**Angular Architecture Overview**

Angular is a framework for building client applications in HTML and either JavaScript or a language like Type Script that compiles to JavaScript.

The framework consists of several libraries, some of them core and some optional.

You write Angular applications by

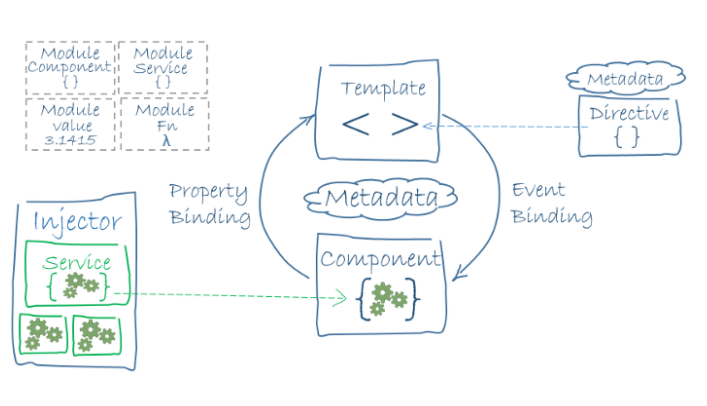
composing HTML *templates*,

writing *component* classes to manage those templates,

adding application logic in *services*, and

boxing components and services in *modules*

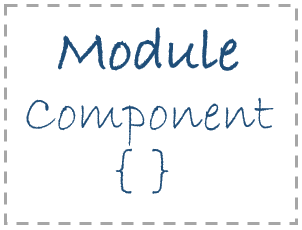
***Then you launch the app by bootstrapping the root module. Angular presenting your application content in a browser and responding to user interactions according to the instructions you’ve provided.***



**The Eight main building blocks of an Angular application:**

* [Modules](https://v2.angular.io/docs/ts/latest/guide/architecture.html#modules)
* [Components](https://v2.angular.io/docs/ts/latest/guide/architecture.html#components)
* [Templates](https://v2.angular.io/docs/ts/latest/guide/architecture.html#templates)
* [Metadata](https://v2.angular.io/docs/ts/latest/guide/architecture.html#metadata)
* [Data binding](https://v2.angular.io/docs/ts/latest/guide/architecture.html#data-binding)
* [Directives](https://v2.angular.io/docs/ts/latest/guide/architecture.html#directives)
* [Services](https://v2.angular.io/docs/ts/latest/guide/architecture.html#services)
* [Dependency injection](https://v2.angular.io/docs/ts/latest/guide/architecture.html#dependency-injection)

**Modules**



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.

If we want to use another custom Angular module, then we need to register that module inside the **app.module.ts** file. Organizing your code into distinct functional modules helps in managing the development of complex applications, and in designing for re usability.

**Angular libraries**

Angular ships as a collection of JavaScript modules. You can think of them as library modules.

Each Angular library name begins with the @angular prefix.

You install them with the **npm** package manager and import parts of them with JavaScript importstatements.

For example, import Angular’s Component decorator from the @angular/core library like this:

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

You also import Angular *modules* from Angular *libraries* using JavaScript import statements:

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

In the example of the simple root module above, the application module needs material from within that BrowserModule. To access that material, add it to the @NgModule metadata imports like this.

imports:      [ BrowserModule ],

In this way you’re using both the Angular and JavaScript module systems *together*.

**Components**

Every Angular project has at least one component, the *root component and* root component connects the component hierarchy with a page document object model (DOM). Each component defines the class that contains application data and logic, and it is associated with the HTML template that defines the view to be displayed in a target app.A *component* controls a patch of screen called a *view*.

The @Component decorator identifies the class immediately below it as the component and provides the template and related component-specific metadata.

*// app.component.ts*

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

**Templates**



The angular template combines the 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.

<div style="text-align:center">

  <h1>

    {{2 | power: 5}}

  </h1>

</div>

In the above HTML file, we have used a template. We have also used the pipe inside the template to transform the values to the desired output.

**Metadata**

https://miro.medium.com/max/170/0*pzRv5rN0v_LdibZN.png

Metadata tells Angular how to process a class.It is used to decorate the class so that it can configure the expected behavior of a class. Decorators are the core concept when developing with Angular (versions 2 and above). The user can use metadata to a class to tell Angular app that AppComponent is the component. Metadata can be attached to the TypeScript using the decorator.

