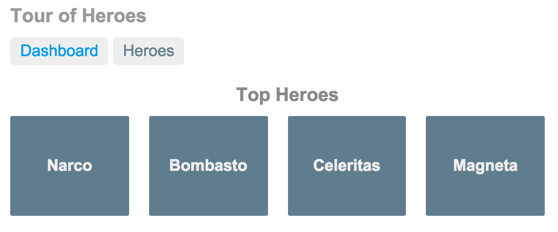
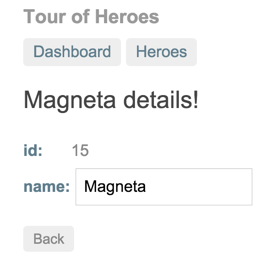
1. use one-way data binding for read-only data.
2. Our final application looks like,



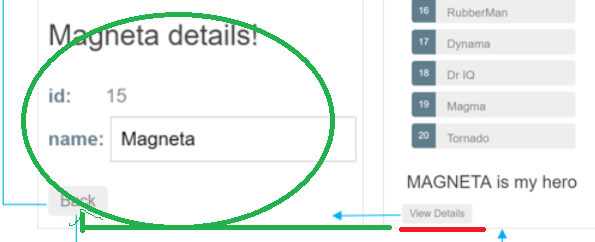
1. You can click the two links above the dashboard ("Dashboard" and "Heroes") to navigate between this Dashboard view and a Heroes view.
2. If you click the dashboard hero "Magneta," the router opens a "**Hero Details**" view where you can change the hero's name.



1. Clicking the "Back" button returns you to the Dashboard. Links at the top take you to either of the main views. If you click "Heroes," the app displays the "Heroes" master list view.
2. When you click a different hero name, the read-only mini detail beneath the list reflects the new choice.



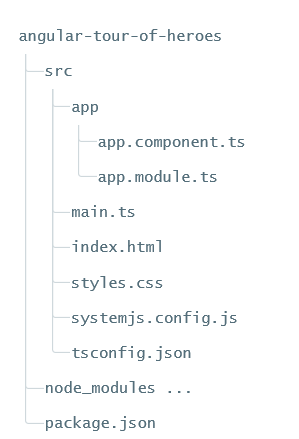
1. You can click the "View Details" button to drill into the editable details of the selected hero.

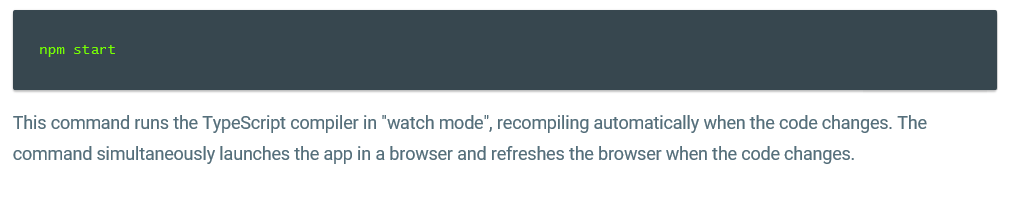
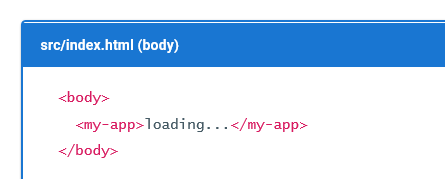


1. The following diagram captures all of the navigation options.

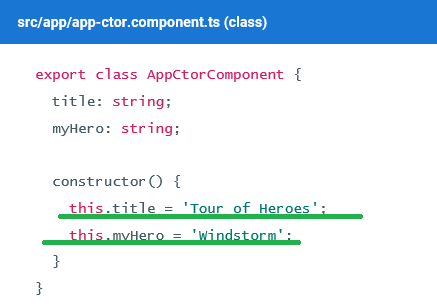


1. The file structure should look like this:

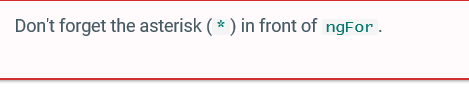


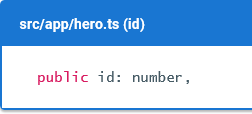
1. 
2. The double curly braces are Angular's interpolation binding syntax.
3. Property binding helps show app data in the UI
4. The template is a multi-line string within ECMAScript 2015 backticks (`). The backtick (`)—which is not the same character as a single quote (')—allows you to compose a string over several lines, which makes the HTML more readable
5. 🡺

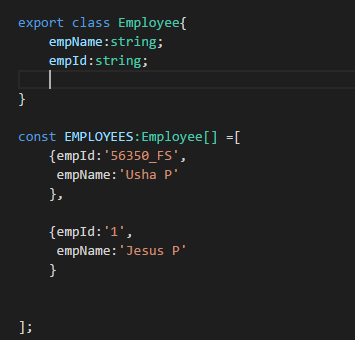
When you bootstrap with the AppComponent class (in main.ts), Angular looks for a <my-app> in the index.html, finds it, instantiates an instance of AppComponent, and renders it inside the <my-app> tag.

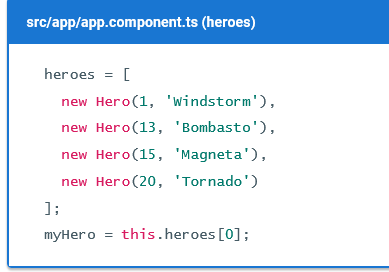
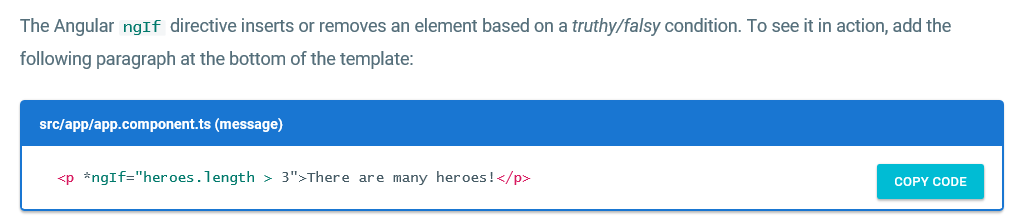
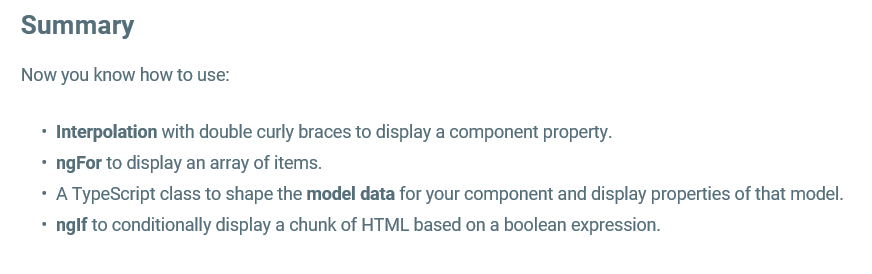
1. Template inline or template file? 🡺 You can store your component's template in one of two places. You can define it inline using the template property, or you can define the template in a separate HTML file and link to it in the component metadata using the @Component decorator's templateUrl property.
2. 
3. 

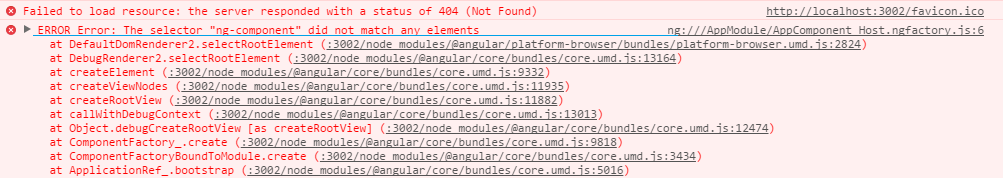




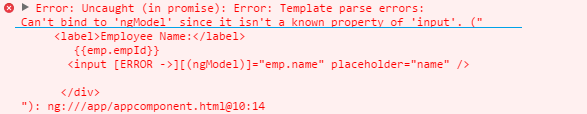
1. 

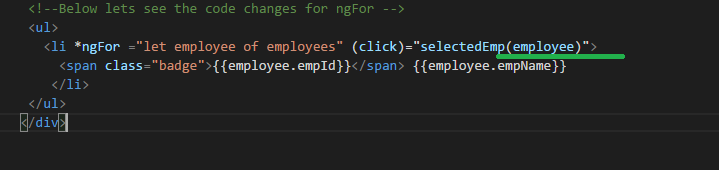
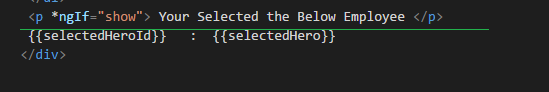


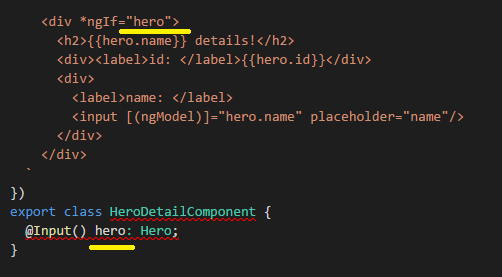
1. 
2. 
3. Don't forget the leading asterisk (\*) in \*ngIf.
4. 
5. Although NgModel is a valid Angular directive, it isn't available by default. It belongs to the optional FormsModule
6. import the FormsModule symbol from the @angular/forms library.
7. @NgModule metadata's imports array, **which contains the list of external modules that the app uses**
8. Practical Scenario🡺 I had not used the Selector in my AppComponent this resulted in the following error



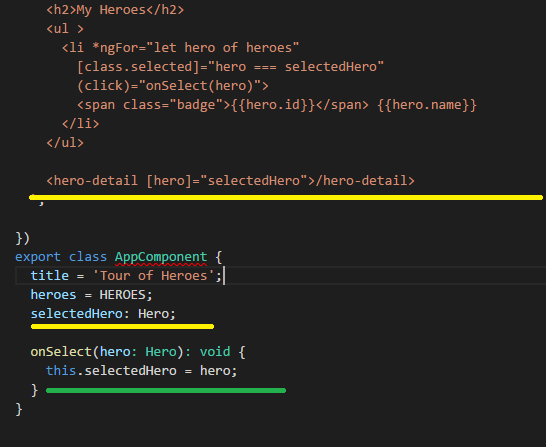
1. Import BrowserModule in the root module and CommonModule in other modules where you want to use common directives
2. CAN I USE THE "const" key word inside the class? 🡺 I was getting some error in the Studio
3. Without the FormModule, the Error in the Console is🡺

