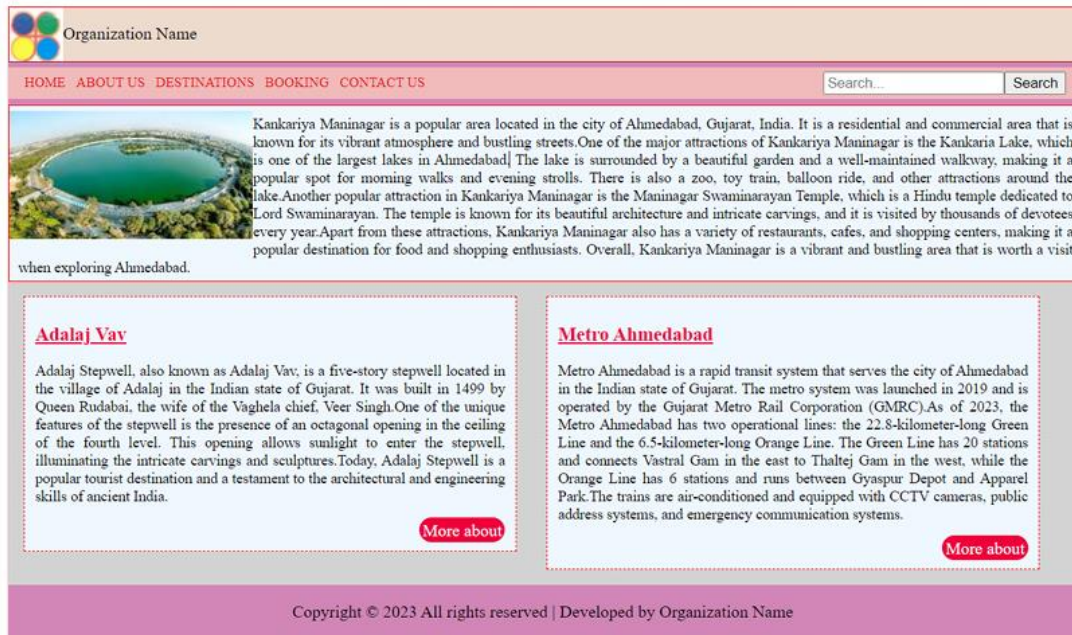
  toggleActive() {
    this.isActive = !this.isActive;
  }
}
```

When the user clicks the button, the **toggleActive()** method is called, which toggles the value of the **isActive** property. When **isActive** is true, the button's background color will be green, and when **isActive** is false, the background color will be red.

You can also use **ngStyle** in combination with other directives, such as **ngFor** or **ngIf**, to dynamically apply different inline styles to elements based on their state or data.

J. Practical related Exercises.

1. Design following tourist package webpage using various components of angular or teacher can assign any webpage of website.



Practical No.5

Aim: Design webpage using *ngIf,*ngFor,*ngswitch and ngStyle Directives.

A. Objective:

Manipulate the structure of the DOM (Document Object Model) based on conditions and other logical statements.

B. Expected Program Outcomes (POs)

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the *engineering* problems.

2. **Problem analysis:** Identify and analyse well-defined *engineering* problems using codified standard methods.

3. **Design/ development of solutions:** Design solutions for *engineering* well-defined technical problems and assist with the design of systems components or processes to meet specified needs.

4. **Engineering Tools, Experimentation and Testing:** Apply modern *engineering* tools and appropriate technique to conduct standard tests and measurements.

5. **Life-long learning:** Ability to analyze individual needs and engage in updating in the context of technological changes *in field of engineering*.

C. Expected Skills to be developed based on competency:

Learner can get various skills such as understanding of basic Angular concepts, Familiarity with Type Script, Knowledge of Angular directives, Experience with Angular CLI.

D. Expected Course Outcomes(Cos)

Apply angular directives, components and pipes in different web page development.

E. Practical Outcome(PRo)

Design a web page to display student grading system in tabular format with alternate color style using ngSwitch, ngStyle Directives.

F. Expected Affective domain Outcome(ADos)

1. Follow Coding standards and practices.
2. Maintain tools and equipment.
3. Follow safety practices.
4. Follow ethical practices

G. Prerequisite Theory:

Structural directives in Angular are a type of directive that allow you to modify the structure of the DOM based on certain conditions or states. They are used to add, remove, or update elements in the DOM based on the result of an expression or the state of the application.

There are three common structural directives in Angular:

1. ngIf: The ngIf directive is used to conditionally add or remove an element from the DOM based on a given expression. If the expression is true, the element is added to the DOM, and if it is false, the element is removed from the DOM.
2. ngFor: The ngFor directive is used to repeat a section of HTML code for each item in an array or collection. It can be used to loop through an array of objects, an array of strings, or any other iterable object.
3. ngSwitch: The ngSwitch directive is used to conditionally display content based on a given expression. It allows you to define a set of possible values and associate each value with a template to render.

Structural directives are identified by the prefix "ng" followed by the directive name. They are enclosed in square brackets and are typically used in combination with other directives and HTML tags to create dynamic and responsive user interfaces.

***ngIf Directory Structure**

*ngIf is a structural directive in Angular that allows you to conditionally render or remove HTML elements based on an expression. The directive evaluates the expression passed to it and renders the element if the expression is truthy, otherwise it removes the element from the DOM.

In addition to `*ngIf`, there is also an `*ngIf-else` directive that allows you to specify an alternative template to render when the expression evaluates to false. The syntax for `*ngIf-else` is as follows:

```
<div *ngIf="condition; else elseBlock">
  <!-- content to show when condition is true -->
</div>
<ng-template #elseBlock>
  <!-- content to show when condition is false -->
</ng-template>
```

In this example, the **div** element will be rendered if the **condition** expression is truthy. If **condition** is falsy, the **div** element will be removed from the DOM and the **elseBlock** template will be rendered instead.

The **elseBlock** template is defined using the **ng-template** element with a template reference variable of **#elseBlock**. This allows us to refer to the template later in the `*ngIf-else` directive.

You can also use multiple `*ngIf-else` directives to conditionally render different templates based on different expressions. Here's an example:

```
<div *ngIf="condition1; else elseBlock1">
  <!-- content to show when condition1 is true -->
</div>
<ng-template #elseBlock1>
  <div *ngIf="condition2; else elseBlock2">
    <!-- content to show when condition1 is false and condition2 is true -->
  </div>
  <ng-template #elseBlock2>
    <!-- content to show when both condition1 and condition2 are false -->
  </ng-template>
</ng-template>
```

In this example, we first check **condition1**. If it is true, we render the first **div** element. If it is false, we check **condition2**. If **condition2** is true, we render the second **div** element. If both **condition1** and **condition2** are false, we render the content in the second **ng-template**.

ngFor Directory

***ngFor** is a structural directive in Angular that iterates over a collection and creates a template for each item in the collection. It is commonly used to render a list of items or to repeat a section of a template.

The basic syntax of ***ngFor** is:

```
*ngFor="let item of collection"
```

where **item** is a variable that represents each item in the collection, and **collection** is an array or any iterable object.

Here are some examples of how ***ngFor** can be used in Angular:

1. Rendering a list of items

```
<ul>
<li *ngFor="let item of items">{{ item }}</li>
</ul>
```

```
items=['apple', 'banana', 'orange']
```

In this example, **items** is an array of strings that is declared in ts file, and ***ngFor** creates an **li** element for each item in the array.

2. Looping over an array of objects

```
<table>
<tr *ngFor="let user of users">
<td>{{ user.name }}</td>
<td>{{ user.age }}</td>
</tr>
</table>
```

```
users = [
  {
    name: 'chintan',
    age: 36
  },
  {
    name: 'mohit',
```

```
age: 45
  },
  {
    name: 'hiral',
    age: 25
  },
]
```

In this example, **users** is an array of objects that is declared in ts file, and ***ngFor** creates a **tr** element for each object in the array. The **td** elements contain the **name** and **age** properties of each object.

3: Using index for conditional rendering