*// app.component.ts*

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

Here is the @Component decorator, which identifies the class immediately below it as a component class.

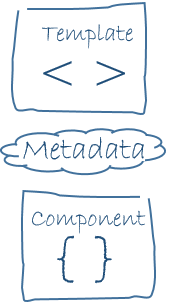
The @Component decorator takes a required configuration object with the information Angular needs to create and present the component and its view.

Here are a few of the most useful @Component configuration options:

selector: CSS selector that tells Angular to create and insert an instance of this component where it finds a <app-root>tag which is Parent Component.

templateUrl: module-relative address of this component's HTML template.

providers: array of **dependency injection providers** for services that the component requires.



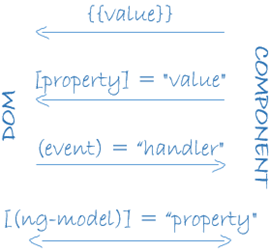
The metadata in the @Component tells Angular where to get the major building blocks you specify for the component.

The template, metadata, and component together describe a view.

Apply other metadata decorators in a similar fashion to guide Angular behavior. @Injectable, @Input, and @Output are a few of the more popular decorators.

**Data binding**

Data binding plays an important role in communication between a template and its component.



Data binding is also important for communication between parent and child components.Angular allows defining communication between a component and the DOM, making it very easy to define interactive applications without worrying about pulling and pushing the data.

**From the Component to the DOM**

Interpolation: {{ value }}: Interpolation adds the value of the property from the component.

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

<p>College: {{ student.college }}</p>

**Event binding: (Event)=”myFunction($event)”**

With Event binding, a value is passed from a child Component to Parent Component,which can often be a simple html attribute.

<input type="text" (myEvent)="onClick($event)" />

**Property binding: [property]=”value”**

With property binding, a value is passed from a component to a specified property, which can often be a simple html attribute.

<input type="text" [value]="student.name" />

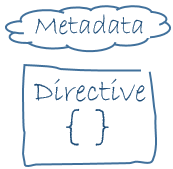
<input type="text" [value]="student.college" />

**Two-Way binding: [(ngModel)]=”property”**

It is an important fourth form that combines property and event binding in a single notation, using the ngModel directive.

<input [(ngModel)]="hero.name">

**Directives**



An Angular component isn’t more than a directive with the template. When we say that components are the building blocks of Angular applications, we are saying that directives are the building blocks of Angular projects. Let us use built-in Angular directive like ngClass, which is a better example of the existing Angular attribute directive.

<p [ngClass]="{'coffee'=true, 'red'=false}">

    Angular 7 Directives Example

</p>

<style>

    .coffee{color: coffee}

    .red{color: red}

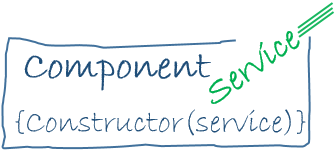
</style>

Here, based on the [ngClass] directive’s value, the text has color. In our example, the text will be coffee because it is true.

**Services**

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. The @Injectable decorator immediately precedes the service class definition. The decorator provides the metadata that allows your service to be *injected* into client components as a dependency. Angular distinguishes components from services to increase modularity and reusability. By separating a component’s view-related functionality from other kinds of processing, you can make your component classes lean and efficient.

**Dependency injection**

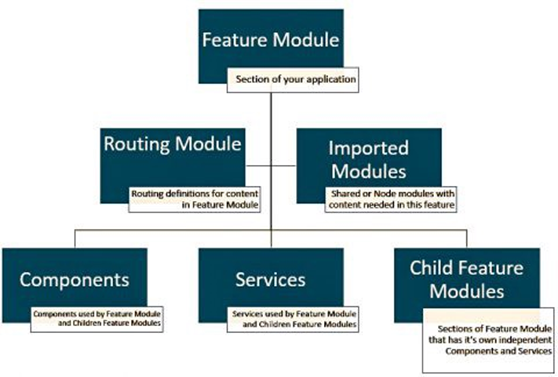


Dependency injection (DI) lets you keep your component classes lean and efficient. DI does not fetch data from a server, validate the user input, or log directly to the console instead they delegate such tasks to the services. DI is wired into a Angular framework and used everywhere to provide new components with the services or other things they need. Components consume services; that is, you can *inject* a service into a component, giving the component access to that service class.