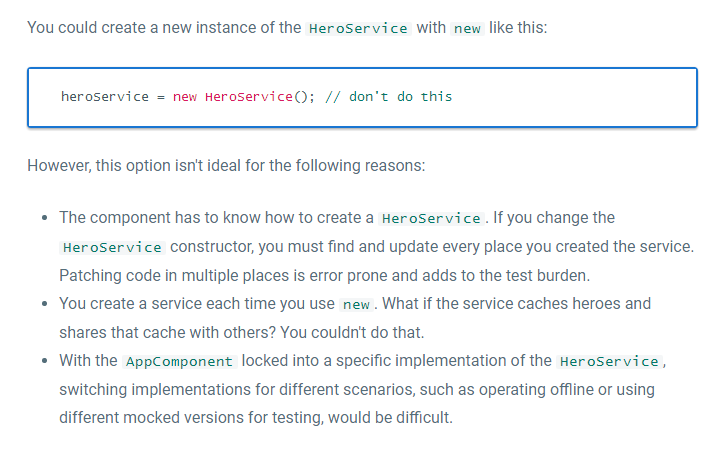


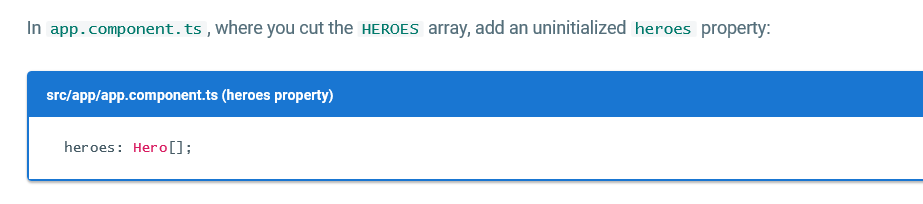
1. 
2. Syntax for \*ngIf is 🡺
3. The @Component decorator provides the Angular metadata for the component.
4. In general, the declarations array contains a list of application components, pipes, and directives that belong to the module.A component must be declared in a module before other components can reference it
5. Note🡺I thought that \*ngIf can accept value only true or false, but its actually like javaScript which accepts, non zero +ve and –ve no, non empty string, any object which is not undefined, for example see the below screen shot



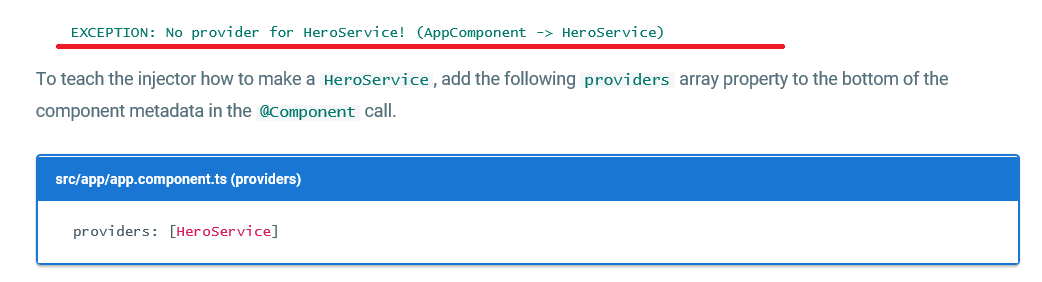
See here it's Quite Different way how the data is sent from parent to child compared to our Video Tutorials



1. Although the HeroService doesn't have any dependencies at this moment, applying the @Injectable() decorator ​from the start ensures consistency and future-proofing.
2. 
3. Don't forget the parentheses. Omitting them leads to an error that's difficult to diagnose.
4. 



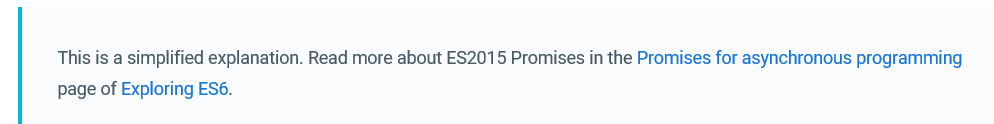
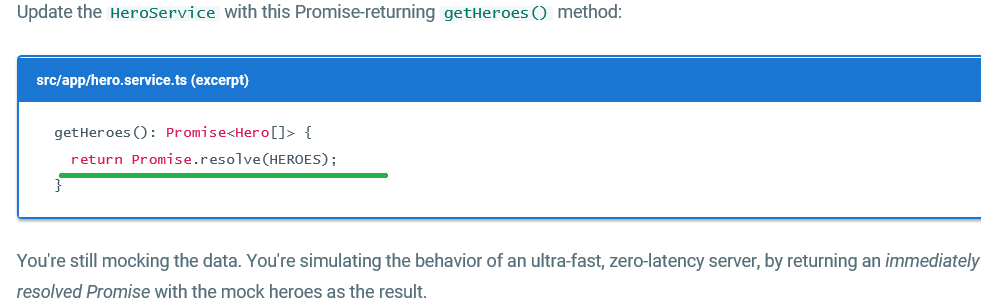


1. 
2. The providers array tells Angular to create a fresh instance of the HeroService when it creates an AppComponent. The AppComponent, as well as its child components, can use that service to get hero data.
3. The ngOnInit lifecycle hook
4. AppComponent should fetch and display hero data with no issues.
5. You might be tempted to call the getHeroes() method in a constructor, but a constructor should not contain complex logic, especially a constructor that calls a server, such as as a data access method. The constructor is for simple initializations, like wiring constructor parameters to properties.
6. To have Angular call getHeroes(), you can implement the Angular *ngOnInit lifecycle hook*. Angular offers interfaces for tapping into critical moments in the component lifecycle: at creation, after each change, and at its eventual destruction.
7. Each interface has a single method. When the component implements that method, Angular calls it at the appropriate time.
8. 

Async services and Promises

1. The HeroService returns a list of mock heroes immediately; its getHeroes() signature is synchronous.
2. Eventually, the hero data will come from a remote server. When using a remote server, users don't have to wait for the server to respond; additionally, you aren't able to block the UI during the wait.
3. To coordinate the view with the response, you can use *Promises*, which is an asynchronous technique that changes the signature of the getHeroes() method.

The hero service makes a Promise

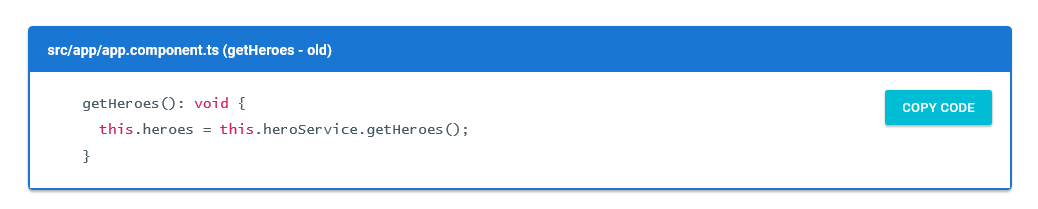
1. A Promise essentially promises to call back when the results are ready. You ask an asynchronous service to do some work and give it a callback function. The service does that work and eventually calls the function with the results or an error.
2. 
3. 

Act on the Promise

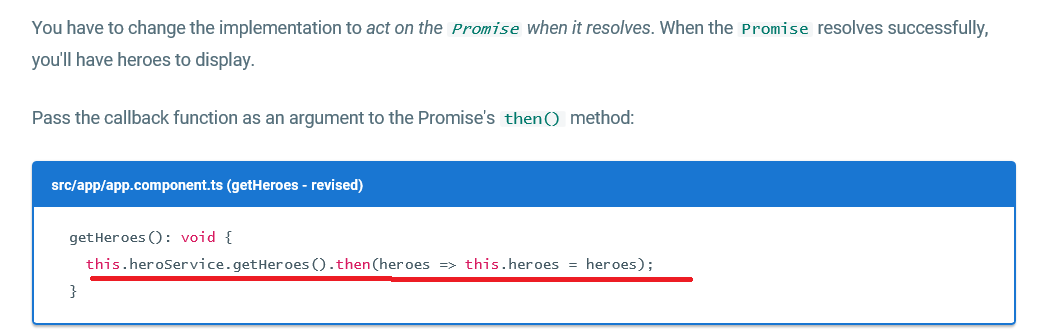
1. As a result of the change to HeroService, this.heroes is now set to a Promise rather than an array of heroes.

Ms 🡺 thus now let’s see the differece/ comparison between older code and latest revisited/modified code in app component

OLDER🡺



LATEST🡺



The callback sets the component's heroes property to the array of heroes returned by the service.

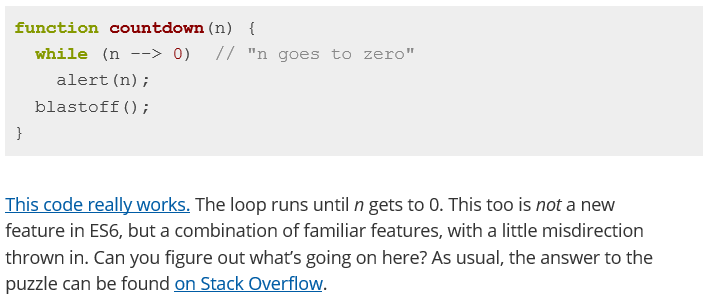
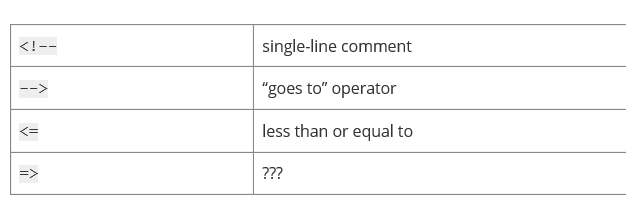
Ms🡺 lets see few concept with respect to Arrow functions,

* 1. Arrow functions make it easy to write anonymous functions, and also bind to the current context.
  2. Example

