INDEX

Angular Architecture overview

Angular is a platform and framework for building client applications in HTML and TypeScript. Angular is written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you import into your apps.

The basic building blocks of an Angular application are *NgModules*, which provide a compilation context for *components*. NgModules collect related code into functional sets; an Angular app is defined by a set of NgModules. An app always has at least a *root module* that enables bootstrapping, and typically has many more *feature modules*.

* Components define *views*, which are sets of screen elements that Angular can choose among and modify according to your program logic and data.
* Components use *services*, which provide specific functionality not directly related to views. Service providers can be *injected* into components as *dependencies*, making your code modular, reusable, and efficient.

Both components and services are simply classes, with *decorators* that mark their type and provide metadata that tells Angular how to use them.

* The metadata for a component class associates it with a *template* that defines a view. A template combines ordinary HTML with Angular *directives* and *binding markup* that allow Angular to modify the HTML before rendering it for display.
* The metadata for a service class provides the information Angular needs to make it available to components through *dependency injection (DI)*.

An app's components typically define many views, arranged hierarchically. Angular provides the [Router](https://angular.io/api/router/Router) service to help you define navigation paths among views. The router provides sophisticated in-browser navigational capabilities.

Modules

Angular *NgModules* differ from and complement JavaScript (ES2015) modules. An NgModule declares a compilation context for a set of components that is dedicated to an application domain, a workflow, or a closely related set of capabilities. An NgModule can associate its components with related code, such as services, to form functional units.

Every Angular app has a *root module*, conventionally named AppModule, which provides the bootstrap mechanism that launches the application. An app typically contains many functional modules.

Like JavaScript modules, NgModules can import functionality from other NgModules, and allow their own functionality to be exported and used by other NgModules. For example, to use the router service in your app, you import the [Router](https://angular.io/api/router/Router) NgModule.

Organizing your code into distinct functional modules helps in managing development of complex applications, and in designing for reusability. In addition, this technique lets you take advantage of *lazy-loading*—that is, loading modules on demand—to minimize the amount of code that needs to be loaded at startup.

## Components

Every Angular application has at least one component, the root component that connects a component hierarchy with the page document object model (DOM). Each component defines a class that contains application data and logic, and is associated with an HTML template that defines a view to be displayed in a target environment.

The @[Component](https://angular.io/api/core/Component)() decorator identifies the class immediately below it as a component, and provides the template and related component-specific metadata.

### Templates, directives, and data binding

A template combines HTML with Angular markup that can modify HTML elements before they are displayed. Template directives provide program logic, and binding markup connects your application data and the DOM. There are two types of data binding:

* Event binding lets your app respond to user input in the target environment by updating your application data.
* Property binding lets you interpolate values that are computed from your application data into the HTML.

Before a view is displayed, Angular evaluates the directives and resolves the binding syntax in the template to modify the HTML elements and the DOM, according to your program data and logic. Angular supports two-way data binding, meaning that changes in the DOM, such as user choices, are also reflected in your program data.

Your templates can use pipes to improve the user experience by transforming values for display. For example, use pipes to display dates and currency values that are appropriate for a user's locale. Angular provides predefined pipes for common transformations, and you can also define your own pipes.

## Services and dependency injection

For data or logic that isn't associated with a specific view, and that you want to share across components, you create a service class. A service class definition is immediately preceded by the @[Injectable](https://angular.io/api/core/Injectable)() decorator. The decorator provides the metadata that allows other providers to be **injected** as dependencies into your class.

Dependency injection (DI) lets you keep your component classes lean and efficient. They don't fetch data from the server, validate user input, or log directly to the console; they delegate such tasks to services.

### Routing

The Angular [Router](https://angular.io/api/router/Router) NgModule provides a service that lets you define a navigation path among the different application states and view hierarchies in your app. It is modeled on the familiar browser navigation conventions:

* Enter a URL in the address bar and the browser navigates to a corresponding page.
* Click links on the page and the browser navigates to a new page.
* Click the browser's back and forward buttons and the browser navigates backward and forward through the history of pages you've seen.

The router maps URL-like paths to views instead of pages. When a user performs an action, such as clicking a link, that would load a new page in the browser, the router intercepts the browser's behavior, and shows or hides view hierarchies.

If the router determines that the current application state requires particular functionality, and the module that defines it hasn't been loaded, the router can lazy-load the module on demand.

The router interprets a link URL according to your app's view navigation rules and data state. You can navigate to new views when the user clicks a button or selects from a drop box, or in response to some other stimulus from any source. The router logs activity in the browser's history, so the back and forward buttons work as well.

To define navigation rules, you associate navigation paths with your components. A path uses a URL-like syntax that integrates your program data, in much the same way that template syntax integrates your views with your program data. You can then apply program logic to choose which views to show or to hide, in response to user input and your own access rules.

**You've learned the basics about the main building blocks of an Angular application. The following diagram shows how these basic pieces are related.**



* Together, a component and template define an Angular view.
  + A decorator on a component class adds the metadata, including a pointer to the associated template.
  + Directives and binding markup in a component's template modify views based on program data and logic.
* The dependency injector provides services to a component, such as the router service that lets you define navigation among views.

Each of these subjects is introduced in more detail in the following pages.

Introduction to modules

Angular apps are modular and Angular has its own modularity system called *NgModules*. NgModules are containers for a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities. They can contain components, service providers, and other code files whose scope is defined by the containing NgModule. They can import functionality that is exported from other NgModules, and export selected functionality for use by other NgModules.

Every Angular app has at least one NgModule class, [the *root module*](https://angular.io/guide/bootstrapping), which is conventionally named AppModule and resides in a file named app.module.ts. You launch your app by *bootstrapping* the root NgModule.

While a small application might have only one NgModule, most apps have many more *feature modules*. The *root* NgModule for an app is so named because it can include child NgModules in a hierarchy of any depth.

NgModule metadata

An NgModule is defined by a class decorated with @[NgModule](https://angular.io/api/core/NgModule)(). The @[NgModule](https://angular.io/api/core/NgModule)() decorator is a function that takes a single metadata object, whose properties describe the module. The most important properties are as follows.

