October 3, 2016



Authors:

Vo Van Minh

Bui Thi Thuy Quynh

Contents

[I. INTRODUCTION 1](#_Toc463207795)

[1. Concept 1](#_Toc463207796)

[2. Why to use Angular 2? 1](#_Toc463207797)

[3. Features 1](#_Toc463207798)

[4. Advantages/Disadvantages 1](#_Toc463207799)

[II. ARCHITECTURE 2](#_Toc463207800)

[1. Introduction 2](#_Toc463207801)

[2. Modules 4](#_Toc463207802)

[3. Components 6](#_Toc463207803)

[4. Templates 8](#_Toc463207804)

[5. Directives 14](#_Toc463207805)

[6. Pipes 15](#_Toc463207806)

[6.1. Using pipes 15](#_Toc463207807)

[6.2. Parameterizing a Pipe 16](#_Toc463207808)

[6.3. Custom Pipes 18](#_Toc463207809)

[6.4. Pure and Impure Pipes 21](#_Toc463207810)

[7. Services 25](#_Toc463207811)

[8. Forms 28](#_Toc463207812)

[III. TUTORIALS 42](#_Toc463207813)

[1. Install 42](#_Toc463207814)

[1.1. Install nodejs v6.x.x and npm v3.5.x 42](#_Toc463207815)

[1.2. The tool options 43](#_Toc463207816)

[2. Build the application 43](#_Toc463207817)

# I. INTRODUCTION

## 1. Concept

* Angular is a framework for building client applications in HTML and either JavaScript or a language (like Dart or TypeScript) that compiles to JavaScript.
* Provides a way to organize your HTML, JavaScript and CSS to keep your front-end code clean.
* Developed by Misko Hevery at Brat Tech LLC, 2009. Mainly maintained by Google with the help of the open-source community.
* The beta version of Angular 2 has been released in the March of 2014.

## 2. Why to use Angular 2?

* Angular 2 is simpler than Angular 1 and its fewer concepts make it easier to understand.
* You can update the large data sets with minimal memory overhead.
* It will speed up the initial load through server side rendering.

## 3. Features

* Angular 2 often used to develop front-end.
* It supports latest the version of browsers and also supports old browsers including IE9+ and Android 4.1+.
* It is a cross platform framework.
* Angular 2 is mainly focused on mobile apps.
* Code structure is very simplified than the previous version of Angular.

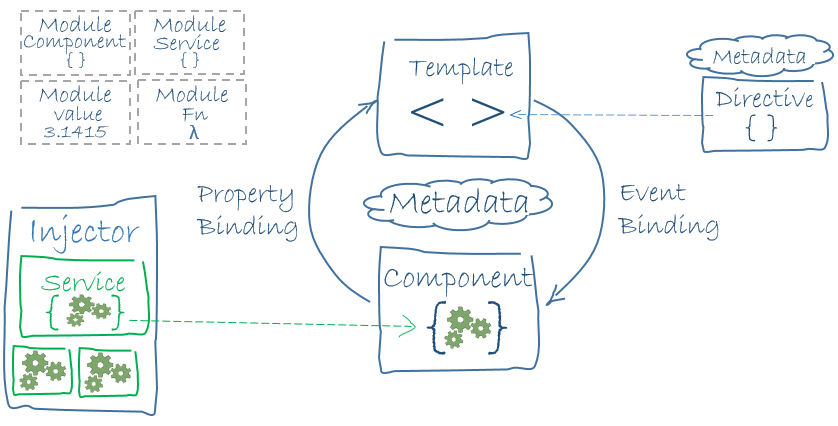
## 4. Advantages/Disadvantages

* *Advantages*
* If an application is a heavy load, then Angular 2 keep it fully UI responsive.
* It uses server side rendering for fast views on mobile.
* It works well with ECMAScript (JavaScript) and other languages that compile to JavaScript.
* It uses dependency injection to maintain applications without writing too long code.
* Everything will be the component based approach.
* *Disadvantages*
* Since Angular 2 is a newly introduced framework, there is less online community support.
* It takes time to learn if you are new to Angular 2.

# II. ARCHITECTURE

1. Introduction

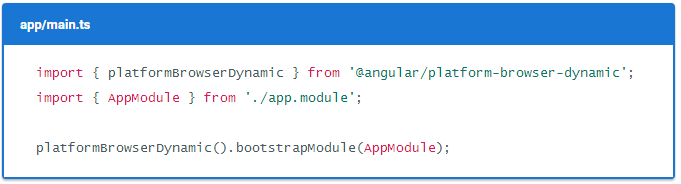
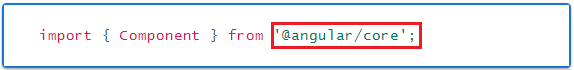
* Angular is a framework for building client applications in HTML and either JavaScript or a language (like Dart or TypeScript) that compiles to JavaScript. This documentation use TypeScript language.
* The framework consists of several libraries, some of them core and some optional.
* Here is how we can write an angular basic projects:
* The first, we write Angular applications by composing HTML *templates* with Angularized markup, writing *component* classes to manage those templates, adding application logic in *services*, and boxing components and services in *modules*.
* Then we launch the app by *bootstrapping* the *root module*. Angular takes over, presenting your application content in a browser and responding to user interactions according to the instructions you've provided.
* Of course, there is more to it than this. We'll learn the details in this documentation that follow. For now, focus on the big picture:



The architecture diagram identifies the main building blocks of an Angular application:

* Modules
* Components
* Template
* Directives
* Pipes
* Services
* Forms

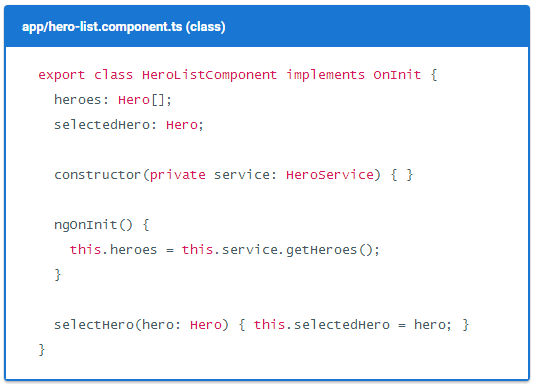
## 2. Modules

* In Angular 2, a module (NgModule) is a mechanism to group components, directives, pipes and services into cohesive block of functionality that are related that can be combined with other modules to create an application. Every Angular app has at least one module, the *root module*, conventionally named AppModule.
* An Angular module, whether a *root* or *feature*, is a class with an @NgModule decorator.
* NgModule is a decorator function that takes a single metadata object whose properties describe the module. The most important properties are:
* *declarations* - the *view classes* that belong to this module. Angular has three kinds of view classes: components, [directives](https://angular.io/docs/ts/latest/guide/architecture.html#directives), and [pipes](https://angular.io/docs/ts/latest/guide/pipes.html).
* *exports* - the subset of declarations that should be visible and usable in the component [templates](https://angular.io/docs/ts/latest/guide/architecture.html#templates) of other modules.
* *imports* - other modules whose exported classes are needed by component templates declared in *this* module.
* *providers* - creators of [services](https://angular.io/docs/ts/latest/guide/architecture.html#services) that this module contributes to the global collection of services; they become accessible in all parts of the app.
* *bootstrap* - the main application view, called the *root component*, that hosts all other app views. Only the *root module* should set this bootstrap property.
* Here's a simple root module:*NOTE:* The export of *AppComponent* is just to show how to export; it isn't actually necessary in this example. A root module has no reason to *export* anything because other components don't need to *import* the root module.
* To launch an Angular application by *bootstrapping* its root module (AppModule) using a *main.ts* file like this one:.
* Angular libraries: A collection of Javascript modules is called a Libraries module. Each Angular library name begins with the @angular prefix. You install them with the *npm* package manager and import parts of them with JavaScript import statements.

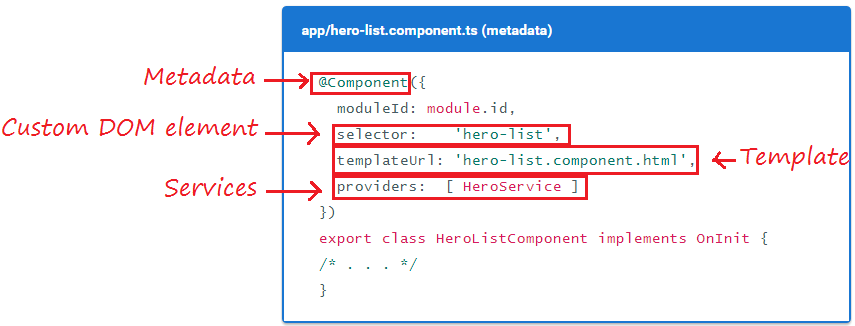
*NOTE*: Angular applications and Angular itself depend upon features and functionality provided by a variety of third-party packages. These packages are maintained and installed with the *Node Package Manager (*[*npm*](https://docs.npmjs.com/)*).*

## 3. Components

* Component encapsulates the template, data and behaviour of the view.
* Component classes should be clean. They don't fetch data from the server, validate user input, or log directly to the console. They delegate such tasks to services.
* A component mediates between the view (rendered by the template) and the application logic (which often includes some notion of a model). A good component presents properties and methods for data binding.
* For example, the *HeroListComponent* has a *heroes* property that returns an array of heroes that it acquires from a service. *HeroListComponent* also has a *selectHero*() method that sets a *selectedHero* property when the user clicks to choose a hero from that list.



* But in fact, *HeroListComponent* really is *just a class*. It's not a component until you *tell Angular about it*.
* To tell Angular that *HeroListComponent* is a component, attach *metadata* to the class.
* In TypeScript, you attach metadata by using a *decorator*. Here's some metadata for *HeroListComponent*:

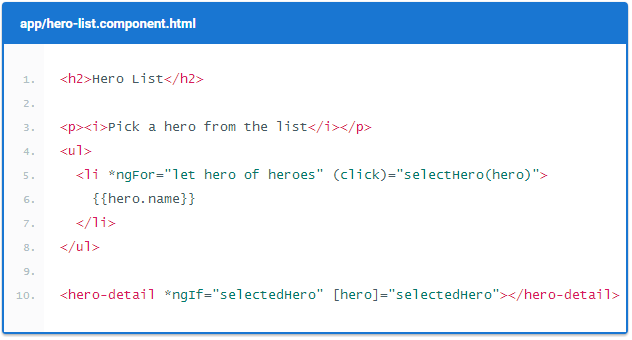


* Here are a few of the possible @Component configuration options:
* *moduleId* - sets the source of the base address (*module.id*) for module-relative URLs such as the *templateUrl*.
* *selector* - CSS selector that tells Angular to create and insert an instance of this component where it finds a <hero-list> tag in *parent* HTML. For example, if an app's HTML contains <hero-list></hero-list>, then Angular inserts an instance of the *HeroListComponent* view between those tags.
* *templateUrl* - module-relative address of this component's HTML template, shown [above](https://angular.io/docs/ts/latest/guide/architecture.html#templates).
* *providers* - array of dependency injection providers for services that the component requires. This is one way to tell Angular that the component's constructor requires a *HeroService* so it can get the list of heroes to display.

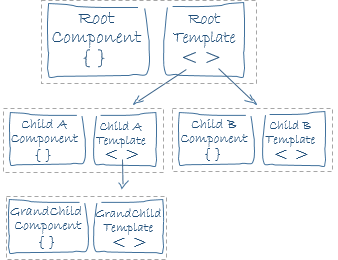
*NOTE*: Decorators are functions that modify JavaScript classes. Angular has many decorators that attach metadata to classes so that it knows what those classes mean and how they should work. [Learn more](https://medium.com/google-developers/exploring-es7-decorators-76ecb65fb841#.x5c2ndtx0) about decorators on the web.

## 4. Templates

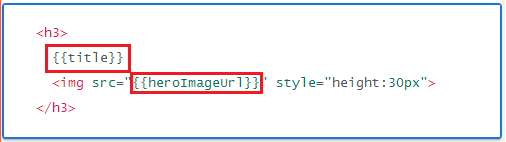
* A template is a form of HTML that tells Angular how to render the component.
* A template looks like regular HTML, except for a few differences. Here is a template for our *HeroListComponent*:



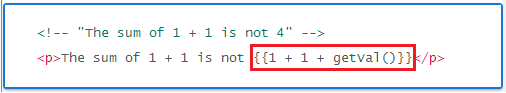
* Some of elements like <h2> and <p> are HTML elements but code like \*ngFor, {{hero.name}}, (click), [hero] and <hero-detail> uses Angular's [template syntax](https://angular.io/docs/ts/latest/guide/template-syntax.html).
* The <hero-detail> tag in the last line of the template is a custom element that represents a new component, *HeroDetailComponent*. The *HeroDetailComponent* is a child of the *HeroListComponent*.