Case 1: let’s see the syntax for anonymous functions

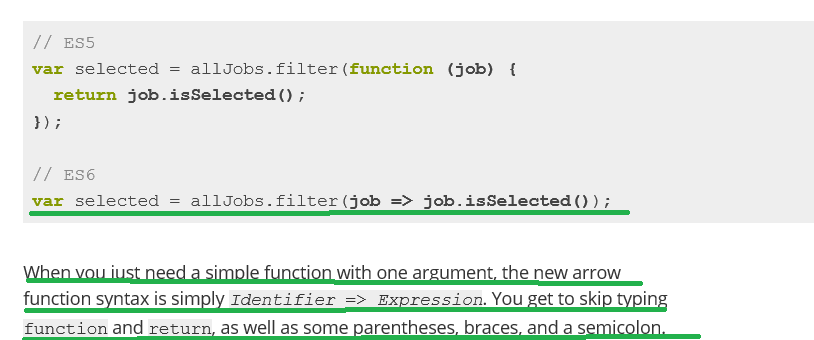


1. Arrow function in Es6🡺 <https://hacks.mozilla.org/2015/06/es6-in-depth-arrow-functions/>
   1. -->

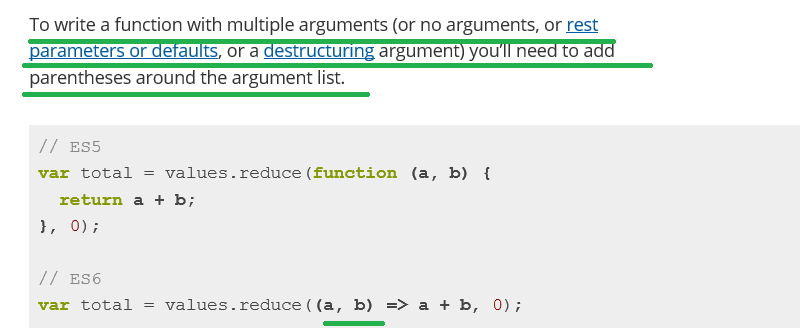
* This arrow indicates a comment *only* when it appears at the start of a line. That’s because in other contexts, --> is an operator in JS, the “goes to” operator!
* 
* 
* What happened to =>? Today, we find out. First, let’s talk a bit about functions.

1. A new arrow in your quiver🡺ES6 introduces a new syntax for writing functions.

Example 1:



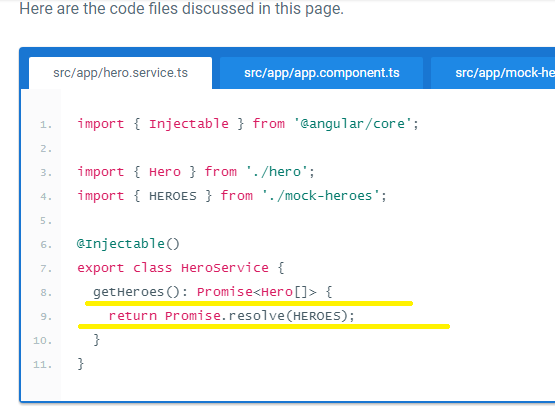
Example 2:



Example 3:

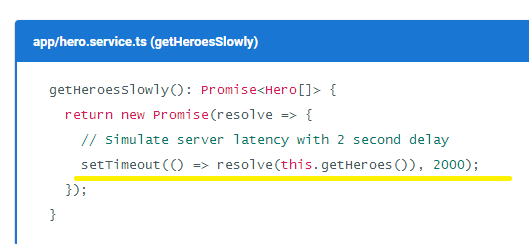


Now let us continue with our Angular Tutorials

1. 

Appendix: Take it slow

1. to simulate a slow connection, import the Hero symbol and add the following **getHeroesSlowly**() method to the HeroService.



1. let us see the difference between the getHeroes() and getHeroesSlowly()

**Like getHeroes(), it also returns a Promise. But this Promise waits two seconds before resolving the Promise with mock heroes.**

**NOTE🡺Back in the AppComponent, replace getHeroes() with getHeroesSlowly() and see how the app behaves.**

**Routing**

1. here we have few set of the requirement

There are new requirements for the Tour of Heroes app:

* Add a *Dashboard* view.
* Add the ability to navigate between the *Heroes* and *Dashboard* views.
* When users click a hero name in either view, navigate to a detail view of the selected hero.
* When users click a *deep link* in an email, open the detail view for a particular hero.

1. That’s finally our app looks like🡺

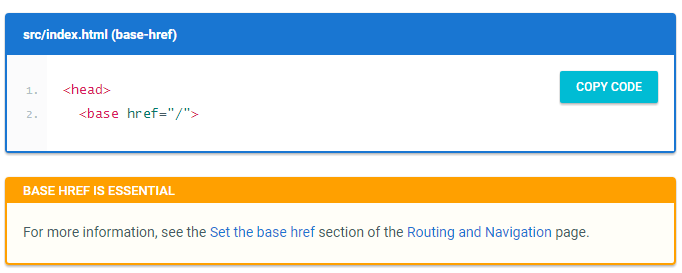


In this part I will just cover the important concept

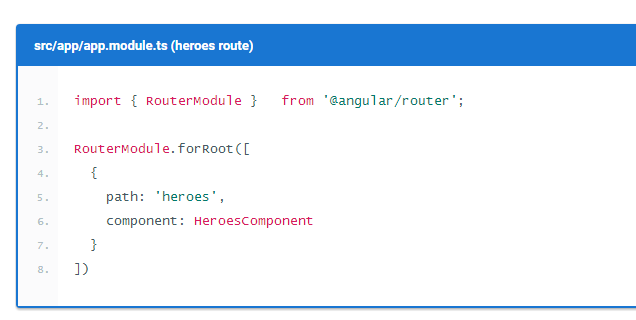
1. Add HeroesComponent to the declarations array of AppModule so Angular recognizes the <my-heroes> tags. 🡺
2. **Add HeroService to the providers array of AppModule because you'll need it in every other view**.
3. The Angular router is an external, optional Angular NgModule called RouterModule. The router is a combination of multiple provided services (RouterModule), multiple directives (RouterOutlet, RouterLink, RouterLinkActive), and a configuration (Routes). You'll configure the routes first.

### **<base href>**

1. Open index.html and ensure there is a <base href="..."> element (or a script that dynamically sets this element) at the top of the <head> section.



Now lets see few points with respect to base href

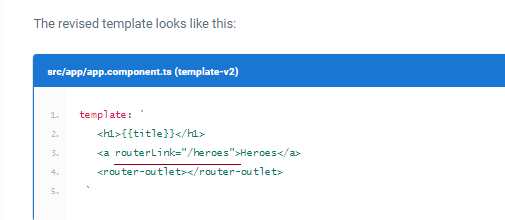
1. Routes tell the router which views to display when a user clicks a link or pastes a URL into the browser address bar.
2. 
3. The forRoot() method supplies the Router service providers and directives needed for routing, and performs the initial navigation based on the current browser URL.Router outlet

### **Router outlet**

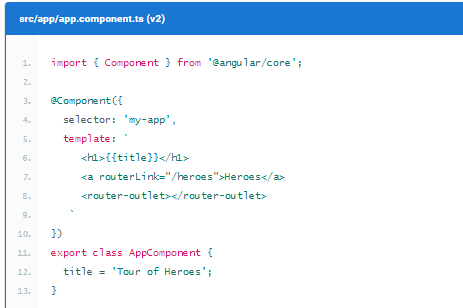
1. However, you have to tell the router where to display the component. To do this, you can add a <router-outlet> element at the end of the template. RouterOutlet is one of the directives provided by the RouterModule. The router displays each component immediately below the <router-outlet> as users navigate through the app.

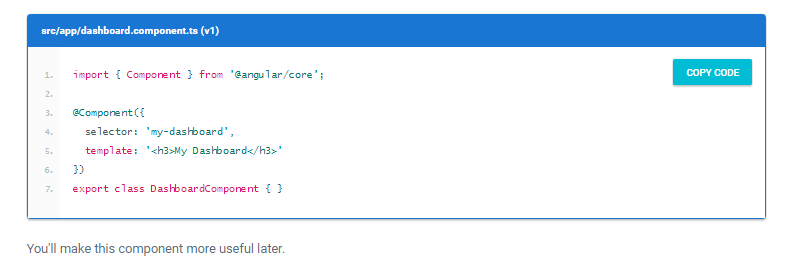
### **Router links**

1. Users shouldn't have to paste a route URL into the address bar. Instead, add an anchor tag to the template that, when clicked, triggers navigation to the HeroesComponent.



1. Now let's see how our appcomponent dashcompo and etc.. looks like

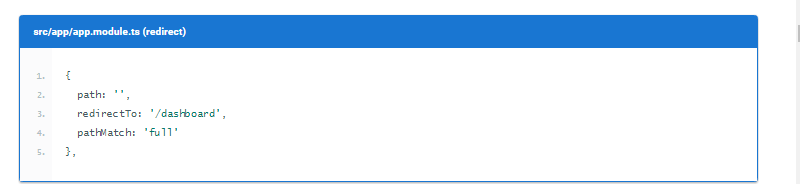




🡺And do the necessary changes in app.module.ts

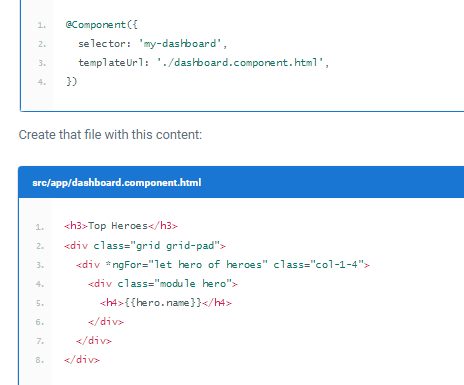
### **Add a redirect route**

1. Currently, the browser launches with / in the address bar. When the app starts, it should show the dashboard and display a /dashboard URL in the browser address bar.To make this happen, use a redirect route. Add the following to the array of route definitions:

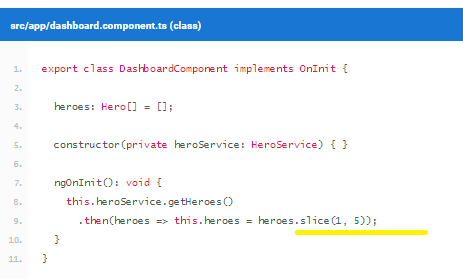


1. The <nav> tags don't do anything yet, but they'll be useful later when you style the links.

Initially our dashboard component was having simple html, the modified looks like

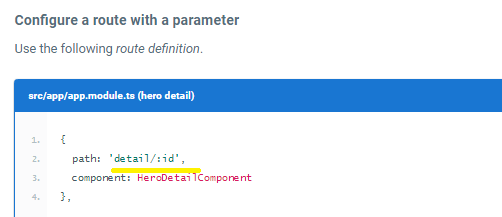


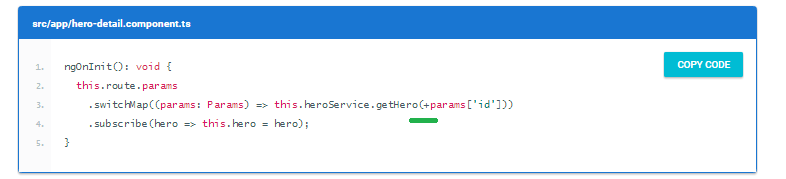
Now let's have a look at the body contents of the dashboard component,



Note🡺 In this dashboard you specify four heroes (2nd, 3rd, 4th, and 5th) with the Array.slice method.