[**Types of NgModule**](https://medium.com/@bhavikagarg8/types-of-ngmodule-73fa62f038f3?source=follow_footer-----1e7cc7483a0----0-------------------------------)

**Angular Module Structure**

When setting up modules in an Angular application, I like to keep this hierarchy picture in your mind.



NgModule Structure

**There are five types of NgModule –**

1. Features Module

2. Routing Module

3. Service Module

4. Widget Module

5. Shared Module

* ***Features Module*** *— The feature modules are* [*NgModules*](https://www.code-sample.com/2018/04/modules-ngmodule-angular-4-5-6.html) *for the purpose of organizing an application code.*
* A feature module is a module in which all of the content is going to be encapsulated inside of a single area.
* A feature would be what I mean by “section”, it usually has a root component that it exports and is used in a parent module. All the rest of the pieces of that feature will be put inside that root component like Import and Export.
* ***Routing Module*** *— The Routing is used to manage routes and also enables navigation from one view to another view as users perform application tasks.*

For example, FooModule in foo.module.ts has a routing module named FooRoutingModule in foo-routing.module.ts.

If the companion module is the root AppModule, the AppRoutingModule adds router configuration to its imports with RouterModule.forRoot(routes). All other routing modules are children that import RouterModule.forChild(routes).

* ***Service Module*** *— The modules that only contain services and providers. It provides utility services such as data access and messaging Ideally they consist entirely of providers and have no declarations.*

The root AppModule is the only module that should import service modules. TheHttpClientModule and Singleton Service are good example of a service.

* ***Widget Module*** *— The third party UI component libraries are widget modules. A widget module should consist entirely of declarations, most of them exported.*

A widget module should rarely have providers.

Import widget modules in any module whose component templates need the widgets.

* ***Shared Module*** *— The shared module allows you to organize your application code. You can put your commonly used components, directives, and pipes into the one module and use whenever required to this module.*
* Use shared modules for pieces of your application that need to be used across multiple areas (features) of your application.
* If a component is going to be re-used in multiple features, declare it in a shared module.
* Services and Pipes are more commonly declared in shared modules.

[**Understanding Angular modules (NgModule) and their scopes**](https://medium.com/@bhavikagarg8/understanding-angular-modules-ngmodule-and-their-scopes-59e02da1cf8f?source=follow_footer-----1e7cc7483a0----1-------------------------------)

NgModule is the first basic structure you meet when coding an app with Angular, but it’s also the most subtle and complex, because of **different scopes**. An NgModule is defined by a class decorated with @NgModule(). The **@NgModule()** decorator is a function that takes a single metadata object, whose properties describe the module.

**Why NgModule?**

It’s done automatically with [Angular CLI](https://github.com/angular/angular-cli), but the first thing you have to do in Angular is to load a root [NgModule](https://angular.io/guide/ngmodules) :

**The purpose of a NgModule is to declare each thing you create in Angular and group them together** (like Java packages or PHP / C# namespaces).

There is two kind of main structures:

* **“declarations” is for things you’ll use in your templates: mainly components** (~ views: the classes displaying data), but also directives and pipes,
* **“providers” is for services** (~ models: the classes getting and handling data).



**NgModule and scopes/visibility**

The confusion starts with **components and services not having the same scope/visibility**:

* declarations / **components are in local scope** (private visibility),
* providers/services **are (generally) in the global scope** (public visibility).

It means the **components you declared are only usable in the current module**. If you need to use them outside, in other modules, you’ll have to export them:

On the contrary,**services you provided will generally be available/injectable anywhere in your app**, in all modules.



**When to import a NgModule?**

The difference of scope between components and services is an important point to know, but for now it’s still OK. Things get messy because, of course, as in any framework and app, you won’t just have one module, but many of them. Angular itself is subdivided in different modules (core, common, http and so on).

So another main thing you do in an Angular module is to **import other NgModules** you need:



Problem is: **you need to know why you import these other modules.**

* Is it to use components (or other template-related things, like directives and pipes)?
* or is is to use services?

Why? Because given the difference of scope between components and services :

* **if the module is imported for components, you’ll need to import it in each module** needing them,
* **if the module is imported for services, you’ll need to import it only once**, in the first app module.