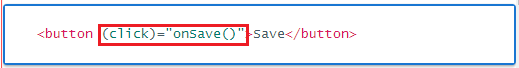
* We’ll cover these basic elements of template syntax:
* *HTML* is the language of the Angular template. Almost all HTML syntax is valid template syntax. The <script> element is forbidden in Angular template because of eliminating the risk of script injection attacks but in practice, <script> is simply ignored. Some legal HTML doesn’t make much sense in a template like <html>, <body>, and <base>. We can extend the HTML vocabulary with components and directives that appear as new *elements and attributes.*
* *Interpolation* is used to weave calculated strings into the text between HTML element tags and within attribute assignments. It is presented by the double-curly braces {{}}*.* Examples:



* *Template expression:* A template expression produces a value. Angular executes the expression and assigns it to a property of a binding target; the target might be an HTML element, a component, or a directive. Template expressions can be written in a language that looks like JavaScript. Many JavaScript expressions are legal template expressions, but not all. Some of JavaScript expressions are prohibited, including *assignments* (=, +=, -=, ...), *new,* chaining expressions with ‘;’ or ‘,’ , increment and decrement operators (++ and --), no support for the bitwise operators ‘|’ and ‘&’ and new [template expression operators](https://angular.io/docs/ts/latest/guide/template-syntax.html#expression-operators), such as ‘|’ and ‘?’. Examples:

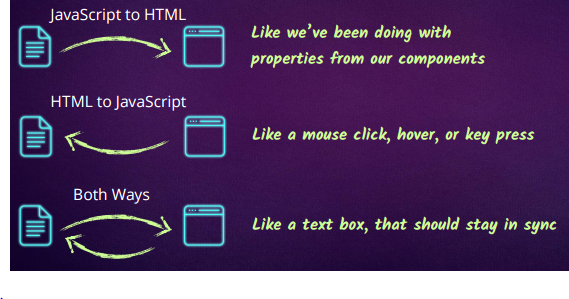


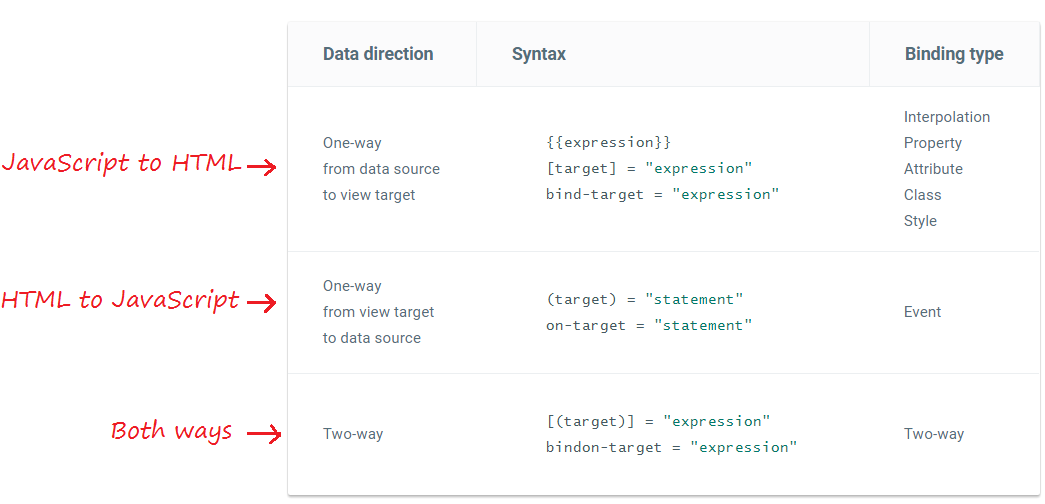
* *Template statement* responds to an event raised by a binding target such as an element, component, or directive. Syntax: *(event)="statement"*. Like template expressions, template *statements* use a language that looks like JavaScript but it supports both basic assignment (=) and chaining expressions (with ; or ,). Examples:



* *Binding syntax*:

*Data binding* is a mechanism for coordinating what users see with application data values. Angular provides many kinds of data binding but they can group into three categories by the direction in which data flows:

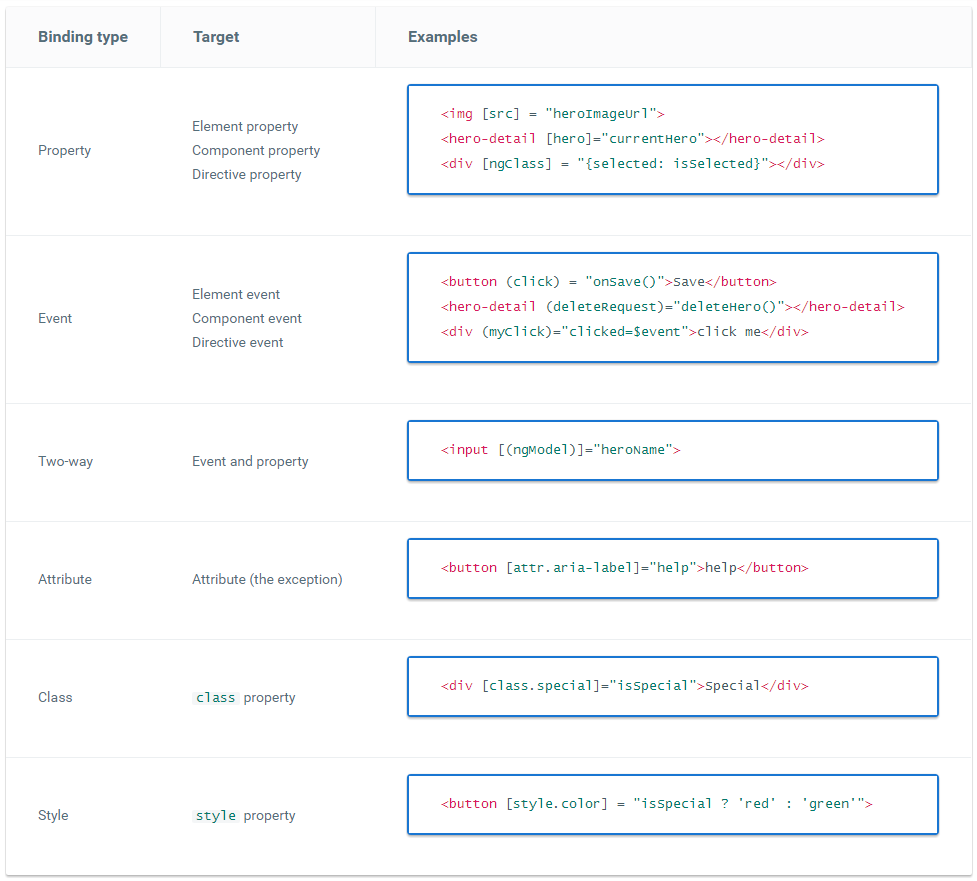




Some of data binding type often use:

* *Interpolation* is used to weave calculated strings into the text between HTML element tags and within attribute assignments.
* *Property binding* is used to set a property of a view element to the value of a [template expression](https://angular.io/docs/ts/latest/guide/template-syntax.html#template-expressions).
* *Event binding* is used to know about a user action is to listen for certain events such as keystrokes, mouse movements, clicks, and touches.
* *Two-way binding with NgModel (combines property and event binding)* is used to display a data property and update that property when the user makes changes.