```
<div *ngFor="let item of items; index as i">
<div *ngIf="i % 2 == 0" class="even-item">{{ item }}</div>
<div *ngIf="i % 2 != 0" class="odd-item">{{ item }}</div>
</div>
```

In this example, we are using the "index as i" syntax to access the current index of the loop iteration and use it for conditional rendering. We are displaying each item on a new line, but alternating the background color based on whether the index is even or odd.

Overall, *ngFor with index is a powerful tool in Angular that allows developers to loop through collections and access the current index of the loop iteration for various use cases.

***ngSwitch**

The *ngSwitch directive is used in Angular to conditionally display content based on a specified expression. It's similar to a switch statement in programming languages. Here's an example of how to use *ngSwitch in an Angular template file:

```
<div [ngSwitch]="dayOfWeek">
<div *ngSwitchCase="Monday">It's Monday!</div>
<div *ngSwitchCase="Tuesday">It's Tuesday!</div>
<div *ngSwitchCase="Wednesday">It's Wednesday!</div>
<div *ngSwitchCase="Thursday">It's Thursday!</div>
<div *ngSwitchCase="Friday">It's Friday!</div>
<div *ngSwitchCase="Saturday">It's Saturday!</div>
<div *ngSwitchCase="Sunday">It's Sunday!</div>
<div *ngSwitchDefault>Invalid day of the week!</div>
</div>
```

In the above example, we have a div with the attribute **[ngSwitch]="dayOfWeek"**, where **dayOfWeek** is a property in the component class. Inside this div, we have several divs that use the ***ngSwitchCase** directive to display different content based on the value of **dayOfWeek**. The ***ngSwitchDefault** directive is used to specify the default case if none of the ***ngSwitchCase** expressions match.

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

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

In the component class, we have a property called **dayOfWeek** that is set to **'Monday'**. Based on this value, the template will display the div with the ***ngSwitchCase="'Monday'"** directive.

Practical related Exercises.

1. Design webpage to displaying different images based on a selected option.

Practical No.6

Aim: Design component to perform following tasks

- [1] To Add or Remove number of students using textbox and button controls and display it in tabular structure format.
- [2] Give row level remove button option to student table and record should be deleted when click on it.

A. Objective:

The learner will help to manipulate data on web page based on event , conditions and generate desired outputs.

B. Expected Program Outcomes (POs)

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the *engineering* problems.
2. **Problem analysis:** Identify and analyse well-defined *engineering* problems using codified standard methods.
3. **Design/ development of solutions:** Design solutions for *engineering* well-defined technical problems and assist with the design of systems components or processes to meet specified needs.
4. **Engineering Tools, Experimentation and Testing:** Apply modern *engineering* tools and appropriate technique to conduct standard tests and measurements.
5. **Life-long learning:** Ability to analyze individual needs and engage in updating in the context of technological changes *in field of engineering*.

C. Expected Skills to be developed based on competency:

Learner should have knowledge of how to Dynamic rendering of lists, efficiently adding and removing elements on web page and Familiarity with template syntax.

D. Expected Course Outcomes(Cos)

Apply angular directives, components and pipes in different web page development.

E. Practical Outcome(Pro)

Learner will be able to Add or Remove number of students using textbox and button controls and display it in tabular structure format along with give row level remove button option to student table and record should be deleted when click on it.

.

F. Expected Affective domain Outcome(ADos)

1. Follow Coding standards and practices.
2. Maintain tools and equipment.
3. Follow safety practices.
4. Follow ethical practices

G. Prerequisite Theory:

Template reference variable

Template reference variable is a variable that you can assign to a template element or component using the # symbol. This allows you to reference the element or component in your template code and in your component code.

Here's an example of using a template reference variable to reference an input element in a template:

```
<input type="text" #nameInput>
```

```
<button (click)="greet(nameInput.value)">Greet</button>
```

In this template, we're using the **#nameInput** syntax to create a template reference variable that we can use to reference the **input** element. We're also using an event binding to call a **greet** method on our component when a button is clicked.

In our component, we can access the value of the input element using the template reference variable:

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