If you fail to understand this, you’ll have errors on components not being available, because you forgot to import their module again.

Or if you import a module for services more than once, it can lead to errors in advanced scenarios like lazy-loading.

**When to import main Angular modules?**

**A good knowledge of Angular modules is then required, to know how many times you need to import them**. Here is a helpful summary.

**Modules to import each time you need them**

* **Common module** (all the basics of Angular templating: bindings, \*ngIf, \*ngFor…), except in the first app module, because it’s already part of the BrowserModule
* FormsModule / ReactiveFormsModule (For Forms and validation)
* MatXModule and other UI modules
* any other module giving you components, directives or pipes

**Modules to import only once**

* **HttpClientModule (**For HTTP Request on Server like Get,Post by using Observable**)**
* BrowserAnimationsModule or NoopAnimationsModule( Api for animations in your App.)
* any other module providing you services *only*.

That’s why with [Angular CLI](https://github.com/angular/angular-cli), CommonModule is automatically imported when you create a new module.

**Mixed NgModules**

It can get messier: how to manage modules with components and services at the same time?

You know one of them : the **RouterModule**. It gives you a component (<router-outlet>) and a directive (routerLink), but also services (ActivatedRoute to get URL params, Router to navigate…).

Fortunately, the mess is managed by the module itself. Routing files are automatically generated by [Angular CLI](https://github.com/angular/angular-cli), but you may have noticed there is a subtle difference between the routing of your first app module and the routing of submodules.

For the AppModule, it does:

For submodules, it does:

Why? Because the first time in app module, forRoot() will give the router components *and* provide the router services. But the next times in submodules, forChild will *only* give the router components (and not providing again the services, which would be bad).

**Lazy-loaded modules**

Last complication : if you lazy-load a module, which is now easy with [Angular CLI](https://github.com/angular/angular-cli).

As it will be a different bundle and module, loaded only on demand by default, **it’s not included in the global scope your app**.

**For components, it doesn’t change anything**: you need to import again the CommonModule and other modules of components, like in any submodule.

For services, there is a difference:

* **you’ll still have access to services already provided in the app** (like HttpClient and your own services),
* but **the services provided in your lazy-loaded module will only be available in this lazy-loaded module**, not everywhere in your app.

**Why?**

Now you know everything about Angular modules, you may ask : why this mess ? Well, it may be difficult for beginners, but there is a good reason:

* services are mostly just ES6 classes : they are imported/exported, so in their namespaces, so no risk of collision! And singletons are usually what you want.
* components create… components, ie. new HTML tags : if they were global, loading two libraries creating components with the same name would create conflicts.



[**AppModule: Root Module**](https://medium.com/@bhavikagarg8/appmodule-root-module-75a6d7cb083c?source=follow_footer-----1e7cc7483a0----2-------------------------------)

An Angular module class describes how the Application parts well-suited together. Every application has at least one Angular module, the *root* module that you [bootstrap](https://v2.angular.io/docs/ts/latest/guide/appmodule.html#main) to launch the application. You can call it anything you want. The conventional name is AppModule.

The [setup](https://v2.angular.io/docs/ts/latest/guide/setup.html) instructions produce a new project with the following minimal AppModule. You'll evolve this module as your application grows.

**src/app/app.module.ts**

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

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

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

@NgModule({

  imports:      [ BrowserModule ],

  declarations: [ AppComponent ],

  bootstrap:    [ AppComponent ]

})

export class AppModule { }

After the import statements, you come to a class adorned with the **@NgModule** [*decorator*](https://v2.angular.io/docs/ts/latest/guide/glossary.html#decorator).

The @NgModule decorator identifies AppModule as an Angular module class (also called an NgModule class). @NgModule takes a *metadata* object that tells Angular how to compile and launch the application .

* *How to compile a component’s template and how to create an injector at runtime. It identifies the module’s own components, directives, and pipes, making some of them public, through the exports property, so that external components can use them. @NgModule can also add service providers to the application dependency injectors.*
* ***imports*** — the BrowserModule that this and every application needs to run in a browser.
* ***declarations*** — the application’s lone component, which is also …
* ***providers* —** the service providers.
* ***bootstrap*** — the *root* component that Angular creates and inserts into the index.html host web page.