### **Parameterized route**

* Inject the ActivatedRoute, HeroService, and Location services into the constructor, saving their values in private fields:🡺 ms in the hero-detail.component
* 
* The /detail/ part of the URL is constant. The trailing numeric id changes from hero to hero
* The colon (:) in the path indicates that :id is a placeholder for a specific hero id when navigating to the HeroDetailComponent.



inside the ngOnInit() lifecycle hook, use the params Observable to extract the id parameter value from the ActivatedRoute service and use the HeroService to fetch the hero with that id.

The switchMap operator maps the id in the Observable route parameters to a new Observable, the result of the HeroService.getHero() method. If a user re-navigates to this component while a getHero request is still processing, switchMap cancels the old request and then calls HeroService.getHero() again.

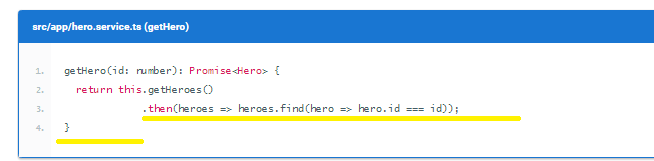
**The hero id is a number. Route parameters are always strings. So the route parameter value is converted to a number with the JavaScript (+) operator.**

### Do you need to unsubscribe?

Ans🡺As described in the [ActivatedRoute: the one-stop-shop for route information](https://angular.io/docs/ts/latest/guide/router.html" \l "activated-route) section of the [Routing & Navigation](https://angular.io/docs/ts/latest/guide/router.html) page, the Router manages the observables it provides and localizes the subscriptions. The subscriptions are cleaned up when the component is destroyed, protecting against memory leaks, so you don't need to unsubscribe from the route params Observable.

### Add HeroService.getHero()

* n the previous code snippet, HeroService doesn't have a getHero() method. To fix this issue, open HeroService and add a getHero() method that filters the heroes list from getHeroes() by id.

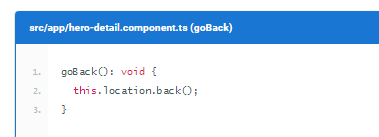


### **Find the way back**

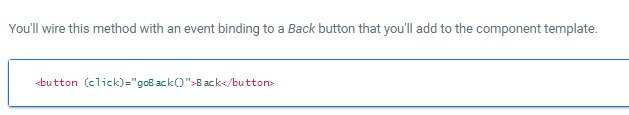
Now let us see the Reason for Location service

* Users have several ways to navigate *to* the HeroDetailComponent.

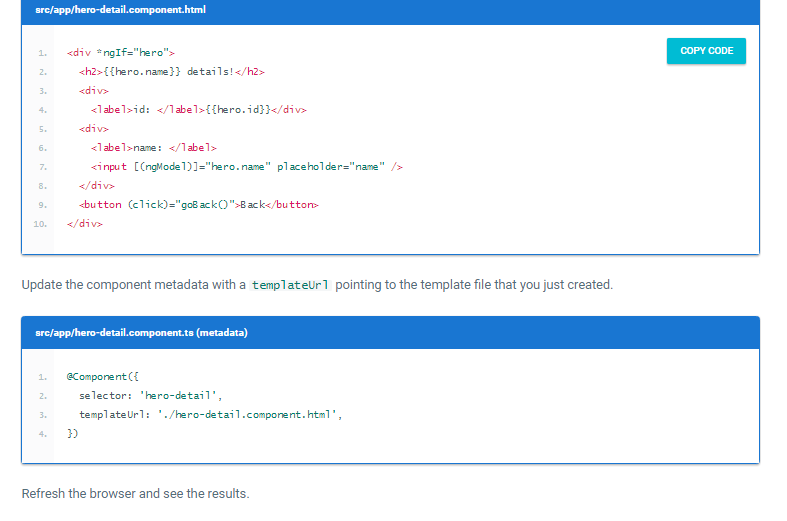
To navigate somewhere else, users can click one of the two links in the AppComponent or click the browser's back button. Now add a third option, a goBack() method that navigates backward one step in the browser's history stack using the Locationservice you injected previously.



* Going back too far could take users out of the app. In a real app, you can prevent this issue with the CanDeactivate guard. Read more on the [CanDeactivate](https://angular.io/docs/ts/latest/api/router/index/CanDeactivate-interface.html) page.

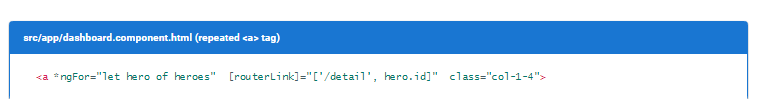


Now lets the Complete code



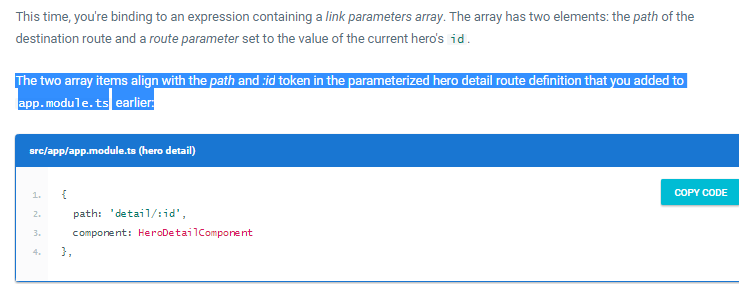
## Select a dashboard hero

* When a user selects a hero in the dashboard, the app should navigate to the HeroDetailComponent to view and edit the selected hero.
* Although the dashboard heroes are presented as button-like blocks, they should behave like anchor tags. When hovering over a hero block, the target URL should display in the browser status bar and the user should be able to copy the link or open the hero detail view in a new tab.
* To achieve this effect, reopen dashboard.component.html and replace the repeated <div \*ngFor...> tags with <a> tags. Change the opening <a> tag to the following:

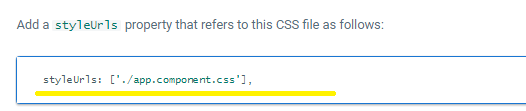


**This time, you're binding to an expression containing a *link parameters array*. The array has two elements: the *path* of the destination route and a *route parameter* set to the value of the current hero's id.**

The two array items align with the *path* and *:id* token in the parameterized hero detail route definition that you added toapp.module.ts earlier:



## Refactor routes to a Routing Module🡺 we will see this later



11/5/2017

**HTTP**

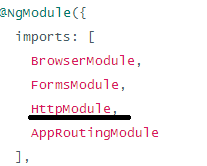
* Here we will see the logic to Convert the service and components to use Angular's HTTP service

**HttpModule**

* The HttpModule is not a core Angular module. HttpModule is Angular's optional approach to web access.
* It exists as a separate add-on module called @angular/http and is shipped in a separate script file as part of the Angular npm package.
* **You're ready to import from @angular/http because systemjs.config configured *SystemJS* to load that library when you need it.**

**Register for HTTP services**

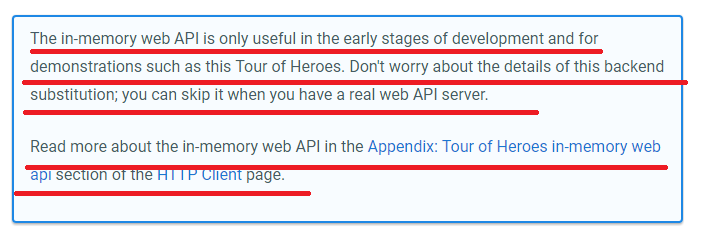
* Our app will depend on the Angular http service, which itself depends on other supporting services.
* The HttpModule from the @angular/http library holds providers for a complete set of HTTP services.
* To allow access to these services from anywhere in the app, add HttpModule to the imports list of the AppModule.



## Simulate the web API

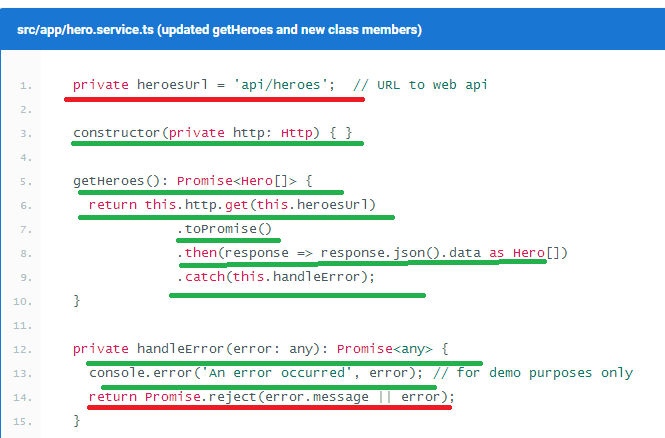
* We recommend registering app-wide services in the root AppModule *providers*.
* Until you have a web server that can handle requests for hero data, the HTTP client will fetch and save data from a mock service, the in-memory web API.
* **Rather than require a real API server, this example simulates communication with the remote server by adding the [InMemoryWebApiModule](https://github.com/angular/in-memory-web-api" \t "_blank" \o "In-memory Web API) to the module imports effectively replacing the Http client's XHR backend service with an in-memory alternative.**



* The forRoot() configuration method takes an InMemoryDataService class that primes the in-memory database. Add the file in-memory-data.service.ts in app with the following content:
* 
* 

## Heroes and HTTP

* W.k.t In the current HeroService implementation, a Promise resolved with mock heroes is returned thus is changed to ultimately fetching heroes with an HTTP client, which must be an asynchronous operation and the latest code changes is

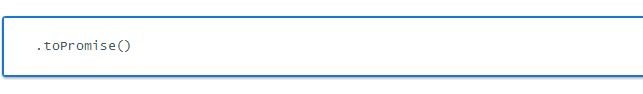


And also the following code changes is done in the above file, that’s is the “import parts”

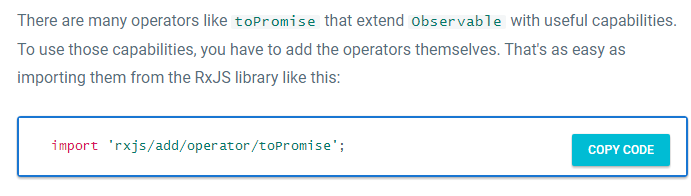
1. import 'rxjs/add/operator/**toPromise'**;

## HTTP Promise

* he Angular http.get returns an RxJS Observable. *Observables* are a powerful way to manage asynchronous data flows.
* For now, you've converted the Observable to a Promise using the toPromise operator.