And this is syntax and examples for each type:

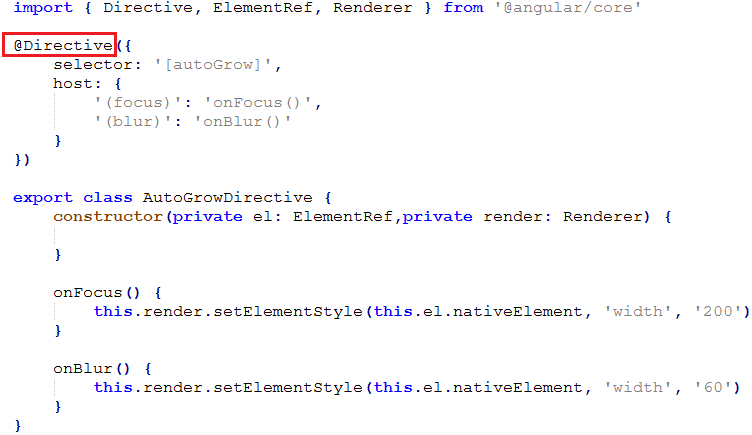


NOTE: Template binding works with DOM *properties* and *events*, not HTML *attributes. Property values can change; attribute values can't.*

* *Built-in directives:* Some of the most frequently used built-in directives: NgClass, NgStyle, NgIf, NgFor,.... You also can refer how to use them at [here](https://angular.io/docs/ts/latest/guide/template-syntax.html#directives).
* You can refer more about Angular template syntax at [template syntax](https://angular.io/docs/ts/latest/guide/template-syntax.html).

## 5. Directives

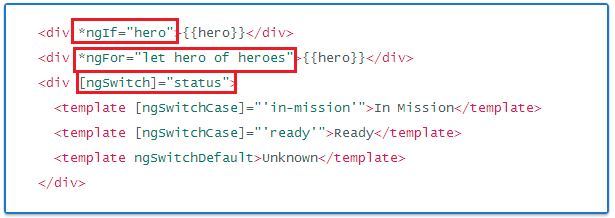
* Angular templates are *dynamic*. When Angular renders them, it transforms the DOM according to the instructions given by directives. Directives is a class that allows to extend or control DOM. We can define custom attributes or custom elements for DOM.
* A directive is a class with directive metadata. In TypeScript, apply the @Directive decorator to attach metadata to the class.



* Here are a few of the possible @Directives configuration options:
* *selector* - CSS selector that tells Angular to create and insert an instance of this directive where it finds a *autoGrow* attribute in *parent* HTML. For example, if an app's HTML contains <input type="text" autoGrow />, then Angular inserts an instance of the *AutoGrowDirective* attribute for this element.
* *host* - is used to describe to events raised from this element. In this examples, *(focus)* event is handled by *onFocus()* method of *AutoGrowDirective .*
* There are three kinds of Angular directives:
* *Components* - A *component* is a *directive-with-a-template*; a @Component decorator is actually a @Directive decorator extended with template-oriented features. It's the most common of the three directives.
* *Attribute directives* - *Attribute directives* alter the appearance or behavior of an existing element. Examples



* *Structural directives* - *Structural directives* alter layout by adding, removing, and replacing elements in DOM. Examples:

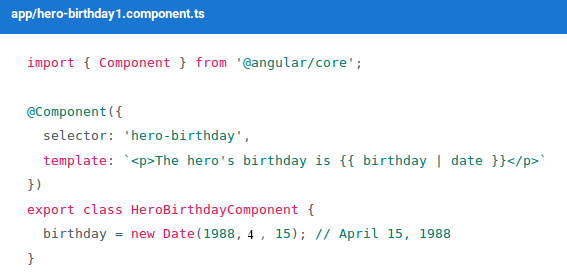


## 6. Pipes

*Pipes* transform displayed values within a template.

### 6.1. Using pipes

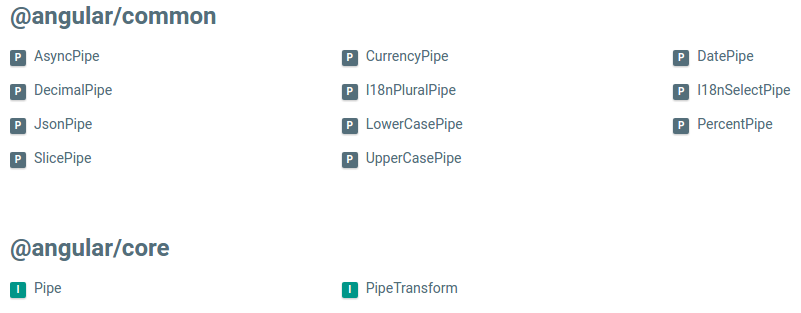
* A pipe takes in data as input and transforms it to a desired output.
* For examples, transform from a component's *birthday* property into a *human-friendly date*.



* Focus on the component's template:

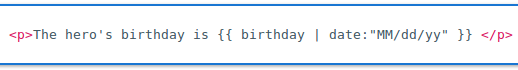
<p>The hero's *birthday* is {{ birthday | date }}</p>

* Inside the interpolation expression we flow the component's *birthday* value through the pipe operator ( | ) to the Date pipe function on the right. All pipes work this way.
* Some of pipes in Angular:

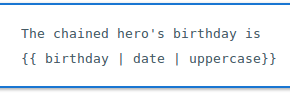


### 6.2. Parameterizing a Pipe

* A pipe may accept any number of optional parameters to fine-tune its output. We can add parameters to a pipe by following the pipe name with a colon ( : ) and then the parameter value (e.g., currency:'EUR'). If our pipe accepts multiple parameters, we separate the values with colons (e.g. slice:1:5)
* We'll modify our *birthday* template to give the date pipe a format parameter. After formatting the hero's April 15th *birthday*, it should render as 04/15/88:



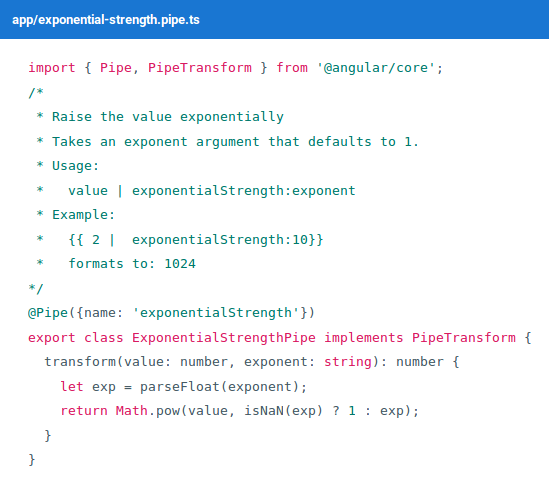
* We can chain pipes together in potentially useful combinations. In the following example, we chain the birthday to the *DatePipe* and on to the *UpperCasePipe* so we can display the birthday in uppercase.



* This example - which displays FRIDAY, APRIL 15, 1988 - chains the same pipes as above, but passes in a parameter to date as well.