The [Angular Modules (NgModule)](https://v2.angular.io/docs/ts/latest/guide/ngmodule.html) guide dives deeply into the details of Angular modules. All you need to know at the moment is a few basics about these three properties.

**The imports array**

The module’s [imports](https://angular.io/api/core/NgModule#imports) array appears exclusively in the @[NgModule](https://angular.io/api/core/NgModule) metadata object. it tells Angular about other NgModules that this particular module needs to function properly.Many features of Angular itself are organized as Angular modules. HTTP services are in the HttpModule. The router is in the RouterModule. Eventually you may create a feature module.

Add a module to the imports array when the application requires its features.

*This* application, like most applications, executes in a browser. Every application that executes in a browser needs the BrowserModule from @angular/platform-browser. So every such application includes the BrowserModule in its *root* AppModule's imports array. Other guide and cookbook pages will tell you when you need to add additional modules to this array.

**Only NgModule classes** go in the imports array. Do not put any other kind of class in imports.

The import statements at the top of the file and the Angular module's imports array are unrelated and have completely different jobs.

The *JavaScript* import statements give you access to symbols *exported* by other files so you can reference them within *this* file. You add import statements to almost every application file. They have nothing to do with Angular and Angular knows nothing about them.

The *module’s* imports array appears *exclusively* in the @NgModule metadata object. It tells Angular about specific *other* Angular modules — all of them classes decorated with @NgModule — that the application needs to function properly.

**The declarations array**

You tell Angular which components belong to the AppModule by listing it in the module's declarations array. As you create more components, you'll add them to declarations.

You must declare *every* component in an NgModule class. If you use a component without declaring it, you'll see a clear error message in the browser console.



Console Error

You’ll learn to create two other kinds of classes — [directives](https://v2.angular.io/docs/ts/latest/guide/attribute-directives.html) and [pipes](https://v2.angular.io/docs/ts/latest/guide/pipes.html) — that you must also add to the declarations array.

**Only *declarables*** — *components*, *directives* and *pipes* — belong in the declarations array. Do not put any other kind of class in declarations; *not* NgModule classes, *not* service classes, *not* model classes.

**The providers array**

The providers array is where you list the services the app needs. When you list services here, they are available app-wide. You can scope them when using feature modules and lazy loading. For more information, see [Providers](https://angular.io/guide/providers).

**The bootstrap array**

You launch the application by [*bootstrapping*](https://v2.angular.io/docs/ts/latest/guide/appmodule.html#main) the root AppModule ,which is also referred to as an entryComponent. Other things, the *bootstrapping* process creates the component(s) listed in the bootstrap array and inserts each one into the browser DOM.

Even When you can put more than one component tree on a host web page, that’s not typical. Most applications have only one component tree and they bootstrap a single *root* component.

This one root component is usually called AppComponent and is in the root module's [bootstrap](https://angular.io/api/core/NgModule#bootstrap) array.

**Bootstrap in main.ts**

There are many ways to bootstrap an application. The variations depend upon how you want to compile the application and where you want to run it.

In the beginning, you will compile the application dynamically with the *Just-in-Time (JIT)* compiler and you’ll run it in a browser. You can learn about other options later.

The recommended place to bootstrap a JIT-compiled browser application is in a separate file in the src folder named src/main.ts

**src/main.ts**

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

import { AppModule }              from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule);

This code creates a browser platform for dynamic (JIT) compilation and bootstraps the AppModule described above.

The *bootstrapping* process sets up the execution environment, digs the *root* AppComponent out of the module's bootstrap array, creates an instance of the component and inserts it within the element tag identified by the component's selector.

The AppComponent selector — here and in most documentation samples — is my-app so Angular looks for a <my-app> tag in the index.html like this one ...

<my-app><!-- content managed by Angular --></my-app>

… and displays the AppComponent there.

This file is very stable. Once you’ve set it up, you may never change it again.

**More about Angular Modules**

Your initial app has only a single module, the *root* module. As your app grows, you’ll consider subdividing it into multiple “feature” modules, some of which can be loaded later (“lazy loaded”) if and when the user chooses to visit those features.