* declarations: The [components](https://angular.io/guide/architecture-components), *directives*, and *pipes* that belong to this NgModule.
* exports: The subset of declarations that should be visible and usable in the *component templates* of other NgModules.
* imports: Other modules whose exported classes are needed by component templates declared in *this* NgModule.
* providers: Creators of [services](https://angular.io/guide/architecture-services) that this NgModule contributes to the global collection of services; they become accessible in all parts of the app. (You can also specify providers at the component level, which is often preferred.)
* bootstrap: The main application view, called the *root component*, which hosts all other app views. Only the *root NgModule* should set the bootstrap property.

Here's a simple root NgModule definition.

src/app/app.module.ts

import { [NgModule](https://angular.io/api/core/NgModule) } from '@angular/core';

import { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';

@[NgModule](https://angular.io/api/core/NgModule)({

imports: [ [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) ],

providers: [ Logger ],

declarations: [ AppComponent ],

exports: [ AppComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

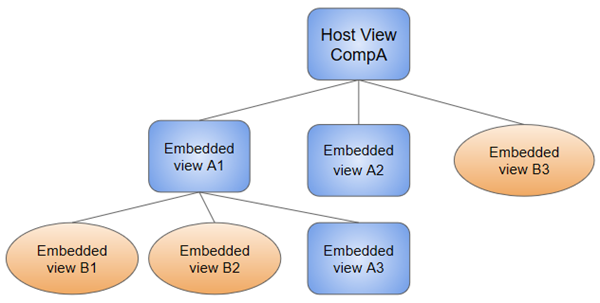
AppComponent is included in the exports list here for illustration; it isn't actually necessary in this example. A root NgModule has no reason to *export* anything because other modules don't need to *import* the root NgModule.

NgModules and components

NgModules provide a *compilation context* for their components. A root NgModule always has a root component that is created during bootstrap, but any NgModule can include any number of additional components, which can be loaded through the router or created through the template. The components that belong to an NgModule share a compilation context.



A component and its template together define a *view*. A component can contain a *view hierarchy*, which allows you to define arbitrarily complex areas of the screen that can be created, modified, and destroyed as a unit. A view hierarchy can mix views defined in components that belong to different NgModules. This is often the case, especially for UI libraries.



When you create a component, it's associated directly with a single view, called the *host view*. The host view can be the root of a view hierarchy, which can contain *embedded views*, which are in turn the host views of other components. Those components can be in the same NgModule, or can be imported from other NgModules. Views in the tree can be nested to any depth.

Note: The hierarchical structure of views is a key factor in the way Angular detects and responds to changes in the DOM and app data.

# Introduction to components

A component controls a patch of screen called a view.

You define a component's application logic—what it does to support the view—inside a class. The class interacts with the view through an API of properties and methods.

For example, HeroListComponent has a heroes property that holds an array of heroes. Its selectHero() method sets a selectedHero property when the user clicks to choose a hero from that list. The component acquires the heroes from a service, which is a TypeScript [parameter property](http://www.typescriptlang.org/docs/handbook/classes.html#parameter-properties) on the constructor. The service is provided to the component through the dependency injection system.

src/app/hero-list.component.ts (class)

export class HeroListComponent implements [OnInit](https://angular.io/api/core/OnInit) {

heroes: Hero[];

selectedHero: Hero;

constructor(private service: HeroService) { }

ngOnInit() {

this.heroes = this.service.getHeroes();

}

selectHero(hero: Hero) { this.selectedHero = hero; }

}

Angular creates, updates, and destroys components as the user moves through the application. Your app can take action at each moment in this lifecycle through optional [lifecycle hooks](https://angular.io/guide/lifecycle-hooks), like ngOnInit().

## Component metadata

Metadata

The @[Component](https://angular.io/api/core/Component) decorator identifies the class immediately below it as a component class, and specifies its metadata. In the example code below, you can see that HeroListComponent is just a class, with no special Angular notation or syntax at all. It's not a component until you mark it as one with the @[Component](https://angular.io/api/core/Component) decorator.

The metadata for a component tells Angular where to get the major building blocks that it needs to create and present the component and its view. In particular, it associates a template with the component, either directly with inline code, or by reference. Together, the component and its template describe a view.

In addition to containing or pointing to the template, the @[Component](https://angular.io/api/core/Component) metadata configures, for example, how the component can be referenced in HTML and what services it requires.

Here's an example of basic metadata for HeroListComponent.

src/app/hero-list.component.ts (metadata)

@[Component](https://angular.io/api/core/Component)({

selector: 'app-hero-list',

templateUrl: './hero-list.component.html',

providers: [ HeroService ]

})

export class HeroListComponent implements [OnInit](https://angular.io/api/core/OnInit) {

/\* . . . \*/

}

This example shows some of the most useful @[Component](https://angular.io/api/core/Component) configuration options:

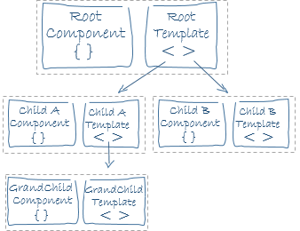
* selector: A CSS selector that tells Angular to create and insert an instance of this component wherever it finds the corresponding tag in template HTML. For example, if an app's HTML contains <app-hero-list></app-hero-list>, then Angular inserts an instance of the HeroListComponent view between those tags.
* templateUrl: The module-relative address of this component's HTML template. Alternatively, you can provide the HTML template inline, as the value of the template property. This template defines the component's host view.
* providers: An array of [providers](https://angular.io/guide/glossary#provider) for services that the component requires. In the example, this tells Angular how to provide the HeroService instance that the component's constructor uses to get the list of heroes to display.

## Templates and views



You define a component's view with its companion template. A template is a form of HTML that tells Angular how to render the component.

Views are typically arranged hierarchically, allowing you to modify or show and hide entire UI sections or pages as a unit. The template immediately associated with a component defines that component's host view. The component can also define a view hierarchy, which contains embedded views, hosted by other components.



A view hierarchy can include views from components in the same NgModule, but it also can (and often does) include views from components that are defined in different NgModules.

## Template syntax

A template looks like regular HTML, except that it also contains Angular [template syntax](https://angular.io/guide/template-syntax), which alters the HTML based on your app's logic and the state of app and DOM data. Your template can use data binding to coordinate the app and DOM data, pipes to transform data before it is displayed, and directives to apply app logic to what gets displayed.