### 6.3. Custom Pipes

* We can write our own custom pipes. Here's a custom pipe named *exponentialStrengthPipe* that can boost a hero's *powers*:

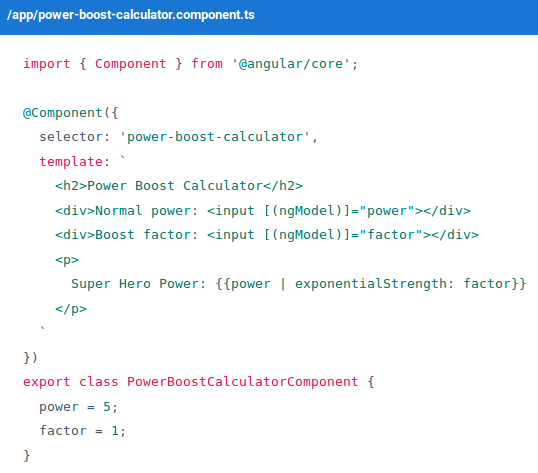


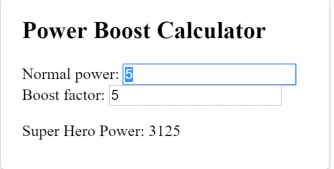
* This pipe definition reveals several key points:
* A pipe is a class decorated with pipe metadata.
* The pipe class implements the *PipeTransform* interface's *transform* method that accepts an input value followed by optional parameters and returns the transformed *value*.
* There will be one additional argument to the transform method for each parameter passed to the pipe. Our pipe has one such parameter: *the exponent*.
* We tell Angular that this is a pipe by applying the @Pipe decorator which we import from the core Angular library.
* The @Pipe decorator allows us to define the pipe name that we'll use within template expressions. It must be a valid JavaScript identifier. Our pipe's name is *exponentialStrength*.
* Now we need a component to demonstrate our pipe.



NOTE: We must include our pipe in the declarations array of the AppModule.

* We could upgrade the example to a "*Power Boost Calculator*" that combines our pipe and two-way data binding with *ngModel*.





### 6.4. Pure and Impure Pipes

* There are two categories of pipes: *pure* and *impure*. Pipes are *pure* by default. Every pipe we've seen so far has been pure. We make a pipe impure by setting its *pure flag* to false.
* Pure pipes
* Angular executes a *pure pipe* only when it detects a *pure change* to the input value. A *pure change* is *either* a change to a primitive input value (String, Number, Boolean, Symbol) *or* a changed object reference (Date, Array, Function, Object).
* Angular ignores changes within (composite) objects. It won't call a pure pipe if we change an input month, add to an input array, or update an input object property.
* Impure pipes
* Angular executes an *impure pipe* during *every* component change detection cycle.
* An impure pipe will be called a lot, as often as every keystroke or mouse-move.
* The impure AsyncPipe

The AsyncPipe accepts a Promise or Observable as input and subscribes to the input automatically, eventually returning the emitted value(s).

* A Promise handles a single callback when an async operation completes or fails.
* An Observable is like a Stream and allows to pass zero or more events where the callback is called for each event.
* Often Observable is preferred over Promise because it provides the features of Promise and more.
* In this next example, this is how to bind an *Observable* of message strings (message$) to a view with the *async pipe*.



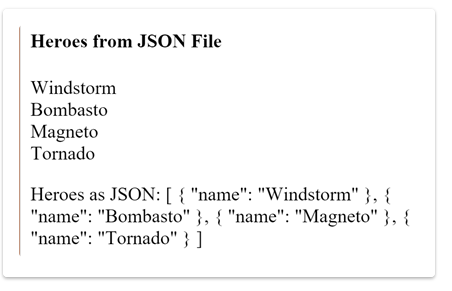
* An impure caching pipe
* Let's write one more *impure pipe*, a pipe that makes an HTTP request to the server. Here's the code, which uses the Angular *http facility* to retrieve a *heroes.json* file:



* Then we demonstrate it in a harness component whose template defines two bindings to this pipe.

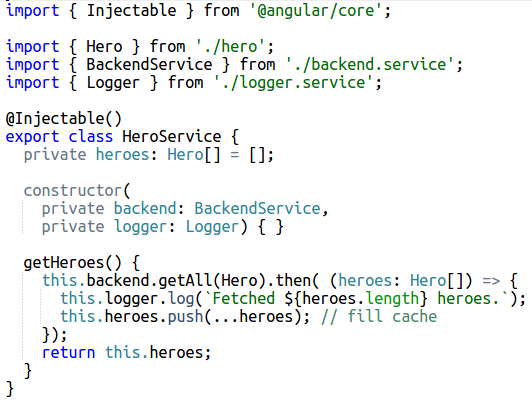


* JsonPipe
* The second binding involving the *FetchPipe* uses more pipe chaining. We take the same fetched results displayed in the first binding and display them again, this time in JSON format by chaining through to the built-in JsonPipe.
* The component renders like this:

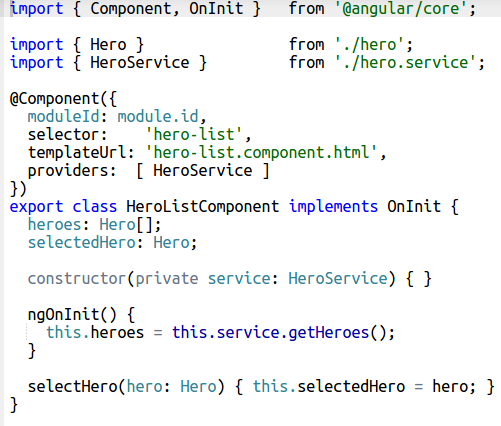


## 7. Services

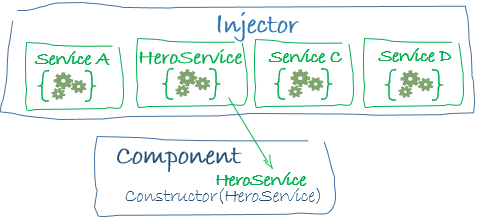
* *Service* is a broad category encompassing any value, function, or feature that your application needs.
* Almost anything can be a service. A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well.
* Examples include:
* logging service
* data service
* message bus
* tax calculator
* application configuration
* Here's an example of a service class:



* Here's a *HeroService* that fetches *heroes* and returns them in a resolved *Promise*.
* The *HeroService* depends on the Logger service and another *BackendService* that handles the server communication grunt work.
* Angular make services available to components through *dependency injection*.
* Dependency injection
* *Dependency injection* is a way to supply a new instance of a class with the fully-formed dependencies it requires. Most dependencies are services. Angular uses dependency injection to provide new components with the services they need.
* Angular can tell which services a component needs by looking at the types of its constructor parameters. For example, the constructor of your *HeroListComponent* needs a *HeroService*:



* When Angular creates a component, it first asks an *injector* for the services that the component requires.