[**Important topics of Angular**](https://medium.com/@bhavikagarg8/important-topics-of-angular-5d51362ae78e?source=follow_footer-----1e7cc7483a0----3-------------------------------)

There are many topics which covers it:-

1. Component
2. Module
3. Data binding with multiple Component

* Parent-Child Communication
* String Interpolation {{}}
* One way Binding — Custom Binding, Property Binding
* Two Way Binding [(ngModel)]

4. Directives and Life cycle

5. Host Binding and Attribute Directives

6. View Children and Content Children

7. Services & Dependency Injection

8. Changing Pages with Routing

* Router Guard

9. Understanding Observable

* Observable with Reactive Programming

10. Handling Forms in Angular Apps

11. Using Pipes to Transform Output

12. Making HTTP Requests

13. Authentication & Route Protection in Angular Apps

14. Using Angular Modules & Optimizing Apps

15.Deploying in Angular App

16. Animation

17. Angular Universal

[**How we Setup the Angular on Machine**](https://medium.com/@bhavikagarg8/how-we-setup-the-angular-on-machine-19c2349f334e?source=follow_footer-----1e7cc7483a0----4-------------------------------)

1. **Set up the Development Environment**

You need to setup your development Environment before you can do anything .

You need to **Install Node.js and Npm** on your machine.

* *For verify you are running node version and npm version by running* ***node-v*** *and* ***npm -v.***

Then Install Angular Cli Globally By **“npm install -g @angular/cli”**

**2. Create A New Project**

Open a terminal Window.

Generate a new project and default app by running the following command: “**ng new project name”**

* *The Angular CLI installs the necessary npm packages, creates the project files, and populates the project with a simple default app. This can take some time.*
* *You can add pre-packaged functionality to a new project by using the* ***ng add*** *command.*

**3. Serve The Application**

Go to project Directory and Launch The Server

**cd project name**

**ng serve — open**

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

Using the **— open** (or just **-o**) option will automatically open your browser on [**http://localhost:4200/**](http://localhost:4200/).

Your app greets you with a message:



Default Angular App

**4. Edit Your First Angular Component**

The CLI created the first Angular component for you. This is the *root component* and it is named **app-root**. You can find it in “**./src/app/app.component.ts”.**

Open the component file and change the **title** property from 'app' to 'My First Angular App!'.

**src/app/app.component.ts**

content\_copyexport class AppComponent {

  title = 'My First Angular App!';

}

The browser reloads automatically with the revised title. That’s nice, but it could look better.

Open src/app/app.component.css and give the component some style.

**src/app/app.component.css**

content\_copyh1 {

  color: #369;

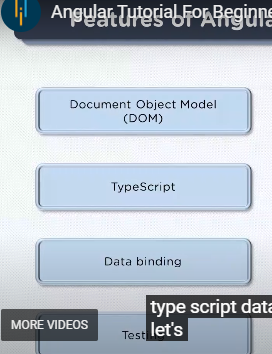
  font-family: Arial, Helvetica, sans-serif;

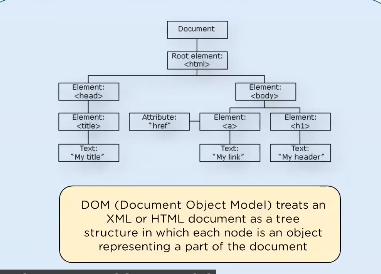
  font-size: 250%;

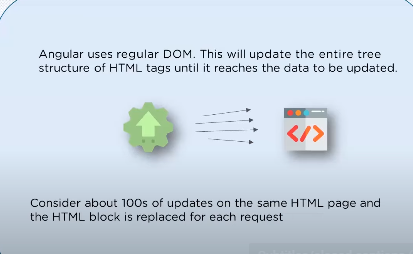
}

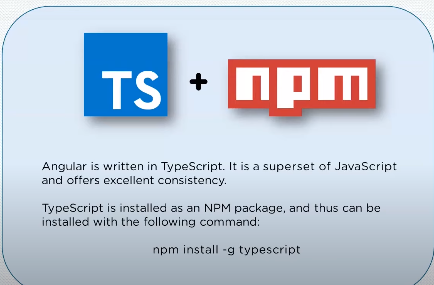


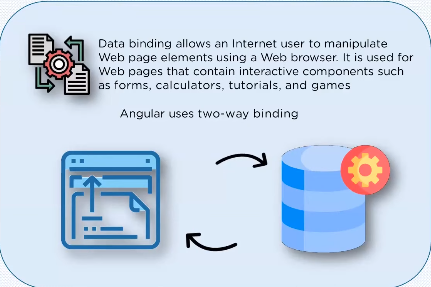
Change the title of Default App on browser by Angular