For example, here is a template for the Tutorial's HeroListComponent.

src/app/hero-list.component.html

<h2>Hero [List](https://angular.io/api/common/NumberSymbol#List)</h2>

<p><i>Pick [a](https://angular.io/api/router/RouterLinkWithHref) hero from the list</i></p>

<ul>

<li \*[ngFor](https://angular.io/api/common/NgForOf)="let hero of heroes" (click)="selectHero(hero)">

{{hero.name}}

</li>

</ul>

<app-hero-detail \*[ngIf](https://angular.io/api/common/NgIf)="selectedHero" [hero]="selectedHero"></app-hero-detail>

This template uses typical HTML elements like <h2> and <p>, and also includes Angular template-syntax elements, \*[ngFor](https://angular.io/api/common/NgForOf), {{hero.name}}, (click), [hero], and <app-hero-detail>. The template-syntax elements tell Angular how to render the HTML to the screen, using program logic and data.

* The \*[ngFor](https://angular.io/api/common/NgForOf) directive tells Angular to iterate over a list.
* {{hero.name}}, (click), and [hero] bind program data to and from the DOM, responding to user input. See more about [data binding](https://angular.io/guide/architecture-components#data-binding) below.
* The <app-hero-detail> tag in the example is an element that represents a new component, HeroDetailComponent. HeroDetailComponent (code not shown) defines the hero-detail child view of HeroListComponent. Notice how custom components like this mix seamlessly with native HTML in the same layouts.

### Data binding

Without a framework, you would be responsible for pushing data values into the HTML controls and turning user responses into actions and value updates. Writing such push and pull logic by hand is tedious, error-prone, and a nightmare to read, as any experienced front-end JavaScript programmer can attest.

Angular supports two-way data binding, a mechanism for coordinating the parts of a template with the parts of a component. Add binding markup to the template HTML to tell Angular how to connect both sides.

The following diagram shows the four forms of data binding markup. Each form has a direction: to the DOM, from the DOM, or both.



This example from the HeroListComponent template uses three of these forms.

src/app/hero-list.component.html (binding)

<li>{{hero.name}}</li>

<app-hero-detail [hero]="selectedHero"></app-hero-detail>

<li (click)="selectHero(hero)"></li>

* The {{hero.name}} [interpolation](https://angular.io/guide/displaying-data#interpolation) displays the component's hero.name property value within the <li> element.
* The [hero] [property binding](https://angular.io/guide/template-syntax#property-binding) passes the value of selectedHero from the parent HeroListComponent to the hero property of the child HeroDetailComponent.
* The (click) [event binding](https://angular.io/guide/user-input#binding-to-user-input-events) calls the component's selectHero method when the user clicks a hero's name.

Two-way data binding (used mainly in [template-driven forms](https://angular.io/guide/forms)) combines property and event binding in a single notation. Here's an example from the HeroDetailComponent template that uses two-way data binding with the [ngModel](https://angular.io/api/forms/NgModel) directive.

src/app/hero-detail.component.html (ngModel)

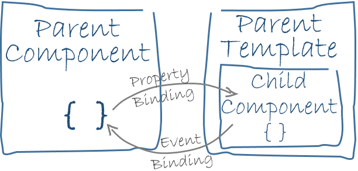
<input [([ngModel](https://angular.io/api/forms/NgModel))]="hero.name">

In two-way binding, a data property value flows to the input box from the component as with property binding. The user's changes also flow back to the component, resetting the property to the latest value, as with event binding.

Angular processes all data bindings once for each JavaScript event cycle, from the root of the application component tree through all child components.



Data binding plays an important role in communication between a template and its component, and is also important for communication between parent and child components.



### Pipes

Angular pipes let you declare display-value transformations in your template HTML. A class with the @[Pipe](https://angular.io/api/core/Pipe) decorator defines a function that transforms input values to output values for display in a view.

Angular defines various pipes, such as the [date](https://angular.io/api/common/DatePipe) pipe and [currency](https://angular.io/api/common/CurrencyPipe) pipe; for a complete list, see the [Pipes API list](https://angular.io/api?type=pipe). You can also define new pipes.

To specify a value transformation in an HTML template, use the [pipe operator (|)](https://angular.io/guide/template-syntax#pipe).

{{interpolated\_value | pipe\_name}}

You can chain pipes, sending the output of one pipe function to be transformed by another pipe function. A pipe can also take arguments that control how it performs its transformation. For example, you can pass the desired format to the date pipe.

<!-- Default format: output 'Jun 15, 2015'-->

<p>Today is {{today | date}}</p>

<!-- fullDate format: output '[Monday](https://angular.io/api/common/WeekDay#Monday), June 15, 2015'-->

<p>The date is {{today | date:'fullDate'}}</p>

<!-- shortTime format: output '9:43 AM'-->

<p>The [time](https://angular.io/) is {{today | date:'shortTime'}}</p>

### Directives



Angular templates are dynamic. When Angular renders them, it transforms the DOM according to the instructions given by directives. A directive is a class with a @[Directive](https://angular.io/api/core/Directive)() decorator.

A component is technically a directive. However, components are so distinctive and central to Angular applications that Angular defines the @[Component](https://angular.io/api/core/Component)() decorator, which extends the @[Directive](https://angular.io/api/core/Directive)() decorator with template-oriented features.

In addition to components, there are two other kinds of directives: structural and attribute. Angular defines a number of directives of both kinds, and you can define your own using the @[Directive](https://angular.io/api/core/Directive)() decorator.

Just as for components, the metadata for a directive associates the decorated class with a selector element that you use to insert it into HTML. In templates, directives typically appear within an element tag as attributes, either by name or as the target of an assignment or a binding.

#### Structural directives

Structural directives alter layout by adding, removing, and replacing elements in the DOM. The example template uses two built-in structural directives to add application logic to how the view is rendered.

src/app/hero-list.component.html (structural)

<li \*[ngFor](https://angular.io/api/common/NgForOf)="let hero of heroes"></li>

<app-hero-detail \*[ngIf](https://angular.io/api/common/NgIf)="selectedHero"></app-hero-detail>

* [\*ngFor](https://angular.io/guide/displaying-data#ngFor) is an iterative; it tells Angular to stamp out one <li> per hero in the heroes list.
* [\*ngIf](https://angular.io/guide/displaying-data#ngIf) is a conditional; it includes the HeroDetail component only if a selected hero exists.