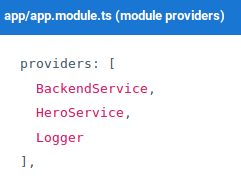


* The *injector* is the main mechanism:
* An injector maintains a *container* of service instances that it created.
* An injector can create a new service instance from a *provider*.

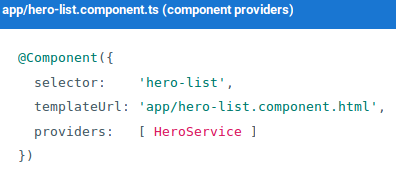
You need previously registered a *provider* of the *HeroService* with the injector. A provider is something that can create or return a service, typically the service class itself.

You can register providers in modules or in components.

* In general, add providers to the root module so that the same instance of a service is available everywhere.



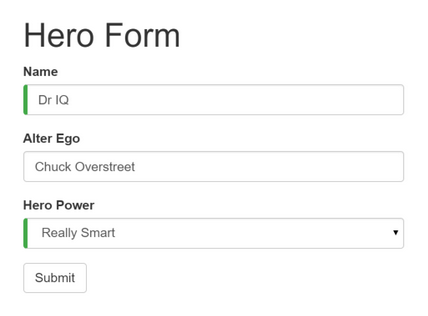
* Alternatively, register at a component level in the providers property of the @Component metadata:



Registering at a component level means you get a new instance of the service with each new instance of that component.

## 8. Forms

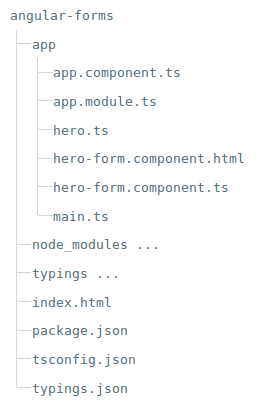
* A form creates a cohesive, effective, and compelling data entry experience. An Angular form coordinates a set of data-bound user controls, tracks changes, validates input, and presents errors.
* An Angular form has two parts: an HTML-based template and a code-based Component to handle data and user interactions.
* We'll discuss and learn to build the following a form:



We will build this form in the following sequence of small steps:

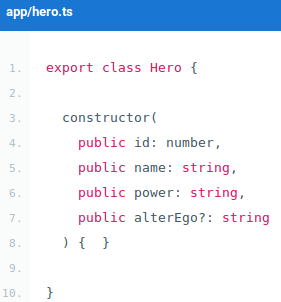
1. Create the *Hero model* class
2. Create the *component* that controls the form
3. Create a *template* with the initial form layout
4. Bind data properties to each form input control with the ngModel two-way data binding syntax
5. Track change-state and validity with ngModel
6. Add custom CSS to provide visual feedback
7. Show and hide validation error messages
8. Handle form submission with *ngSubmit*
9. Disable the form’s submit button until the form is valid

Our project folder structure should look like this:

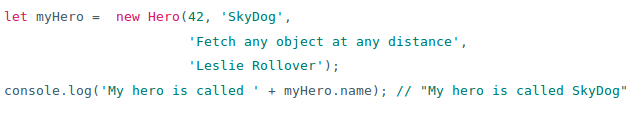


1. Create the *Hero* model class

* A model can be as simple as a "property bag" that holds facts about a thing of application importance. That describes well our *Hero class* with its three required fields (id, name, power) and one optional field (alterEgo).
* Create a new file in the app folder called *hero.ts* and give it the following class definition:

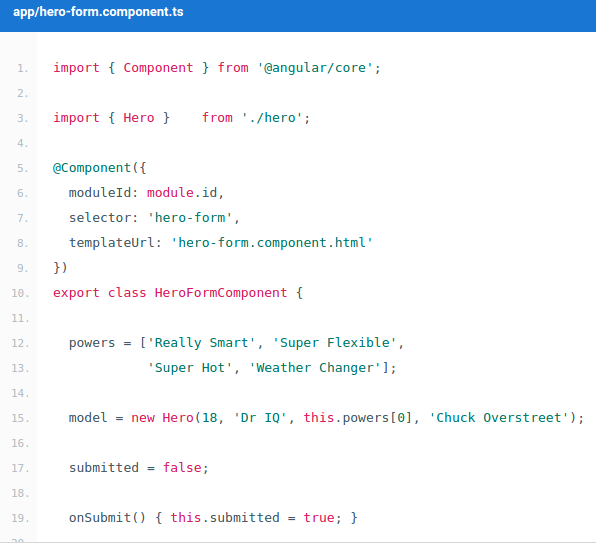


* The TypeScript compiler generates a public field for each public constructor parameter and assigns the parameter’s value to that field automatically when we create new heroes.
* The *alterEgo* is optional and the constructor lets us omit it; note the (?) in alterEgo?.
* We can create a new hero like this:



1. Create the component that controls the form

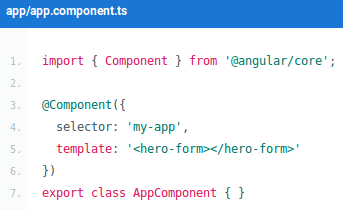
* Create a new file called *hero-form.component.ts* and give it the following definition:



* Understanding this component requires only the Angular concepts we’ve learned in previous chapters:
* We import the Component decorator from the Angular library as we usually do.
* We import the *Hero model* we just created.
* The @Component selector value of "hero-form" means we can drop this form in a parent template with a <hero-form> tag.
* The templateUrl property points to a separate file for the template HTML called *hero-form.component.html*.
* Revise the app.module.ts
* app.module.ts defines the application's root module. In it we identify the external modules we'll use in our application and declare the components that belong to this module, such as our *HeroFormComponent*.
* Because template-driven forms are in their own module, we need to add the *FormsModule* to the array of imports for our application module before we can use forms.



* Revise the app.component.ts
* app.component.ts is the application's root component. It will host our new *HeroFormComponent*.

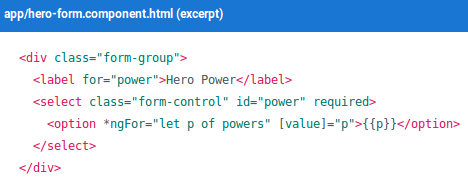


1. Create a template with the initial form layout

* Create a new template file called *hero-form.component.html* and give it the following definition:

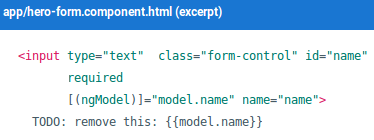


* We're presenting two of the *Hero fields*, *name* and *alterEgo*, and opening them up for user input in input boxes.
* The *Name* <input> control has the HTML5 required attribute.
* The *Alter Ego* <input> control does not because *alterEgo* is optional.
* We've got a *Submit* button at the bottom with some classes on it for styling.
* Add Powers with \*ngFor:
* Our hero may choose one *super power* from a fixed list of Agency-approved powers. We maintain that list internally (in *HeroFormComponent*).
* We'll add a select to our form and bind the options to the powers list using *ngFor.*
* Add the following HTML *immediately below* the *Alter Ego* group.