```
@Component({
```

```
  selector: 'app-greeter',
```

```
  templateUrl: './greeter.component.html',
```

```
  styleUrls: ['./greeter.component.css']
```

```
  })  
  
  export class GreeterComponent {  
  
    greet(name: string) {  
  
      alert(`Hello, ${name}!`);  
  
    }  
  
  }  
}
```

In this component, we've defined a **greet** method that takes a **name** argument and displays an alert message with the name. When the button in the template is clicked, the **greet** method is called with the value of the **nameInput** template reference variable as the argument.

PUSH method

The **push** method in Angular is used to add elements to an array. It's a built-in method of the JavaScript **Array** object, and it can be used in Angular components to manipulate arrays and update the view.

Here's an example of using the **push** method in an Angular component:

```
import { Component } from '@angular/core';  
  
@Component({  
  selector: 'app-product-list',  
  templateUrl: './product-list.component.html',  
  styleUrls: ['./product-list.component.css']  
})  
export class ProductListComponent {  
  products: string[] = [];  
  
  addProduct(productName: string) {  
    this.products.push(productName);  
  }  
}
```

In this example, we've defined a **products** array in our component and initialized it to an empty array. We've also defined an **addProduct** method that takes a **productName** argument and adds it to the **products** array using the **push** method.

Now, in our component's template, we can use **ngFor** to iterate over the **products** array and display each product in a list:

```
<ul>
<li *ngFor="let product of products">{{ product }}</li>
</ul>

<input type="text" #productName>
<button (click)="addProduct(productName.value)">Add Product</button>
```

In this template, we're using **ngFor** to iterate over the **products** array and display each product as a list item. We're also using an input and a button to capture new product names and call the **addProduct** method when the button is clicked.

When the **addProduct** method is called, it uses the **push** method to add the new product to the **products** array. Because the view is bound to the **products** array using **ngFor**, the view is automatically updated to show the new product in the list.

Splice method

The splice method in Angular is used to add or remove elements from an array at a specified index. It's a built-in method of the JavaScript Array object, and it can be used in Angular components to manipulate arrays and update the view.

Here's an example of using the splice method in an Angular component:

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

@Component({
  selector: 'app-product-list',
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
export class ProductListComponent {
  products: string[] = ['apple', 'banana', 'orange'];

  removeProduct(index: number) {
    this.products.splice(index, 1);
  }
}
```



```
}
```

In this example, we've defined a **products** array in our component and initialized it with three products. We've also defined a **removeProduct** method that takes an **index** argument and removes the product at that index from the **products** array using the **splice** method.

Now, in our component's template, we can use **ngFor** to iterate over the **products** array and display each product in a list. We can also use a button to call the **removeProduct** method with the index of the product we want to remove:

```
<ul>
<li *ngFor="let product of products; let i = index">{{ product }}
<button (click)="removeProduct(i)">Remove</button>
</li>
</ul>
```

In this template, we're using **ngFor** to iterate over the **products** array and display each product as a list item. We're also using a button to call the **removeProduct** method with the index of the product we want to remove.

When the **removeProduct** method is called, it uses the **splice** method to remove the product at the specified index from the **products** array. Because the view is bound to the **products** array using **ngFor**, the view is automatically updated to remove the product from the list.

Practical No 7

Aim: Create a component to display a products list from array. the product component should display a product Id, name, purchase date, price, and image for the product and search using various pipes.

A. Objective:

Simplify the process of displaying data in a format that is appropriate for the user by using custom pipe and you can reduce the amount of code needed in your template , improve the performance and usability of your application by using filter pipe.

.

B. Expected Program Outcomes (POs)

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the *engineering* problems.

2. **Problem analysis:** Identify and analyse well-defined *engineering* problems using codified standard methods.

3. **Design/ development of solutions:** Design solutions for *engineering* well-defined technical problems and assist with the design of systems components or processes to meet specified needs.