#### Attribute directives

Attribute directives alter the appearance or behavior of an existing element. In templates they look like regular HTML attributes, hence the name.

The [ngModel](https://angular.io/api/forms/NgModel) directive, which implements two-way data binding, is an example of an attribute directive. [ngModel](https://angular.io/api/forms/NgModel) modifies the behavior of an existing element (typically <input>) by setting its display value property and responding to change events.

src/app/hero-detail.component.html (ngModel)

<input [([ngModel](https://angular.io/api/forms/NgModel))]="hero.name">

Angular has more pre-defined directives that either alter the layout structure (for example, [ngSwitch](https://angular.io/guide/template-syntax" \l "ngSwitch)) or modify aspects of DOM elements and components (for example, [ngStyle](https://angular.io/guide/template-syntax" \l "ngStyle) and [ngClass](https://angular.io/guide/template-syntax" \l "ngClass)).

**Setting up the Local Environment and Workspace**

This guide explains how to set up your environment for Angular development using the [Angular CLI tool](https://angular.io/cli). It includes information about prerequisites, installing the CLI, creating an initial workspace and starter app, and running that app locally to verify your setup.

## Prerequisites

Before you begin, make sure your development environment includes Node.js® and an npm package manager.

### Node.js

Angular requires Node.js version 10.9.0 or later.

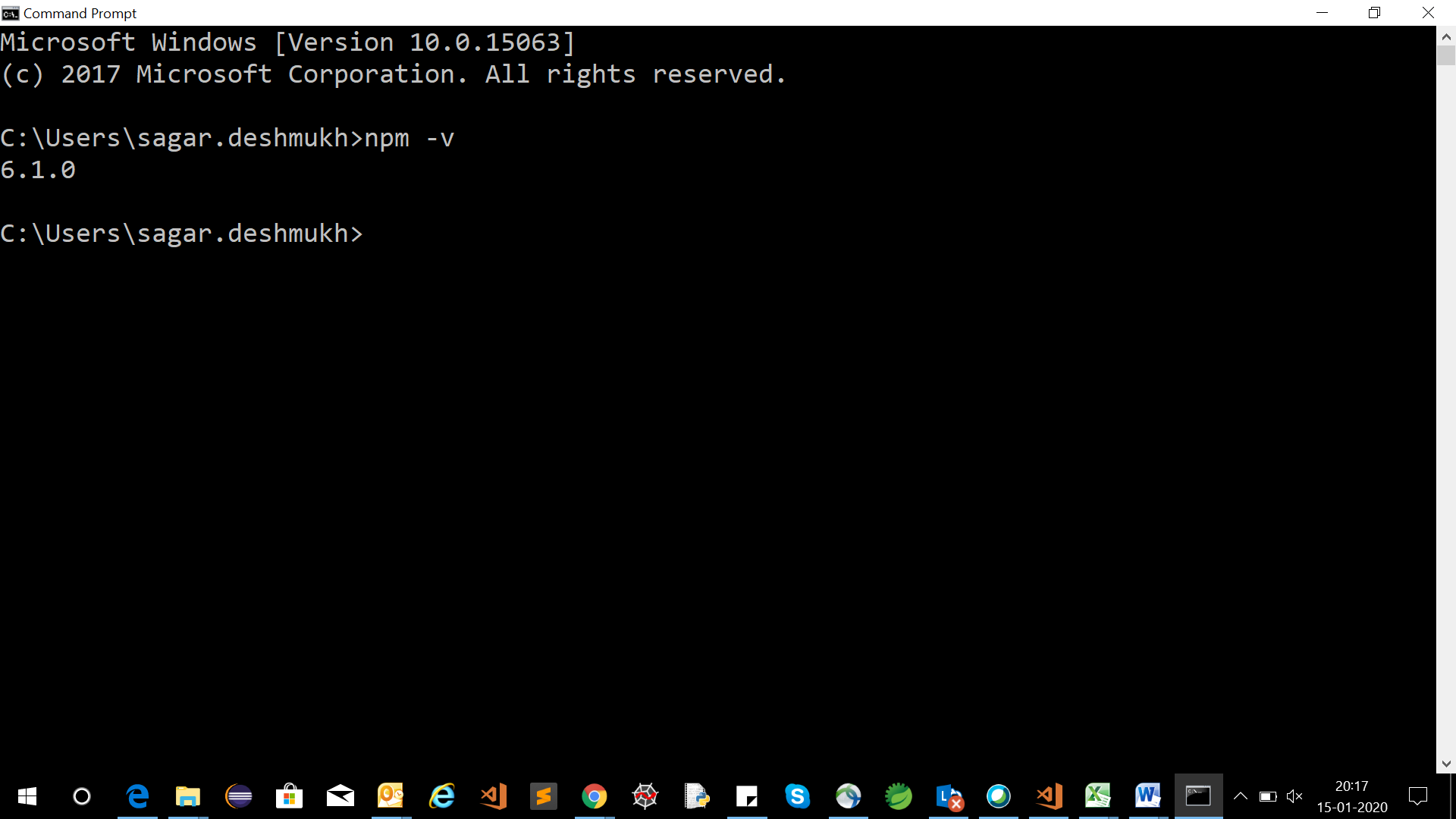
* To check your version, run node -v in a terminal/console window.
* To get Node.js, go to [nodejs.org](https://nodejs.org/).

### npm package manager

Angular, the Angular CLI, and Angular apps depend on features and functionality provided by libraries that are available as [npm packages](https://docs.npmjs.com/getting-started/what-is-npm). To download and install npm packages, you must have an npm package manager.

This setup guide uses the [npm client](https://docs.npmjs.com/cli/install) command line interface, which is installed with Node.js by default.

To check that you have the npm client installed, run npm -v in a terminal/console window.