1. Bind data properties to each form input control with the *ngModel* two-way data binding syntax

* In *hero-form.component.html* file, find the <input> tag for the "Name" and update it like this:



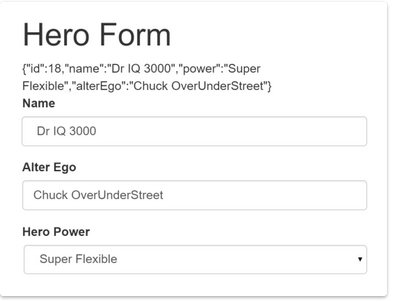
* If we ran the app right now and started typing in the *Name* input box, adding and deleting characters, we'd see them appearing and disappearing from the interpolated text. At some point it might look like this.



The diagnostic is evidence that we really are flowing values from the input box to the model and back again. That's two-way data binding.

* Let's add similar [(ngModel)] bindings and name attributes to *Alter Ego* and *Hero Power*. We'll ditch the input box binding message and add a new binding at the top to the component's diagnostic property. Then we can confirm that two-way data binding works *for the entire Hero model*.

After revision the core of our form should have three [(ngModel)] bindings and name attributes that look much like this:

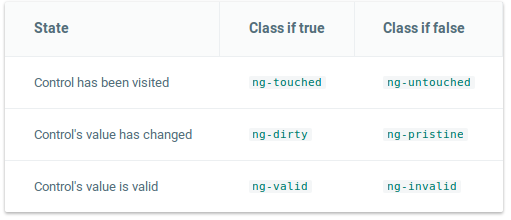


* If we ran the app right now and changed every *Hero model* property, the form might display like this:

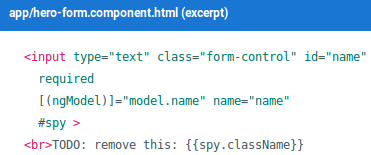
The diagnostic near the top of the form confirms that all of our changes are reflected in the model.

1. Track change-state and validity with ngModel

* A form isn't just about data binding. We'd also like to know the state of the controls on our form.
* Using *ngModel* in a form gives us more than just two way data binding. It also tells us if the user touched the control, if the value changed, or if the value became invalid.
* The *NgModel* directive doesn't just track state; it updates the control with special Angular CSS classes that reflect the state. We can leverage those class names to change the appearance of the control and make messages appear or disappear.

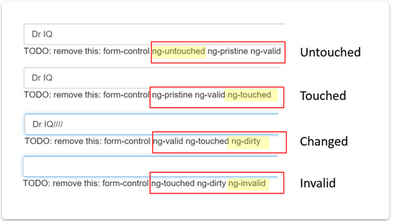


* Let's add a temporary template reference variable named *spy* to the "Name" <input> tag and use the spy to display those classes.



* Now run the app and focus on the *Name* input box. Follow the next four steps *precisely*
* Look but don't touch
* Click in the input box, then click outside the text input box
* Add slashes to the end of the name
* Erase the name

We should be able to see the following four sets of class names and their transitions:

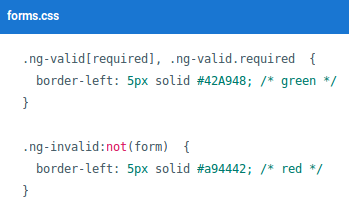


1. Add custom CSS to provide visual feedback

* We realize we can mark required fields and invalid data at the same time with a colored bar on the left of the input box:

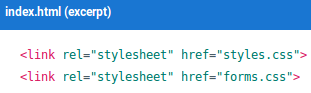


* We achieve this effect by adding two styles to a new forms.css file that we add to our project as a sibling to *index.html*.



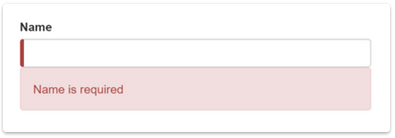
These styles select for the two Angular validity classes and the HTML 5 "required" attribute.

* We update the <head> of the *index.html* to include this style sheet.



1. Show and hide validation error messages

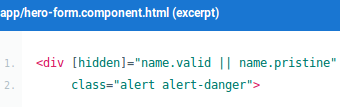
* The "Name" input box is required. Clearing it turns the bar red. That says *something* is wrong but we don't know *what* is wrong or what to do about it. We can leverage the ng-invalid class to reveal a helpful message.



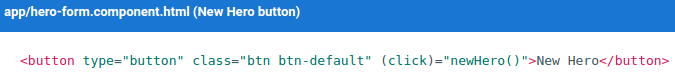
* To achieve this effect we extend the <input> tag with
* a template reference variable
* the "*is required*" message in a nearby <div> which we'll display only if the control is invalid.
* Here's how we do it for the *name* input box:



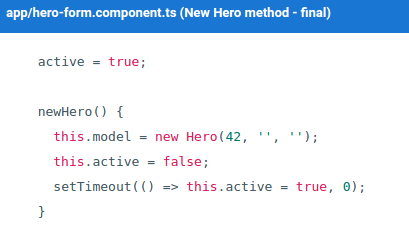
* We need a template reference variable to access the input box's Angular control from within the template. Here we created a variable called name and gave it the value "ngModel".
* Now we can control visibility of the "name" error message by binding properties of the name control to the message <div> element's hidden property.



* Add a *hero* and reset the form:
* We'd like to add a new hero in this form. We place a "New Hero" button at the bottom of the form and bind its click event to a component method.



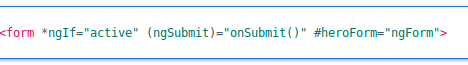
* We'll have to reset the form controls manually with a small trick. We add an active flag to the component, initialized to true. When we add a new hero, we toggle active false and then immediately back to true with a quick setTimeout.



1. Handle form submission with *ngSubmit*

* The user should be able to submit this form after filling it in. The Submit button at the bottom of the form does nothing on its own but it will trigger a form submit because of its type (type="submit").