4. **Engineering Tools, Experimentation and Testing:** Apply modern *engineering* tools and appropriate technique to conduct standard tests and measurements.

5. **Life-long learning:** Ability to analyze individual needs and engage in updating in the context of technological changes *in field of engineering*.

C. Expected Skills to be developed based on competency:

Learner can develop skills in data filtering, working with arrays and objects, applying filter conditions. These skills are valuable for building robust and efficient Angular applications.

D. Expected Course Outcomes(Cos)

Apply angular directives, components and pipes in different web page development.

E. Practical Outcome(Pro)

Learner will be able to display a products list from array and search required product using filter, custom pipes

.

F. Expected Affective domain Outcome(ADos)

1. Follow Coding standards and practices.
2. Maintain tools and equipment.
3. Follow safety practices.
4. Follow ethical practices

G. Prerequisite Theory:

Pipes are used to format and transform data in a template. Pipes take in an input value, transform it, and then return the transformed value. There are two types of pipes available in Angular:

1. Built-in pipes - These are provided by Angular.

Here is a table listing the built-in pipes in Angular along with a brief description of each pipe:

Pipe	Description
DatePipe	Used to format a date according to a specified format string and locale.
DecimalPipe	Used to format a number with a fixed number of digits before and after the decimal point.
LowerCasePipe	Used to convert a string to lowercase.
PercentPipe	Used to format a number as a percentage.
SlicePipe	Used to create a new array or string containing a subset of the elements of the input array or string.
TitleCasePipe	Used to convert a string to title case, where the first letter of each word is capitalized.
UpperCasePipe	Used to convert a string to uppercase.

2. Custom pipes - These are created by developers and can be used to transform data in a specific way.

Custom pipes are useful when you need to transform data in a specific way that is not provided by the built-in pipes. In addition to custom pipes, Angular also provides filter pipes. Filter pipes are used to filter data based on a specific criteria.

Here is an example of using the built-in UpperCasePipe in Angular:

```
<!-- app.component.html -->
<p>{{ 'hello world' | uppercase }}</p>
```

In this example, the string 'hello world' is passed to the uppercase pipe, which transforms it to 'HELLO WORLD'. The transformed string is then displayed in the template using string interpolation.

Here is an example of creating a custom pipe that appends an exclamation mark to a string:

```
// app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { AddExclamationPipe } from './add-exclamation.pipe';
```

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

```
// add-exclamation.pipe.ts
import { Pipe, PipeTransform } from '@angular/core';
```

```
@Pipe({
```

```
name: 'addExclamation'
})
export class AddExclamationPipe implements PipeTransform {
  transform(value: string): string {
    return value + '!';
  }
}

<!-- app.component.html -->
<p>{{ 'hello world' | addExclamation }}</p>
```

In this example, a custom pipe called `AddExclamationPipe` is created using the `Pipe` decorator. The pipe takes in a string value, appends an exclamation mark to it, and then returns the transformed value. The custom pipe is then used in the template by passing the string 'hello world' to it.

here's an example of how to use the **filter** pipe in Angular to filter an array of objects, including the relevant HTML, TypeScript, and pipe files, as well as the app module file:

```
<input type="text" [(ngModel)]="searchQuery">
<ul>
<li *ngFor="let item of items | filter: searchQuery">
  {{ item.name }}
</li>
</ul>
```

In this example, we have an input field where the user can type in a search query. We also have a list of items that we want to filter based on the search query. We use the **filter** pipe to filter the items based on the search query.