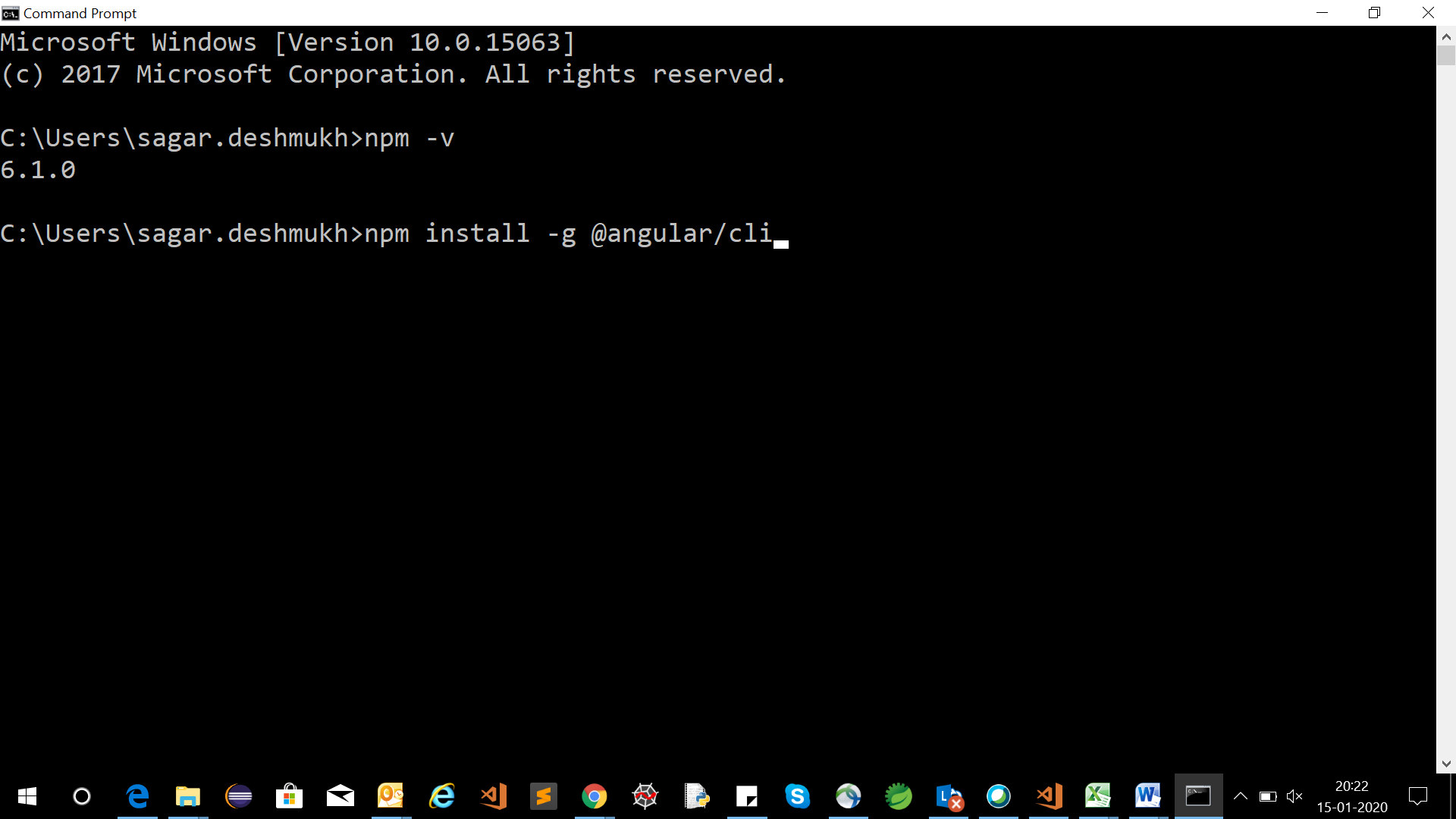
## Step 1: Install the Angular CLI

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

Install the Angular CLI globally.

To install the CLI using npm, open a terminal/console window and enter the following command:

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



## Step 2: Create a workspace and initial application

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

To create a new workspace and initial starter app:

1. Run the CLI command ng new and provide the name **MyFirstAppDemo**, as shown here:

**D:\angulrworkspace> ng new MyFirstAppDemo**

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

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

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

## Step 3: Run the application

The Angular CLI includes a server, so that you can easily build and serve your app locally.

1. Go to the workspace folder (**MyFirstAppDemo**).
2. Launch the server by using the CLI command ng serve, with the --open option.

**D:\angulrworkspace> cd MyFirstAppDemo**

**D:\angulrworkspace\MyFirstAppDemo> ng serve --open**

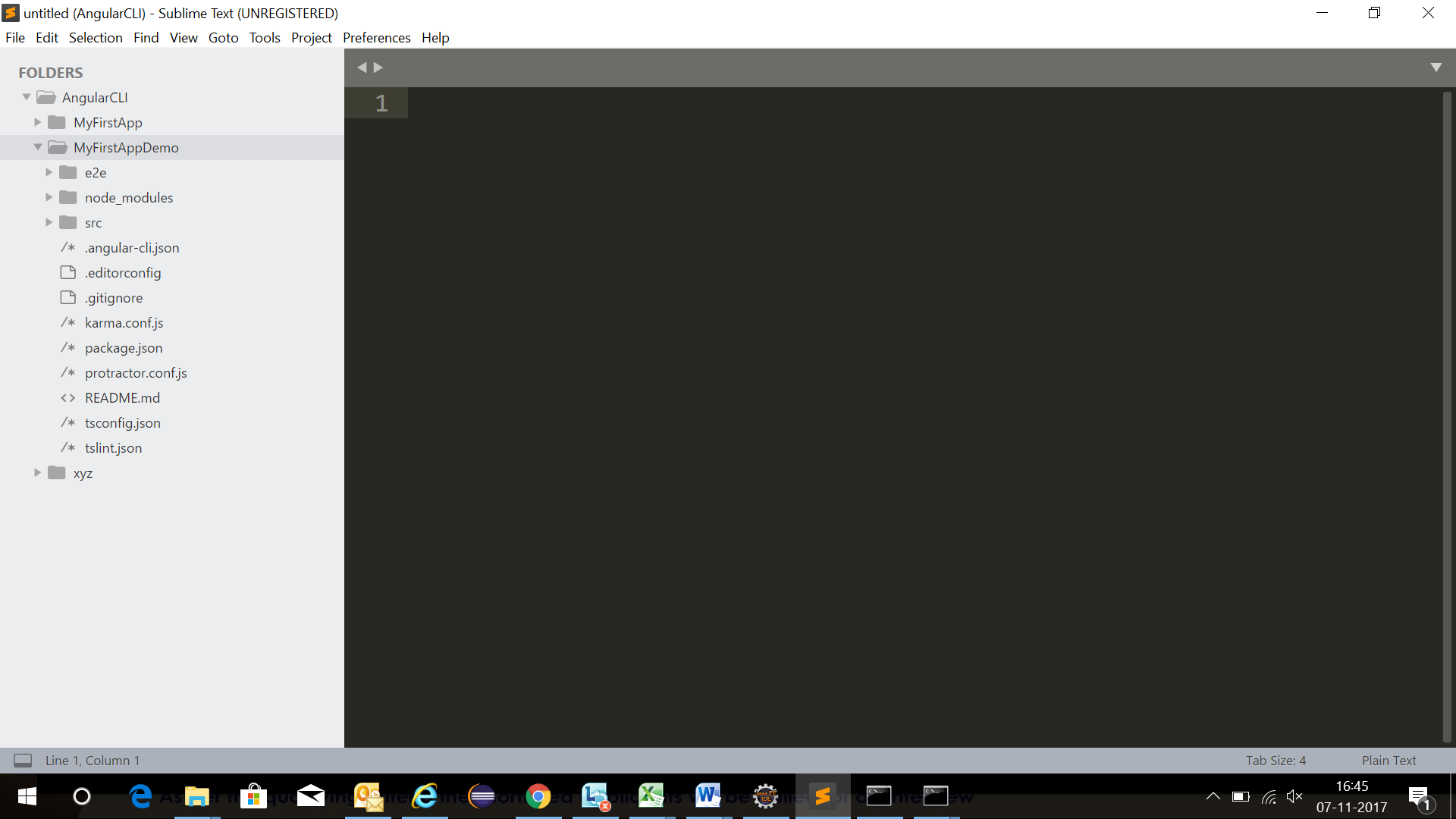
The ng serve command launches the server, watches your files, and rebuilds the app as you make changes to those files.