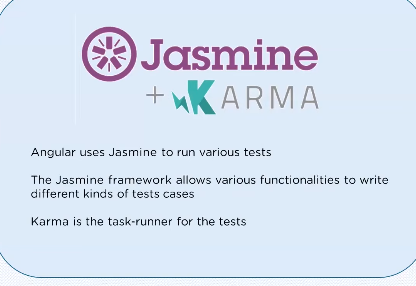


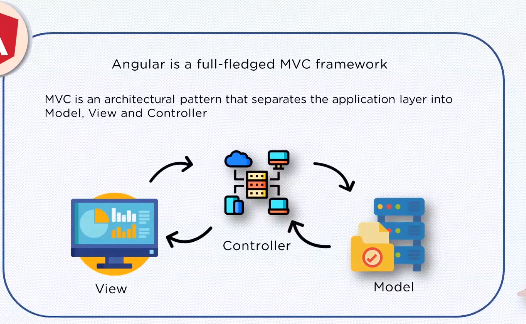




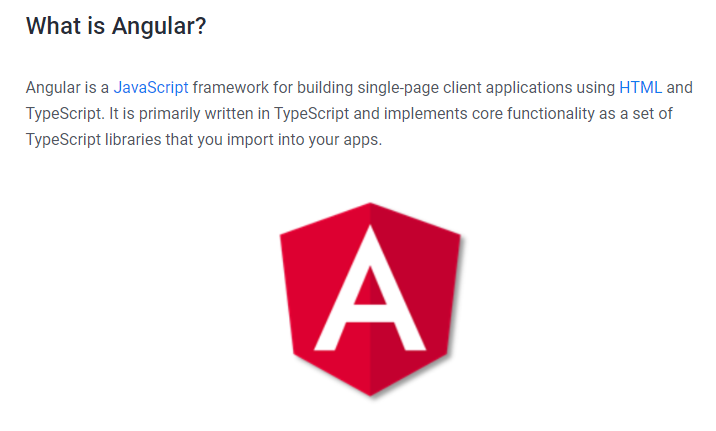


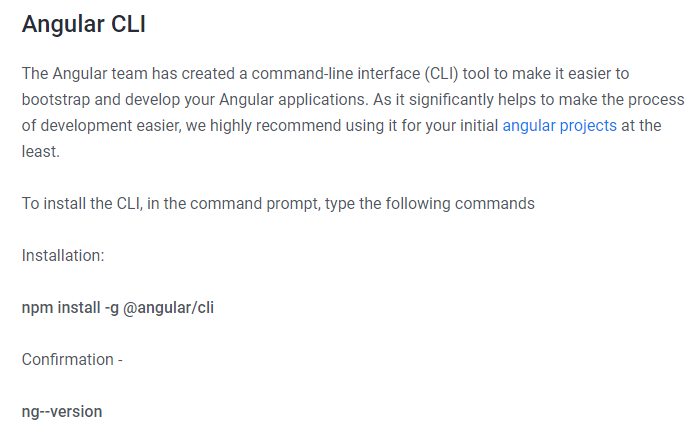


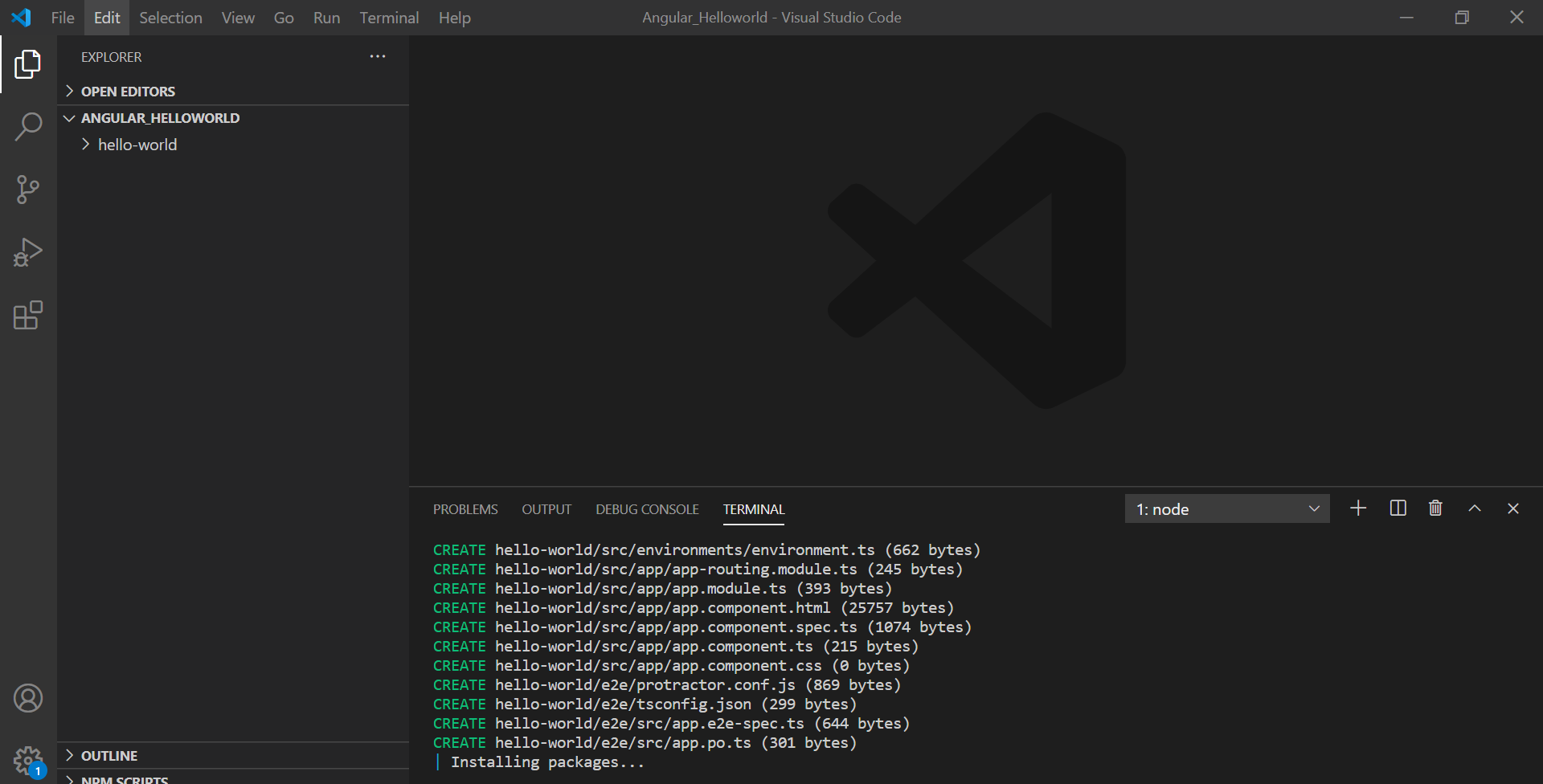