* The Angular Observable doesn't have a toPromise operator out of the box.



## Extracting the data in the *then* callback

* n the *Promise*'s then() callback, you call the json method of the HTTP Response to extract the data within the response. The response JSON has a single data property, which holds the array of heroes that the caller wants. So you grab that array and return it as the resolved Promise value.

**.then(response => response.json().data as Hero[])**

* The caller is unaware that you fetched the heroes from the (mock) server. It receives a Promise of heroes just as it did before.

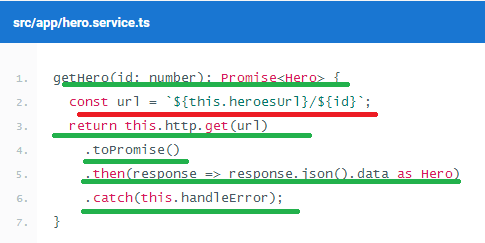
### **Error Handling**

* At the end of getHeroes(), you catch server failures and pass them to an error handler. The code also includes an error to the caller in a rejected promise, so that the caller can display a proper error message to the user.

1. private handleError(error: any): Promise<any> {
2. console.error('An error occurred', error); // for demo purposes only
3. return Promise.reject(error.message || error);
4. }

### **Get hero by id**

* Now lets see the code changes to get the hero based on id by using the Http service
* Most web APIs support a *get-by-id* request in the form **api/hero**/:id (such as **api/hero**/11).
* Now let’s Update the HeroService.getHero() method to make a *get-by-id* request:

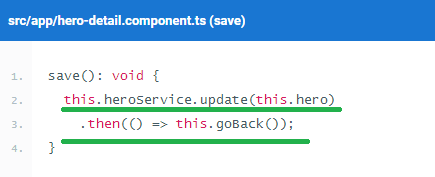


Note heroesUrl is defined in the above screen shot

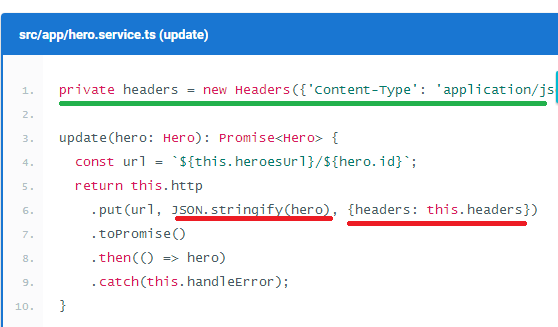
* **The data in the response is a single hero object rather than an array**.

### **Add the ability to save hero details**

* add a save button with a click event binding that invokes a new component method named save()🡺 <button (click)="save()">Save</button>
* Add the following save() method, which persists hero name changes using the hero serviceupdate() method and then navigates back to the previous view.



### **Add a hero service *update()* method**

* **The overall structure of the update() method is similar to that of getHeroes(), but it uses an HTTP put() to persist server-side changes.**
* 

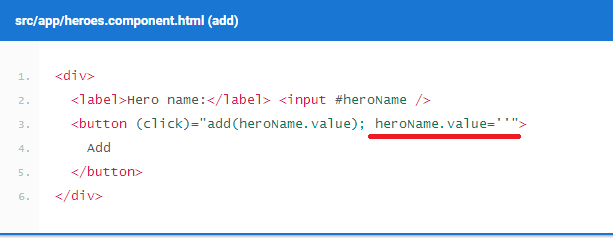
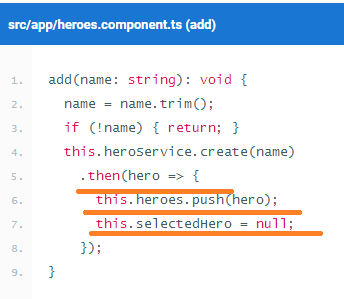
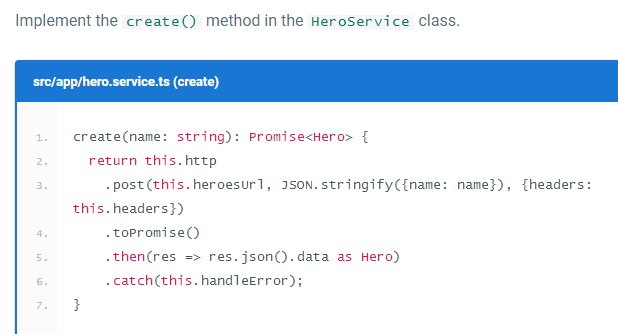
Explanation

🡺To identify which hero the server should update, the hero id is encoded in the URL

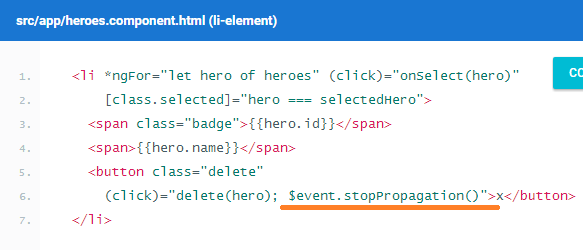
🡺 The put() body is the JSON string encoding of the hero, obtained by calling JSON.stringify

1. 🡺 The body content type (application/json) is identified in the request header. And also the Headers has to be imported from 🡺 import { Headers, Http } from '@angular/http';

### **Add the ability to add heroes**

* To add a hero, the app needs the hero's name. You can use an input element paired with an add button.
* 
* 
* 

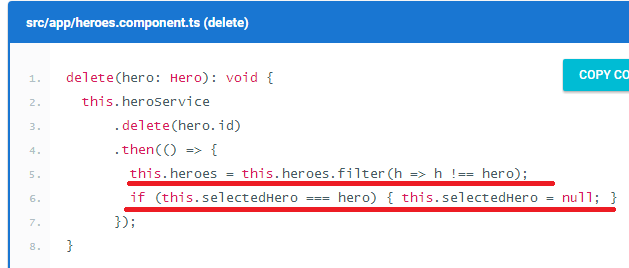
### **Add the ability to delete a hero**

* Each hero in the heroes view should have a delete button.
* Add the following button element to the heroes component HTML, after the hero name in the repeated <li> element.
  + <button class="delete"
  + (click)="delete(hero); $event.stopPropagation()">x</button>
* The heroes.component.html is,
* 

**Explanation**

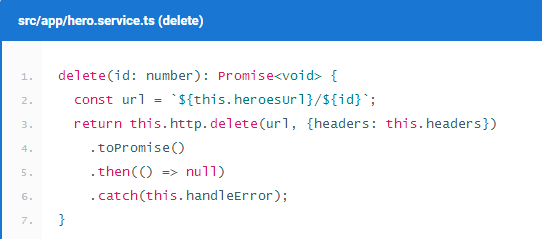
**In addition to calling the component's delete() method, the delete button's click handler code stops the propagation of the click event—you don't want the <li> click handler to be triggered because doing so would select the hero that the user will delete.**

* The logic of the delete() handler is a bit trickier:



Explanation:

Of course you delegate hero deletion to the hero service, but the component is still responsible for updating the display: it removes the deleted hero from the array and resets the selected hero, if necessary.

* NOW LETS SEE THE delete() SYNTAX IN THE SERVICE CLASS
* 

**Observables**

1. Each Http service method returns an Observable of HTTP Response objects.
2. The HeroService converts that Observable into a Promise and returns the promise to the caller.
3. This section shows you how, when, and why to return the Observable .

### Background

1. **An Observable is a stream of events that you can process with array-like operators.**
2. Angular core has basic support for observables. Developers augment that support with operators and extensions from the [RxJS library](http://reactivex.io/rxjs" \t "_blank" \o "RxJS).
3. Converting Observable to a Promise is often a good choice. You typically ask http.get() to fetch a single chunk of data. When you receive the data, you're done. The calling component can easily consume a single result in the form of a Promise.
4. But requests aren't always done only once. You may start one request, cancel it, and make a different request before the server has responded to the first request.

A *request-cancel-new-request* sequence is difficult to implement with Promises, but easy with Observables.

**Add the ability to search by name**

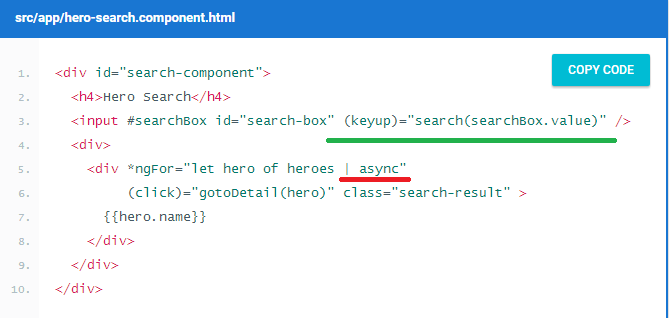
1. As the user types a name into a search box, you'll make repeated HTTP requests for heroes filtered by that name.
2. Now let’s see the code changes
   1. Start by creating HeroSearchService that sends search queries to the server's web API.



* 1. The http.get() call in HeroSearchService is similar to the one in the HeroService, although the URL now has a query string.