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

Your app greets you with a message:

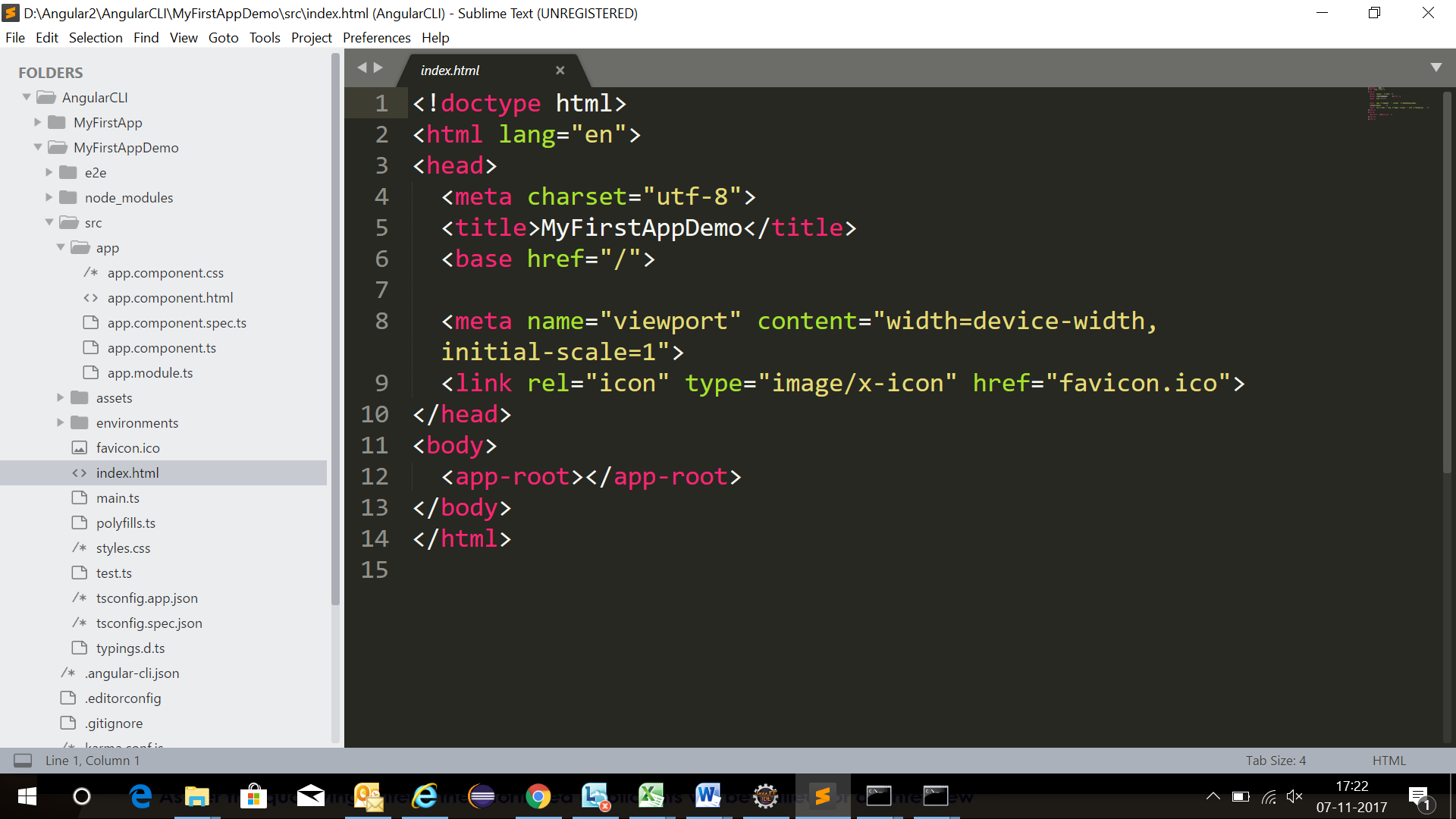
**Welcome to MyFirstAppDemo!**

**To Check Project Structure- Open Project using Visual Studio Code**



**Project Structure**

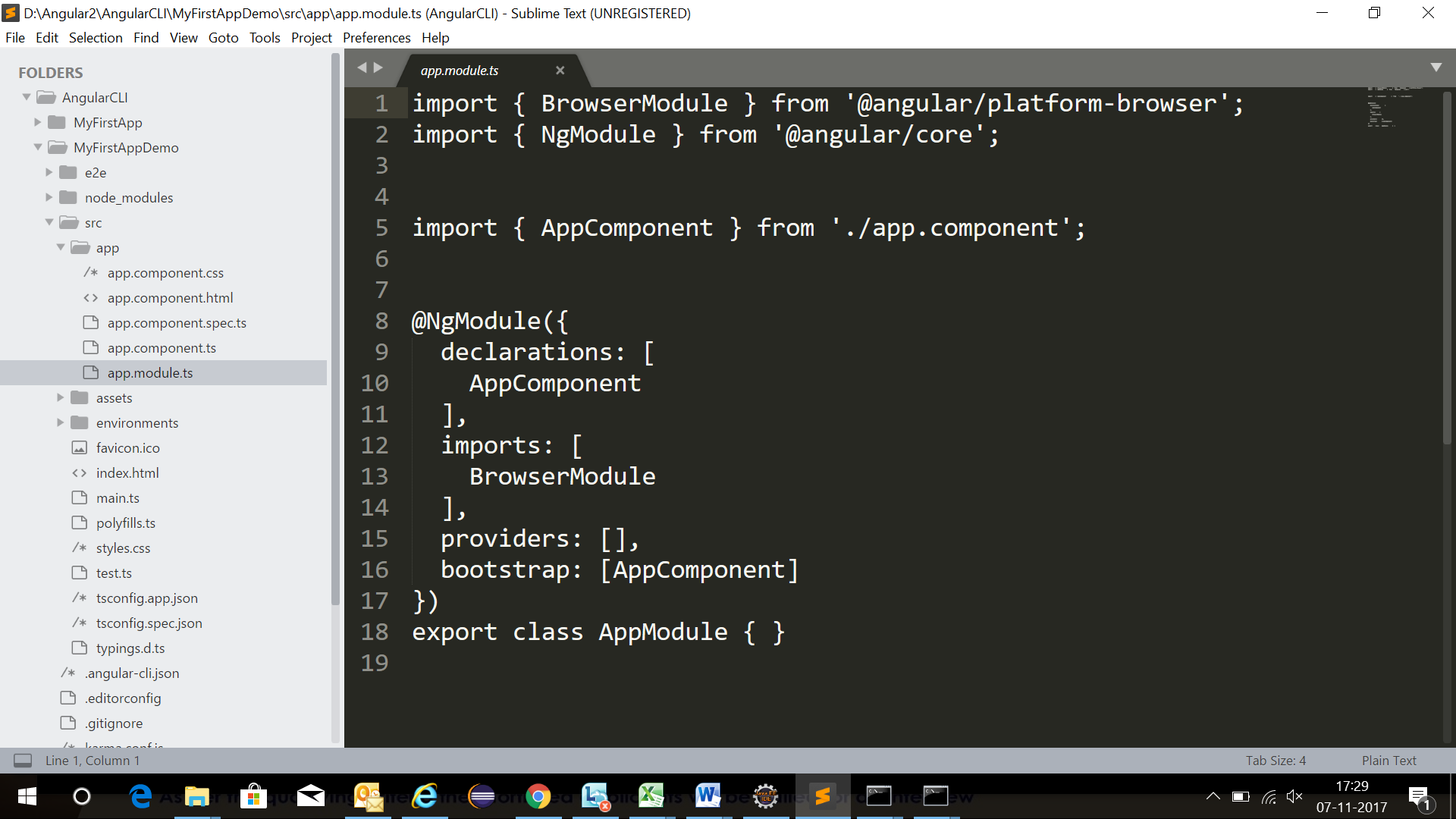
**Index.html**



**Component.ts**



**module.ts**

Is a container which contain declaration of Component , Module imports, service Providers ,bootstrap component.

# Workspace and project file structure

You develop applications in the context of an Angular [workspace](https://angular.io/guide/glossary#workspace). A workspace contains the files for one or more [projects](https://angular.io/guide/glossary#project). A project is the set of files that comprise a standalone application or a shareable library.

The Angular CLI ng new command creates a workspace.

ng new <my-project>

When you run this command, the CLI installs the necessary Angular npm packages and other dependencies in a new workspace, with a root-level application named my-project. The workspace root folder contains various support and configuration files, and a README file with generated descriptive text that you can customize.

By default, ng new creates an initial skeleton application at the root level of the workspace, along with its end-to-end tests. The skeleton is for a simple Welcome application that is ready to run and easy to modify. The root-level application has the same name as the workspace, and the source files reside in the src/ subfolder of the workspace.

This default behavior is suitable for a typical "multi-repo" development style where each application resides in its own workspace. Beginners and intermediate users are encouraged to use ng new to create a separate workspace for each application.

## Workspace configuration files

All projects within a workspace share a [CLI configuration context](https://angular.io/guide/workspace-config). The top level of the workspace contains workspace-wide configuration files, configuration files for the root-level application, and subfolders for the root-level application source and test files.