TypeScript:

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

@Component({
  selector: 'app-item-list',
  templateUrl: './item-list.component.html',
  styleUrls: ['./item-list.component.css']
})
export class ItemListComponent {
  items = [
    { id: 1, name: 'Item 1' },
```

```
{ id: 2, name: 'Item 2' },  
{ id: 3, name: 'Item 3' },  
{ id: 4, name: 'Item 4' },  
{ id: 5, name: 'Item 5' }  
];  
searchQuery: string = ""  
}
```

In the TypeScript file, we define an array of items that we want to filter. We also define a **searchQuery** variable to hold the search query entered by the user.

Filter Pipe:

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

```
@Pipe({  
  name: 'filter'  
})  
export class FilterPipe implements PipeTransform {  
  transform(items: any[], searchText: string): any[] {  
    if (!items) {  
      return [];  
    }  
    if (!searchText) {  
      return items;  
    }  
    searchText = searchText.toLowerCase();  
    return items.filter(item => {  
      return item.name.toLowerCase().includes(searchText);  
    });  
  }  
}
```

In the filter pipe, we define a custom **filter** pipe to filter the items based on the search query. The pipe takes in an array of items and a search query, and returns a filtered array based on the search query. The filter function checks if each item's name includes the search query and returns only those items that match.

App Module:

```
import { NgModule } from '@angular/core';  
import { BrowserModule } from '@angular/platform-browser';  
import { FormsModule } from '@angular/forms';  
import { FilterPipe } from './filter.pipe';
```

```
@NgModule({  
  declarations: [  
    FilterPipe,  
    ItemListComponent  
  ],  
  imports: [  
    BrowserModule,  
    FormsModule  
  ],  
  providers: [],  
  bootstrap: []  
})
```

```
export class AppModule { }
```

In the app module, we import the **FilterPipe** and declare them in the declarations array.

Practical No 8

Aim :Design a student registration page using template driven form approach and utilize different form and controls level ng validation classes.

A. Objective:

To create template driven form that is used in design small to medium level application. It allows us to create custom form controls classes, apply validate error messages and quickly create forms with minimal coding to submit data.

B. Expected Program Outcomes (POs)

1. **Basic and Discipline specific knowledge:** Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the *engineering* problems.

2. **Problem analysis:** Identify and analyse well-defined *engineering* problems using codified standard methods.

3. **Design/ development of solutions:** Design solutions for *engineering* well-defined technical problems and assist with the design of systems components or processes to meet specified needs.

4. **Engineering Tools, Experimentation and Testing:** Apply modern *engineering* tools and appropriate technique to conduct standard tests and measurements.

C. Expected Skills to be developed based on competency:

Learner can develop skills related to HTML, Angular directives, TypeScript, two-way data binding, custom form controls, and form validation. These skills can be valuable for building forms and various types of Angular applications.

D. Expected Course Outcomes(Cos)

Utilize angular template driven and reactive forms in different problem solutions.

E. Practical Outcome(PRo)

Learner will be able to create template driven form approach for angular application and apply different ng validation classes, custom validation message, respond to form submission.

F. Expected Affective domain Outcome(ADos)

1. Follow Coding standards and practices.
2. Maintain tools and equipment.
3. Follow safety practices.
4. Follow ethical practices

G. Prerequisite Theory:

Template-driven forms is a way to create forms in Angular using templates, directives, and data binding. With template-driven forms, the form layout and behavior are defined in the HTML template, using Angular directives, rather than in the component class.

Template-driven forms are ideal for creating simple forms that don't require complex form logic or dynamic form controls. They are easy to use and require minimal coding, making them a good choice for prototyping or creating small forms.

FormsModule :

The **FormsModule** is a built-in module that provides a set of directives, services, and pipes for creating and managing forms. It is an essential module for working with forms in Angular and is included by default in most Angular applications.

The **FormsModule** provides several features that make it easy to work with forms in Angular, including:

1. Two-way data binding: The **FormsModule** provides the **ngModel** directive, which enables two-way data binding between form controls and component properties.
2. Validation: The **FormsModule** provides a set of validation directives, such as **ngRequired**, **ngMinLength**, and **ngPattern**, that can be used to validate form inputs.
3. Form submission: The **FormsModule** provides the **ngSubmit** directive, which is used to handle form submission events and trigger form validation.
4. Form control status tracking: The **FormsModule** provides the **ngModelGroup** and **ngForm** directives, which can be used to group form controls and track their status, such as validity and touched status.

To use the **FormsModule** in an Angular application, you need to import it in the root module of your application, typically in the **app.module.ts** file, like this:

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