**app/heroes/?name=${term}**

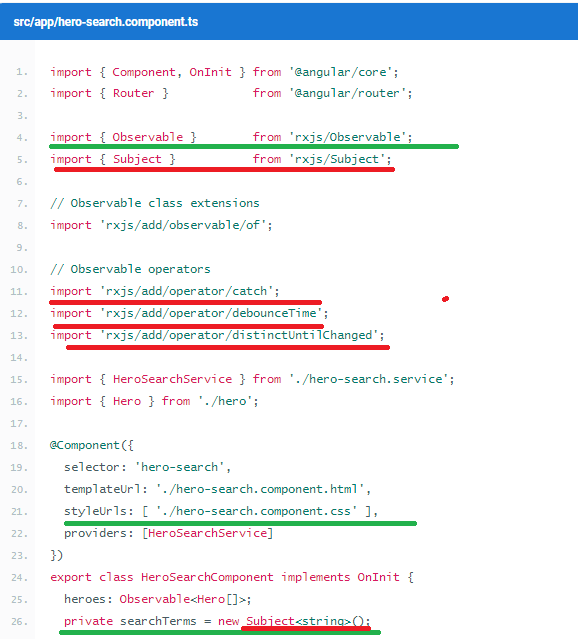
* 1. **More importantly, you no longer call toPromise(). Instead you return the *Observable* from the  http.get(), after chaining it to another RxJS operator, map(), to extract heroes from the response data.**
  2. Now lets see the html file changes,



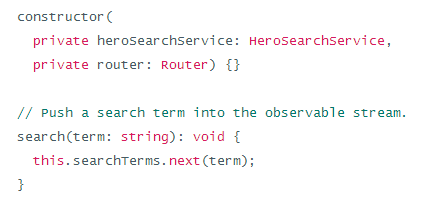
As the user types in the search box, a *keyup* event binding calls the component's search()method with the new search box value. **But as you'll soon see, the heroes property is now an *Observable* of hero arrays, rather than just a hero array. The \*ngFor can't do anything with an Observable until you route it through the async pipe (AsyncPipe). The async pipe subscribes to the Observable and produces the array of heroes to \*ngFor.**

* 1. Now Let’s Create a HeroSearchComponent,

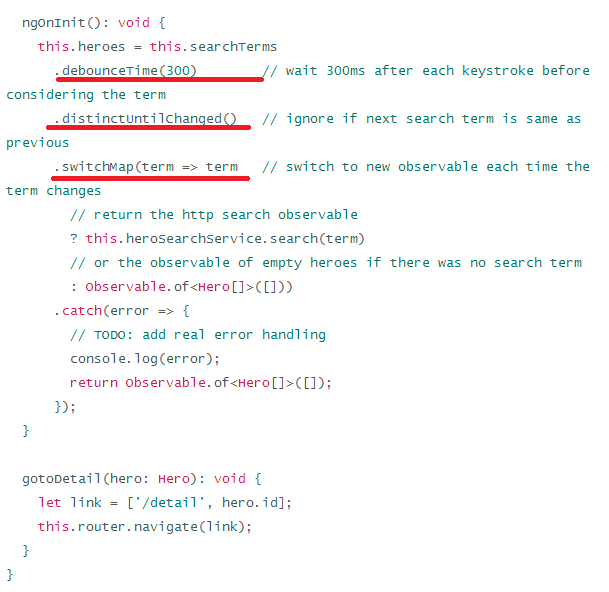
Part1🡺



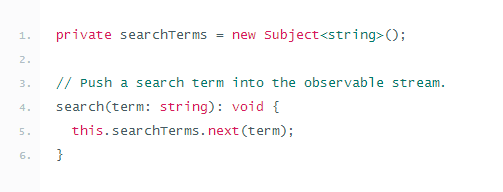
Part2🡺



Part3🡺



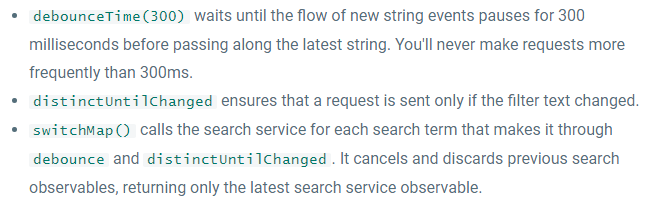
* 1. Now lets focus on the code line no 33 and 34( of Part2 Section)

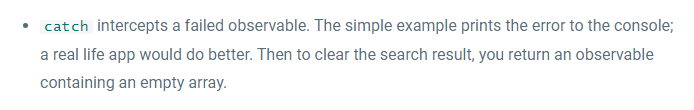


**A Subject is a producer of an *observable* event stream; searchTerms produces an Observable of strings, the filter criteria for the name search.Each call to search() puts a new string into this subject's *observable* stream by calling next().**

#### INITIALIZE THE HEROES PROPERTY (NGONINIT)

* A Subject is also an Observable. You can turn the stream of search terms into a stream of Hero arrays and assign the result to the heroes property.
* Now lets see the purpose of, debounceTime(300), distinctUntilChanged(),switchMap operators purpose,
* Passing every user keystroke directly to the HeroSearchService would create an excessive amount of HTTP requests, taxing server resources and burning through the cellular network data plan
* Instead, you can chain Observable operators that reduce the request flow to the string Observable. You'll make fewer calls to the HeroSearchService and still get timely results.
* Here's how we do that:





* With the [switchMap operator](http://www.learnrxjs.io/operators/transformation/switchmap.html) (formerly known as flatMapLatest), every qualifying key event can trigger an http() method call. Even with a 300ms pause between requests, you could have multiple HTTP requests in flight and they may not return in the order sent.
* If the search text is empty, the http() method call is also short circuited and an observable containing an empty array is returned.

### ***Import RxJS operators***

* Most RxJS operators are not included in Angular's base Observable implementation
* **The base implementation includes only what Angular itself requires.**
* When you need more RxJS features, extend Observable by *importing* the libraries in which they are defined.

# **ATTRIBUTE DIRECTIVES**

1. Attribute directives attach behavior to elements.
2. An **Attribute** directive changes the appearance or behavior of a DOM element.
3. There are three kinds of directives in Angular:

* Components—directives with a template.
* Structural directives—change the DOM layout by adding and removing DOM elements🡺 Structural Directives change the structure of the view. Two examples are [NgFor](https://angular.io/docs/ts/latest/guide/template-syntax.html" \l "ngFor) and [NgIf](https://angular.io/docs/ts/latest/guide/template-syntax.html" \l "ngIf).
* Attribute directives—change the appearance or behavior of an element, component, or another directive🡺 Attribute directives are used as attributes of elements. The built-in [NgStyle](https://angular.io/docs/ts/latest/guide/template-syntax.html" \l "ngStyle) directive can change several element styles at the same time.

## Build a simple attribute directive

1. An attribute directive minimally requires building a controller class annotated with@Directive, which specifies the selector that identifies the attribute. The controller class implements the desired directive behavior.

Example:

Write the directive code



Explanation

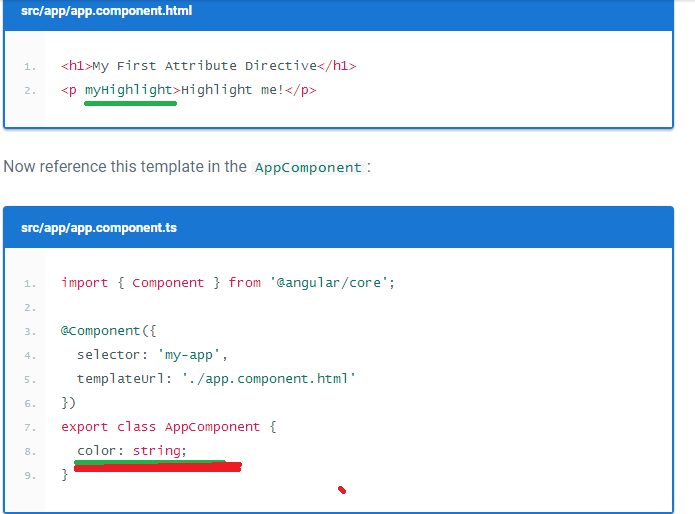
* Directive provides the functionality of the @Directive decorator.
* ElementRef [injects](https://angular.io/docs/ts/latest/guide/dependency-injection.html) into the directive's constructor so the code can access the DOM element.
* Input allows data to flow from the binding expression into the directive.

**After the @Directive metadata comes the directive's controller class, called HighlightDirective, which contains the logic for the directive. Exporting HighlightDirective makes it accessible to other components.**

**ElementRef is a service that grants direct access to the DOM element through its nativeElement property.**

**Apply the attribute directive**

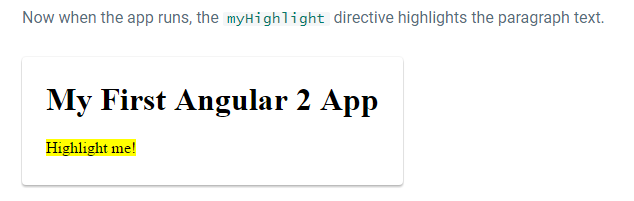
1. To use the new HighlightDirective, create a template that applies the directive as an attribute to a paragraph (<p>) element. In Angular terms, **the <p> element is the attribute host.**



After specifying HighlightDirective in the declarations array, Angular knows it can apply the directive to components declared in this module.

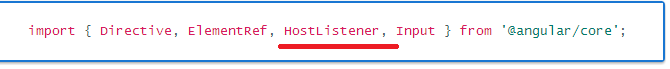


Explanation🡺 add an import statement to fetch the Highlight directive and add that class to the declarations NgModule metadata. This way Angular recognizes the directive when it encounters myHighlight in the template.