A "form submit" is useless at the moment. To make it useful, we'll update the <form> tag with another Angular directive, *NgSubmit*, and bind it to the *HeroFormComponent.submit()* method with an event binding

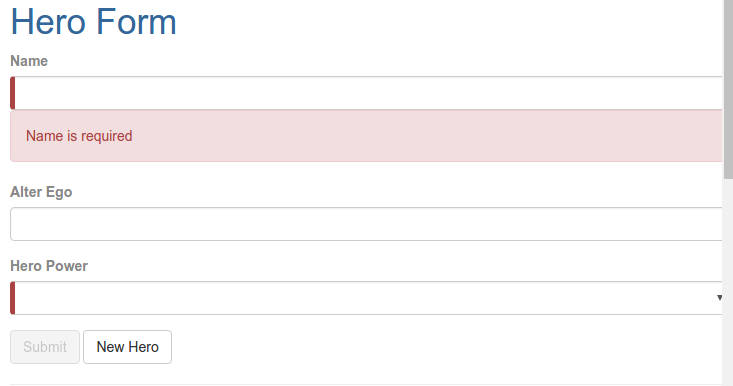


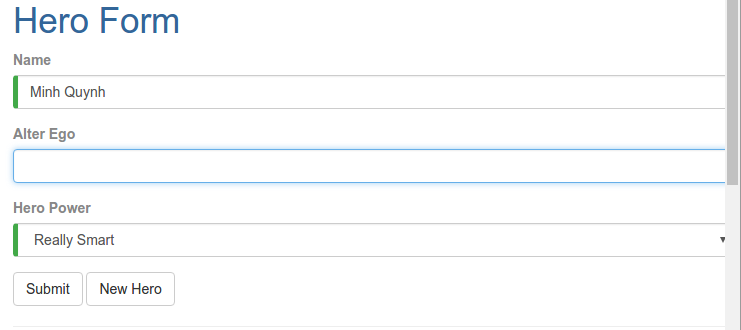
1. Disable the form’s submit button until the form is valid

* Later in the template we bind the button's disabled property to the form's over-all validity via the heroForm variable. Here's that bit of markup:



* Here are some pictures of the project:





# III. TUTORIALS

## 1. Install

### 1.1. Install nodejs v6.x.x and npm v3.5.x

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.

* On Windows:
* Download and Install at <https://nodejs.org/en/download/>
* On Ubuntu 16.04:
* Step 1: curl -sL https://deb.nodesource.com/setup\_6.x | sudo -E bash -
* Step 2: sudo apt-get install -y nodejs
* You can test it with the following command: node -v or npm -v

### 1.2. The tool options

* angular-cli: A command line interface for Angular.
* Install: npm install -g angular-cli
* Create a new project: ng new projectName
* Run: cd projectName & ng serve
* Use eclipse:
* Install 2 plugin are angular2-eclipse and TypeScript.
* Other:
* Sublime text
* Notepad
* ...
* NOTE: This document use only NodeJS and Text Editor to implement the code on Ubuntu 16.04.

## 2. Build the application

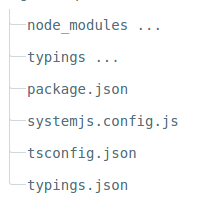
* Here are the steps:
* Step 1: Create and configure the project.
* Step 2: Define the web page that hosts the application.
* Step 3: Create your application.
* Step 4: Create model, components, services, html, css,.. and add it to your application.
* Step 5: Start up your application.
* Step 6: Build and run the application

**Step 1: Create and configure the project**

* Create the project folder:
* mkdir angular2
* cd angular2
* Create configuration files: Our typical Angular project needs several configuration files:
* *package.json:* identifies npm package dependencies for the project.
* *tsconfig.json*: defines how the TypeScript compiler generates JavaScript from the project's files.
* *typings.json*: provides additional definition files for libraries that the TypeScript compiler doesn't natively recognize.
* *systemjs.config.js*: provides information to a module loader about where to find application modules, and registers all the necessary packages. It also contains other packages that will be needed by later documentation examples.
* Create each of these files in your project directory. Please refer to our project to get code detail of them.
* Install packages
* Using npm from the command line, install the packages listed in package.json with the command:



* You should now have the following structure in angular2 project:

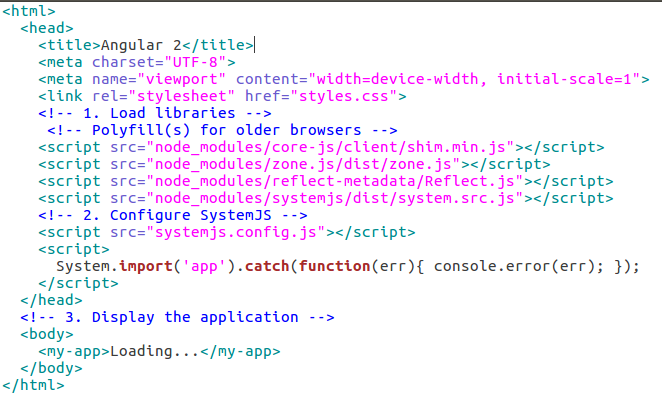


* If the typings folder doesn't show up after running npm install, you'll need to install it manually with the command:



**Step 2: Define the web page that hosts the application.**

* In the project root folder, create an *index.html* file and paste the following lines into it:



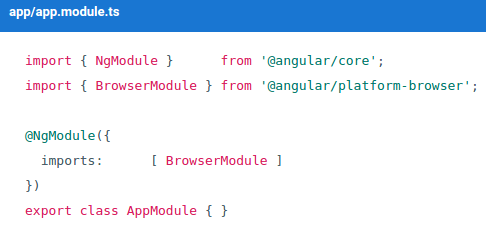
* The noteworthy sections here are:
* JavaScript libraries: core-js polyfills for older browsers, the zone.js and reflect-metadata libraries needed by Angular, and the SystemJS library for module loading.
* Configuration file for SystemJS, and a script where you import and run the app module which refers to the main file that you just wrote.
* The <my-app> tag in the <body> which is where your app lives!

**Step 3: Create your application.**

* Every Angular application has at least one module: the root module, named AppModule here.
* Create an app subfolder off the project root directory:



* Create the file app/app.module.ts with the following content:

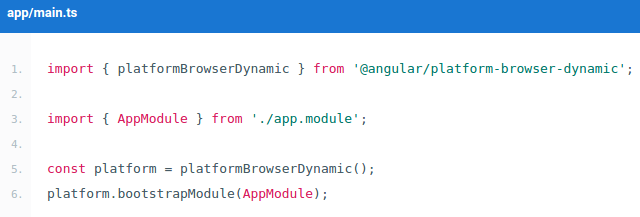


**Step 4: Create model, components, services, html, css,.. and add it to your application.**

* Let's write the components, services,... that needed for your project.
* You can refer to our project which is attached to this document in order to understand more clearly their writing.

**Step 5: Start up your application.**

* Now you need to tell Angular to start up your application.
* Create the file app/main.ts with the following content:



* This code initializes the platform that your application runs in, then uses the platform to bootstrap your AppModule.

**Step 6: Build and run the application**

* Open a terminal window and enter this command:



* That command runs the following two parallel node processes:
* The TypeScript compiler in watch mode.
* A static file server called lite-server that loads index.html in a browser and refreshes the browser when application files change.

REFERENCES

1. <https://angularjs.org/>
2. <https://angular.io/docs/ts/latest/>
3. <https://www.youtube.com/watch?v=_-CD_5YhJTA>