```
@NgModule({  
  imports: [  
    // other modules  
    FormsModule
```

```
],  
  // other configuration  
})  
export class AppModule { }
```

Once imported, you can use the **FormsModule** directives, services, and pipes in your components to create and manage forms in Angular.

ngModel, ngForm, and ngSubmit directives

let's look at the **ngModel**, **ngForm**, and **ngSubmit** directives purposes are given below.

1. **ngModel Directive:** The **ngModel** directive is used to bind the form control values to the component properties, enabling two-way data binding between the form and the component. In the example, we use the **ngModel** directive to bind the input values to the **name** and **email** component properties.
2. **ngForm Directive:** The **ngForm** directive is used to define the form and track its state, such as validity and touched status. In the example, we use the **ngForm** directive with the **#myForm** template reference variable to define the form and track its state.
3. **ngSubmit Directive:** The **ngSubmit** directive is used to handle the form submission event and trigger form validation. In the example, we use the **ngSubmit** directive to call the **submitForm()** method when the form is submitted.

Overall, the combination of these directives enables us to create and manage forms in Angular with minimal coding and powerful functionality.

let's take an example of the **ngModel**, **ngForm**, and **ngSubmit** directives.

HTML Template:

```
<form #myForm="ngForm" (ngSubmit)="submitForm(myForm)">  
<div>  
<label for="name">Name:</label>  
<input type="text" name="name" [(ngModel)]="name" required>  
</div>  
<div>  
<label for="email">Email:</label>  
<input type="email" name="email" [(ngModel)]="email" required>  
</div>
```

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

In this example, the form contains two form controls, a text input and an email input, both of which are required. The **ngModel** directive is used to bind the input values to the **name** and **email** component properties, enabling two-way data binding. The **ngForm** directive is used to define the form and track its state. Finally, the **ngSubmit** directive is used to handle the form submission event and trigger form validation.

TypeScript File:

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

@Component({
  selector: 'app-my-form',
  templateUrl: './my-form.component.html',
  styleUrls: ['./my-form.component.css']
})
export class MyFormComponent {
  name: string;
  email: string;

  submitForm(form: any): void {
    console.log(form);
  }
}
```

In this example, we have created a component called **MyFormComponent** that defines the component's behavior and interacts with the template. The component has two properties, **name** and **email**, which are used to store the input values. The **submitForm()** method is called when the form is submitted and receives the **myForm** object as a parameter, which is an instance of the **NgForm** class. The method logs the form object to the console.

form validation state

In template-driven forms, form validation is an essential feature that ensures that user inputs are correct before submitting the form. The validation process involves tracking the state of each form control and providing feedback to the user about the validity of their inputs.

There are several states that a form control can be in, including:

1. untouched: The form control has not been touched by the user yet.
2. touched: The form control has been touched by the user.
3. pristine: The form control has not been modified by the user yet.
4. dirty: The form control has been modified by the user.
5. valid: The form control value is valid according to the validation rules.
6. invalid: The form control value is invalid according to the validation rules.

To display the validation state of a form control, you can use CSS classes that reflect the control's state. Angular provides several CSS classes for this purpose, including **ng-untouched**, **ng-touched**, **ng-pristine**, **ng-dirty**, **ng-valid**, and **ng-invalid**.

HTML Template

The HTML template is where we define the form layout and validation directives that will be used to validate the form input. In the following example, we have a simple form that requires a name and an email, with validation for both fields.

```
<form #myForm="ngForm" (ngSubmit)="onSubmit(myForm)">
<div>
<label for="name">Name:</label>
<input type="text" name="name" [(ngModel)]="model.name" required minlength="3"
maxlength="50" #nameInput="ngModel">
<div *ngIf="nameInput.invalid && (nameInput.dirty || nameInput.touched)">
<div *ngIf="nameInput.errors.required">Name is required.</div>
<div *ngIf="nameInput.errors.minlength">Name must be at least 3 characters long.</div>
<div *ngIf="nameInput.errors.maxlength">Name cannot be longer than 50
characters.</div>
</div>
</div>
<div>
<label for="email">Email:</label>
<input type="email" name="email" [(ngModel)]="model.email" required email
#emailInput="ngModel">
<div *ngIf="emailInput.invalid && (emailInput.dirty || emailInput.touched)">
<div *ngIf="emailInput.errors.required">Email is required.</div>
<div *ngIf="emailInput.errors.email">Email must be a valid email address.</div>
</div>
</div>
<button type="submit" [disabled]="myForm.invalid">Submit</button>
</form>
```