| WORKSPACE CONFIG FILES | PURPOSE |
| --- | --- |
| .editorconfig | Configuration for code editors. See [EditorConfig](https://editorconfig.org/). |
| .gitignore | Specifies intentionally untracked files that [Git](https://git-scm.com/) should ignore. |
| README.md | Introductory documentation for the root app. |
| angular.json | CLI configuration defaults for all projects in the workspace, including configuration options for build, serve, and test tools that the CLI uses, such as [TSLint](https://palantir.github.io/tslint/), [Karma](https://karma-runner.github.io/), and [Protractor](http://www.protractortest.org/). For details, see [Angular Workspace Configuration](https://angular.io/guide/workspace-config). |
| package.json | Configures [npm package dependencies](https://angular.io/guide/npm-packages) that are available to all projects in the workspace. See [npm documentation](https://docs.npmjs.com/files/package.json) for the specific format and contents of this file. |
| package-lock.json | Provides version information for all packages installed into node\_modules by the npm client. See [npm documentation](https://docs.npmjs.com/files/package-lock.json) for details. If you use the yarn client, this file will be [yarn.lock](https://yarnpkg.com/lang/en/docs/yarn-lock/) instead. |
| src/ | Source files for the root-level application project. |
| node\_modules/ | Provides [npm packages](https://angular.io/guide/npm-packages) to the entire workspace. Workspace-wide node\_modules dependencies are visible to all projects. |
| tsconfig.json | Default [TypeScript](https://www.typescriptlang.org/) configuration for projects in the workspace. |
| tslint.json | Default [TSLint](https://palantir.github.io/tslint/) configuration for projects in the workspace. |

## Application project files

By default, the CLI command ng new my-app creates a workspace folder named "my-app" and generates a new application skeleton in a src/ folder at the top level of the workspace. A newly generated application contains source files for a root module, with a root component and template.