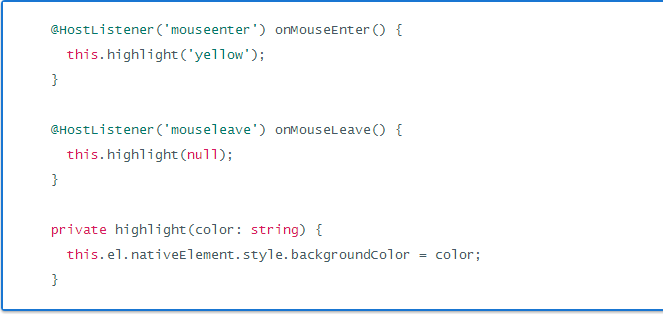


**Respond to user-initiated events**

1. Currently, myHighlight simply sets an element color. The directive could be more dynamic. It could detect when the user mouse’s into or out of the element and respond by setting or clearing the highlight color.
2. Begin by adding HostListener to the list of imported symbols



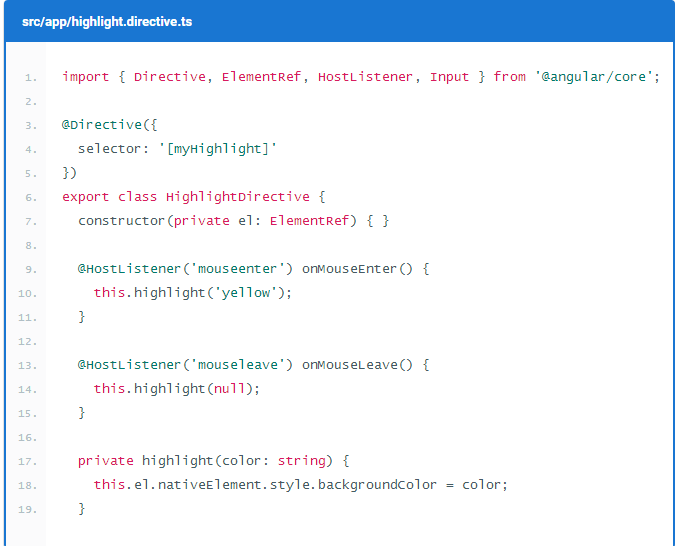
1. Then add two event handlers that respond when the mouse enters or leaves, each adorned by the HostListener decorator.



Explanation:

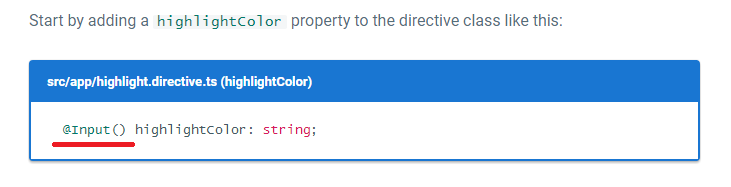
**The @HostListener decorator lets you subscribe to events of the DOM element** that hosts an attribute directive, the <p> in this case.

1. Here's the updated directive in full:



Output🡺 See the Tutorial

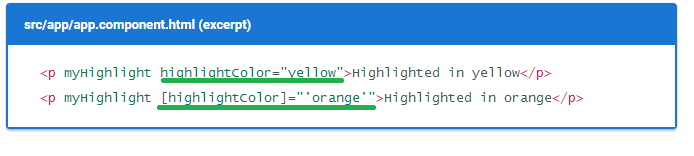
**Pass values into the directive with an *@Input* data binding**

1. Currently the highlight color is hard-coded within the directive. That's inflexible. In this section, you give the developer the power to set the highlight color while applying the directive.
2. 

**Binding to an *@Input* property**

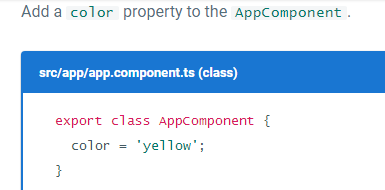
1. **Notice the @Input decorator. It adds metadata to the class that makes the directive's highlightColor property available for binding.**

It's called an input property because data flows from the binding expression into the directive. Without that input metadata

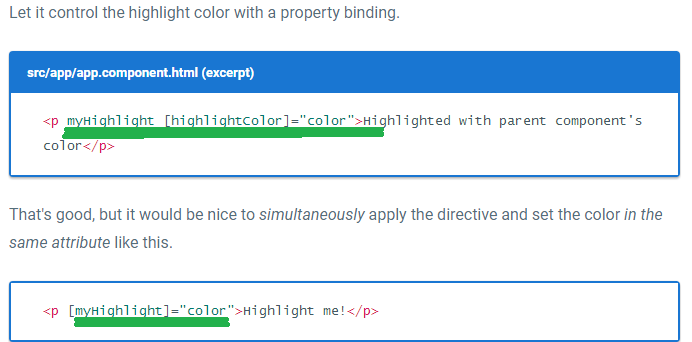


1. Ms 🡺In the above example the value for highlightColor flows from view to binding, now as a next step let us have the property binding as shown below

Step 1🡺



Step 2🡺



Note 🡺 You'll have to rename the directive's highlightColor property to myHighlight because that's now the color property binding name **🡺 @Input() myHighlight: string;**

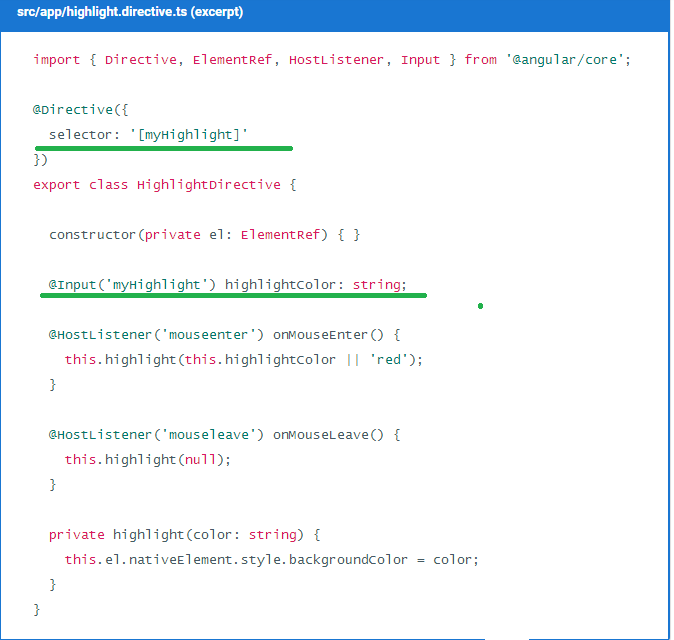
This is disagreeable. The word, myHighlight, is a terrible property name and it doesn't convey the property's intent

**Bind to an *@Input* alias**

1. Fortunately you can name the directive property whatever you want and **alias it** for binding purposes.



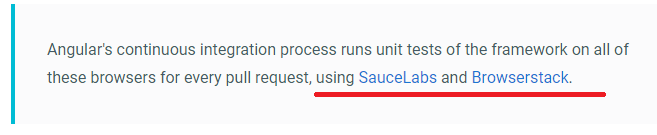
1. Here's the latest version of the directive class.



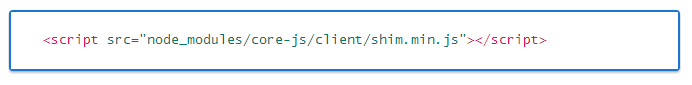
15/5/2017

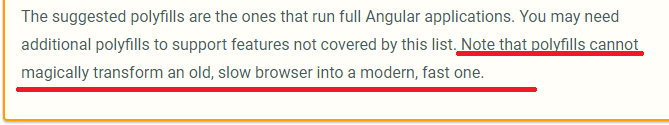
BROWSER SUPPORT

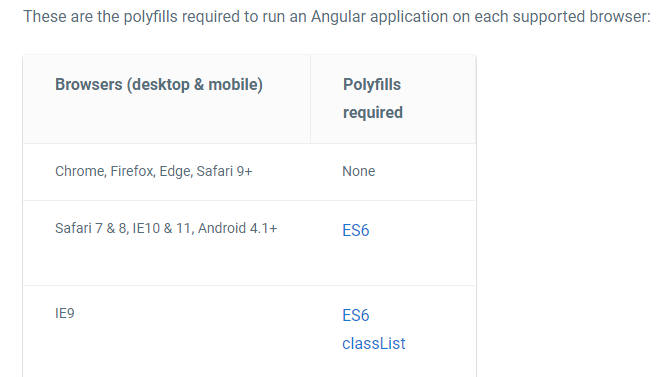
Browser support and **polyfills** guide.

1. Angular supports most recent browsers.
2. 

## Polyfills

1. Angular is built on the latest standards of the web platform. Targeting such a wide range of browsers is challenging because they do not support all features of modern browsers.
2. You can compensate by loading polyfill scripts ("polyfills") on the host web page (index.html) that implement missing features in JavaScr
3. A particular browser may require at least one polyfill to run any Angular application. You may need additional polyfills for specific features.
4. The tables below can help you determine which polyfills to load, depending on the browsers you target and the features you use.

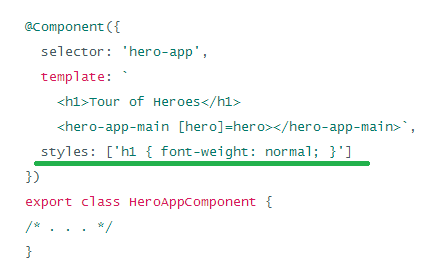


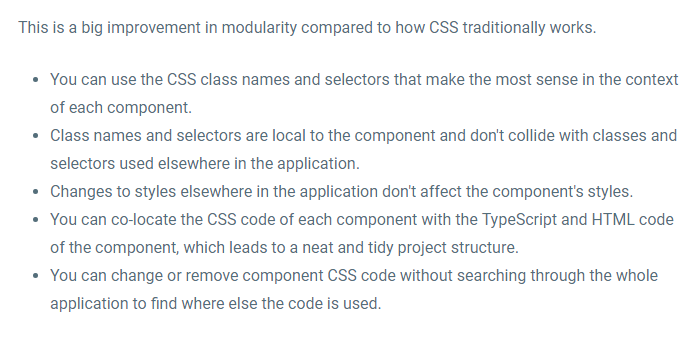
1. 
2. FOR MORE NOTES ON POLYFILLS SEE THE DOCUMENT🡺 I HAD NOT COVERED THIS TOPIC FULLY

# **COMPONENT STYLES**

1. Angular applications are styled with standard CSS. That means you can apply everything you know about CSS stylesheets, selectors, rules, and media queries directly to Angular applications.
2. Additionally, Angular can bundle component styles with components, enabling a more modular design than regular stylesheets.