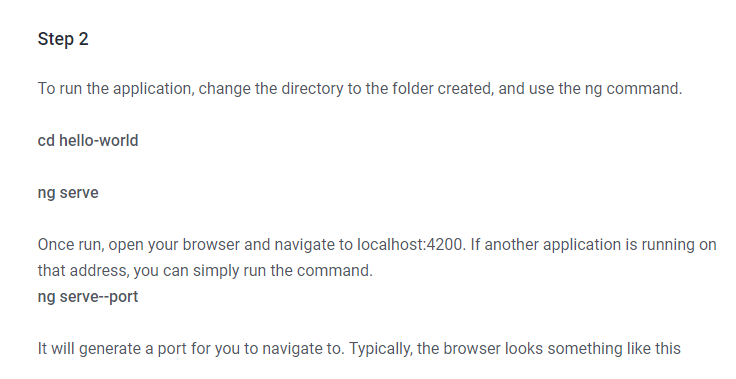


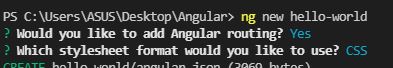












https://paper-attachments.dropbox.com/s_B734522FAD400541FDC73A380D2C9E0ACB3E13D65CEF0445E5F7B6F032562D0E_1637937437108_image_2021-11-26_200716.png