When the workspace file structure is in place, you can use the ng generate command on the command line to add functionality and data to the application. This initial root-level application is the default app for CLI commands (unless you change the default after creating [additional apps](https://angular.io/guide/file-structure#multiple-projects)).

For a single-application workspace, the src/ subfolder of the workspace contains the source files (application logic, data, and assets) for the root application. For a multi-project workspace, additional projects in the projects/ folder contain a project-name/src/ subfolder with the same structure.

### Application source files

Files at the top level of src/ support testing and running your application. Subfolders contain the application source and application-specific configuration.

| APP SUPPORT FILES | PURPOSE |
| --- | --- |
| app/ | Contains the component files in which your application logic and data are defined. See details [below](https://angular.io/guide/file-structure#app-src). |
| assets/ | Contains image and other asset files to be copied as-is when you build your application. |
| environments/ | Contains build configuration options for particular target environments. By default there is an unnamed standard development environment and a production ("prod") environment. You can define additional target environment configurations. |
| favicon.ico | An icon to use for this application in the bookmark bar. |
| index.html | The main HTML page that is served when someone visits your site. The CLI automatically adds all JavaScript and CSS files when building your app, so you typically don't need to add any <script> or<link> tags here manually. |
| main.ts | The main entry point for your application. Compiles the application with the [JIT compiler](https://angular.io/guide/glossary#jit) and bootstraps the application's root module (AppModule) to run in the browser. You can also use the [AOT compiler](https://angular.io/guide/aot-compiler) without changing any code by appending the --aot flag to the CLI build and serve commands. |
| polyfills.ts | Provides polyfill scripts for browser support. |
| styles.sass | Lists CSS files that supply styles for a project. The extension reflects the style preprocessor you have configured for the project. |
| test.ts | The main entry point for your unit tests, with some Angular-specific configuration. You don't typically need to edit this file. |

**Inside the src/ folder, the app/ folder contains your project's logic and data. Angular components, templates, and styles go here**.

| SRC/APP/ FILES | PURPOSE |
| --- | --- |
| app/app.component.ts | Defines the logic for the app's root component, named AppComponent. The view associated with this root component becomes the root of the [view hierarchy](https://angular.io/guide/glossary#view-hierarchy) as you add components and services to your application. |
| app/app.component.html | Defines the HTML template associated with the root AppComponent. |
| app/app.component.css | Defines the base CSS stylesheet for the root AppComponent. |
| app/app.component.spec.ts | Defines a unit test for the root AppComponent. |
| app/app.module.ts | Defines the root module, named AppModule, that tells Angular how to assemble the application. Initially declares only the AppComponent. As you add more components to the app, they must be declared here. |

Lifecycle Hooks

A component has a lifecycle managed by Angular.

Angular creates and renders components along with their children, checks when their data-bound properties change, and destroys them before removing them from the DOM.

Angular offers lifecycle hooks that provide visibility into these key life moments and the ability to act when they occur.

A directive has the same set of lifecycle hooks.

Component lifecycle hooks overview

Directive and component instances have a lifecycle as Angular creates, updates, and destroys them. Developers can tap into key moments in that lifecycle by implementing one or more of the *lifecycle hook* interfaces in the Angular core library.

Each interface has a single hook method whose name is the interface name prefixed with ng. For example, the [OnInit](https://angular.io/api/core/OnInit) interface has a hook method named ngOnInit() that Angular calls shortly after creating the component:

peek-a-boo.component.ts (excerpt)

export class PeekABoo implements [OnInit](https://angular.io/api/core/OnInit) {

constructor(private logger: LoggerService) { }

// implement [OnInit](https://angular.io/api/core/OnInit)'s `ngOnInit` method

ngOnInit() { this.logIt(`[OnInit](https://angular.io/api/core/OnInit)`); }

logIt(msg: string) {

this.logger.log(`#${nextId++} ${msg}`);

}

}

No directive or component will implement all of the lifecycle hooks. Angular only calls a directive/component hook method *if it is defined*.

Lifecycle sequence

*After* creating a component/directive by calling its constructor, Angular calls the lifecycle hook methods in the following sequence at specific moments:

|  |  |
| --- | --- |
| **Hook** | **Purpose and Timing** |
| ngOnChanges() | Respond when Angular (re)sets data-bound input properties. The method receives a [SimpleChanges](https://angular.io/api/core/SimpleChanges) object of current and previous property values.  Called before ngOnInit() and whenever one or more data-bound input properties change. |
| ngOnInit() | Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties.  Called *once*, after the *first* ngOnChanges(). |
| ngDoCheck() | Detect and act upon changes that Angular can't or won't detect on its own.  Called during every change detection run, immediately after ngOnChanges() and ngOnInit(). |
| [ngAfterContentInit()](https://angular.io/api/router/RouterLinkActive#ngAfterContentInit) | Respond after Angular projects external content into the component's view / the view that a directive is in.  Called *once* after the first ngDoCheck(). |
| ngAfterContentChecked() | Respond after Angular checks the content projected into the directive/component.  Called after the [ngAfterContentInit()](https://angular.io/api/router/RouterLinkActive" \l "ngAfterContentInit) and every subsequent ngDoCheck(). |
| [ngAfterViewInit()](https://angular.io/api/forms/NgForm#ngAfterViewInit) | Respond after Angular initializes the component's views and child views / the view that a directive is in.  Called *once* after the first ngAfterContentChecked(). |
| ngAfterViewChecked() | Respond after Angular checks the component's views and child views / the view that a directive is in.  Called after the [ngAfterViewInit()](https://angular.io/api/forms/NgForm" \l "ngAfterViewInit) and every subsequent ngAfterContentChecked(). |
| ngOnDestroy() | Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks.  Called *just before* Angular destroys the directive/component. |