In this example, we define the form using the **ngForm** directive, which allows us to track the form state and handle the form submission. We then define two input fields, one for the name and one for the email, using the **ngModel** directive to bind the input values to the **model** object. We also add several validation directives to each input field, including **required**, **minlength**, **maxlength**, and **email**.

To display the validation messages, we use the ***ngIf** directive to check if the input field is invalid and has been touched or is dirty. If this condition is met, we display the appropriate validation message.

Finally, we disable the submit button if the form is invalid, using the **[disabled]** directive.

CSS Styling

The CSS styling for the form validation is entirely optional, but it can help to provide visual feedback to the user when the form is in an invalid state. In the following example, we add some simple styling to display the validation messages in red text.

```
input.ng-invalid {  
  border-color: red;  
}
```

```
.error-message {  
  color: red;  
}
```

This CSS styling changes the border color of an input field when it is in an invalid state, making it more visually distinct. It also applies a red color to the validation messages, making them more noticeable.

TypeScript Code

Finally, we need to add some TypeScript code to handle the form submission and set the **model** object. In the following example, we define a **model** object in the component class and add an **onSubmit()** method to handle the form submission.

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

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

```

export class FormComponent {
  model = {
    name: "",
    email: ""
  };
  onSubmit (form: NgForm) {
    if (form.valid) {
      // form is valid, submit data
    }
  }
}

```

In this example, we define a model object with **name** and **email** properties in the component class. We use two-way data binding with the **[(ngModel)]** directive to bind the input values to the properties of the model object.

We also define an **onSubmit()** method in the component class that is called when the form is submitted. This method can perform other actions such as submitting the form data to a server.

Note that we use the **NgForm** class to access the form data in the **onSubmit()** method. We pass the **myForm** template reference variable to the method to access the form data.

Form Submission Data

In template-driven forms in Angular, you can submit form data and display it in the template using template reference variables and data binding.

Here is an example of how to submit form data and display it in the template:

```

<form #myForm="ngForm" (ngSubmit)="onSubmit(myForm)">
  <div class="form-group">
    <label for="name">Name:</label>
    <input type="text" name="name" [(ngModel)]="model.name" required>
  </div>
  <div class="form-group">
    <label for="email">Email:</label>
    <input type="email" name="email" [(ngModel)]="model.email" required email>
  </div>
  <button type="submit">Submit</button>
</form>

```

```
<div *ngIf="submitted">
<h3>Form Data</h3>
<p>Name: {{ model.name }}</p>
<p>Email: {{ model.email }}</p>
</div>
```

Typescript file

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

@Component({
  selector: 'app-my-form',
  templateUrl: './my-form.component.html',
  styleUrls: ['./my-form.component.css']
})
export class MyFormComponent {
  model = {
    name: "",
    email: ""
  };
  submitted = false;

  onSubmit(form: NgForm) {
    console.log(this.model);
    // submit form data to server or perform other actions
    this.submitted = true;
  }
}
```

In this example, we define a **submitted** property in the component class that is initially set to **false**. We also add a ***ngIf** directive to the **div** element that displays the form data to only display it when **submitted** is **true**.

In the **onSubmit()** method, we set **submitted** to **true** after performing any necessary actions with the form data.

When the form is submitted, the **submitted** property is set to **true** and the form data is displayed in the template. The **{{ }}** syntax is used for data binding to display the values of the **name** and **email** properties of the **model** object.

Note that this is just one example of how to submit and display form data in template-driven forms in Angular. There are other ways to achieve this functionality depending on your specific requirements.