**Using component styles**

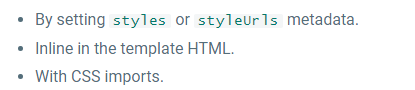
1. For every Angular component you write, you may define not only an HTML template, but also the CSS styles that go with that template, specifying any selectors, rules, and media queries that you need.
2. One way to do this is to set the styles property in the component metadata. The stylesproperty takes an array of strings that contain CSS = code.  
3. The selectors you put into a component's styles apply only within the template of that component. The h1 selector in the preceding example applies only to the <h1> tag in the template of HeroAppComponent. Any <h1> elements elsewhere in the application are unaffected.
4. Advantages of Component Styles,

**Special selectors**

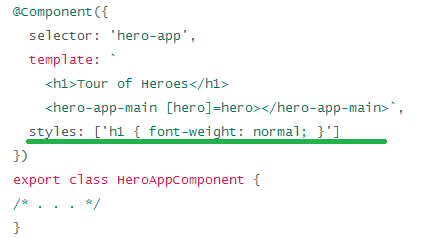
### Component styles have a few special selectors from the world of shadow DOM style scoping. The following sections describe these selectors**🡺 host: , host-context, /deep/**

**Loading styles into components**

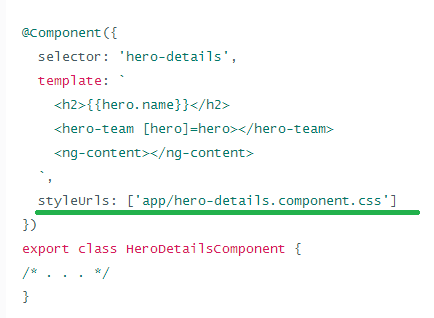
1. There are several ways to add styles to a component:



### **Styles in metadata**

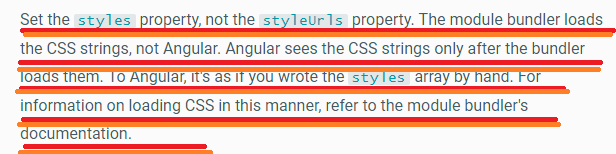
1. 

### **Style URLs in metadata**

1. 

NOTE🡺 **index.html web page that hosts the application**

1. If you use module bundlers like Webpack, you can also use the styles attribute to load styles from external files at build time. You could write🡺 

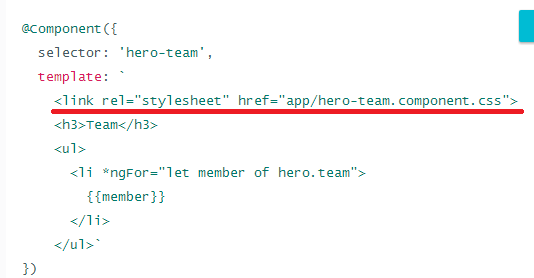


### **Template inline styles**

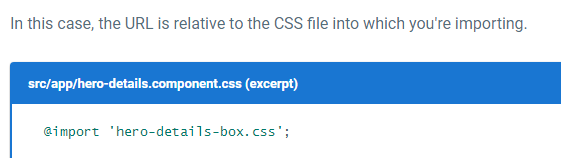
1. You can embed styles directly into the HTML template by putting them inside <style> tags.

### **Template link tags**

1. You can also embed <link> tags into the component's HTML template.
2. **As with styleUrls, the link tag's href URL is relative to the application root, not the component file.**



### **CSS @imports**

1. You can also import CSS files into the CSS files using the standard CSS @import rule. For details, see [@import](https://developer.mozilla.org/en/docs/Web/CSS/@import) on the [MDN](https://developer.mozilla.org/) site.
2. 

**Controlling view encapsulation: native, emulated, and none**

1. As discussed earlier, component CSS styles are encapsulated into the component's view and don't affect the rest of the application.

THIS PART WE WILL SEE LATER

### **DEPLOYMENT**

**We will see it later**

1. This page describes tools and techniques for deploy and optimize your Angular application.

**Overview**

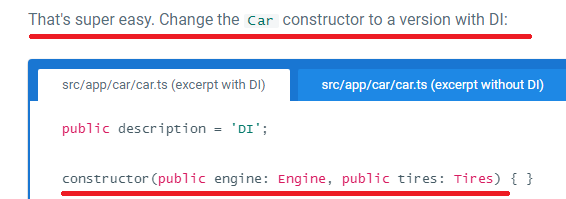
1. This guide describes techniques for preparing and deploying an Angular application to a server running remotely. The techniques progress from easy but suboptimal to more optimal and more involved.
2. MS 🡺Now let’s see the Ways🡺

* The [simple way](https://angular.io/docs/ts/latest/guide/deployment.html#dev-deploy) is to copy the development environment to the server.
* [Ahead of Time compilation (AOT)](https://angular.io/docs/ts/latest/guide/deployment.html#aot) is the first of [several optimization strategies](https://angular.io/docs/ts/latest/guide/deployment.html#optimize). You'll also want to read the [detailed instructions in the AOT Cookbook](https://angular.io/docs/ts/latest/cookbook/aot-compiler.html).
* [Webpack](https://angular.io/docs/ts/latest/guide/deployment.html#webpack) is a popular general purpose packaging tool with a rich ecosystem, including plugins for AOT. The Angular [webpack guide](https://angular.io/docs/ts/latest/guide/webpack.html" \o "Webpack: an introduction) can get you started and this page provides additional optimization advice, but you'll probably have to learn more about webpack on your own.
* The [Angular configuration](https://angular.io/docs/ts/latest/guide/deployment.html#angular-configuration) section calls attention to specific client application changes that could improve performance.
* The [Server configuration](https://angular.io/docs/ts/latest/guide/deployment.html#server-configuration) section describes server-side changes that may be necessary, no matter how you deploy the application.

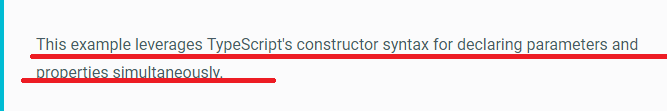
# HIERARCHICAL DEPENDENCY INJECTORS

1. **Dependency injection** is an important application design pattern. Angular has its own dependency injection framework, and you really can't build an Angular application without it. It's used so widely that almost everyone just calls it DI.
2. **MS Reason For using the DI is that the code breaks if the Constructor syntax is changed,**

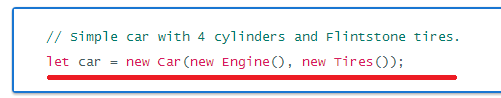
**Thus to make our app** more robust, flexible, and testable



See what happened? The definition of the dependencies are now in the constructor. The Carclass no longer creates an engine or tires. It just consumes them.

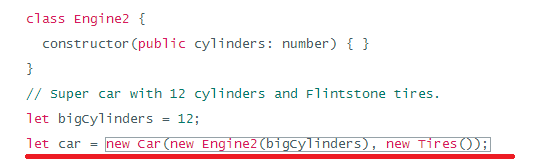


1. Example MS🡺

Now let’s see how to create a Car object🡺

Now the problem is, what if the User Who needs only the Car Class Method, he needs to create the object of Car class by passing the Engine and Tire object

Now let’s assume that the Constructor of Engine class is changed then the User who wants to use only the Car object also needed to do the changes w.r.t Engine also as shown in below



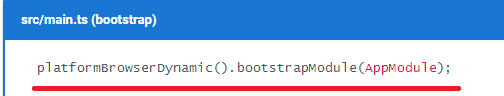
Which is the actually disadvantage

**Angular dependency injection**

1. A service is nothing more than a class in Angular. It remains nothing more than a class until you register it with an Angular injector.

**Configuring the injector**

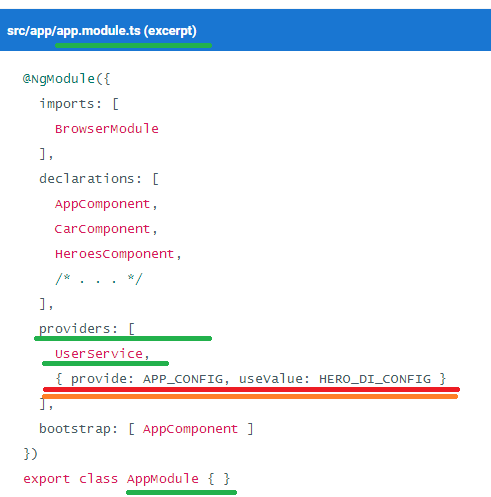
1. You don't have to create an Angular injector. Angular creates an application-wide injector for you during the bootstrap process.



1. You can either register a provider within an [NgModule](https://angular.io/docs/ts/latest/guide/ngmodule.html) or in application components.

**Registering providers in an NgModule**

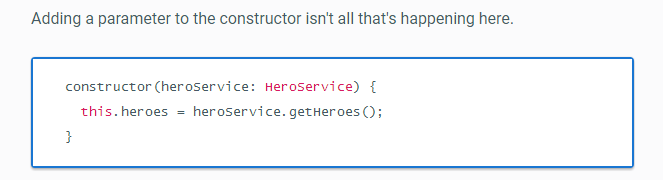
Here's the AppModule that registers two providers, UserService and an **APP\_CONFIGprovider**, in its providers array.

1. 

**Registering providers in a component**

1. Because the HeroService is used *only* within the HeroesComponent and its subcomponents, the top-level HeroesComponent is the ideal place to register it.
2. 

**When to use *NgModule* versus an application component**

1. On the one hand, a provider in an NgModule is registered in the root injector. That means that every provider registered within an NgModule will be accessible in the *entire application*. On the other hand, a provider registered in an application component is available only on that component and all its children.
2. **Here, the APP\_CONFIG service needs to be available all across the application, so it's registered in the AppModule @NgModule providers array. But since the HeroService is only used within the *Heroes* feature area and nowhere else, it makes sense to register it in the HeroesComponent.**
3. 

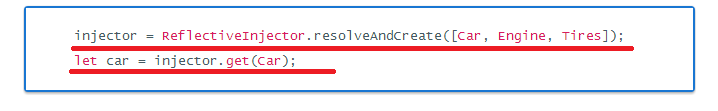
Explanation:

Note that the constructor parameter has the type HeroService, and that the HeroListComponent class has an @Component decorator (scroll up to confirm that fact). Also recall that the parent component (HeroesComponent) has providers information for HeroService.

The constructor parameter type, the @Component decorator, and the parent's providers information combine to tell the Angular injector to inject an instance of HeroService whenever it creates a new HeroListComponent.

**Implicit injector creation**

1. You saw how to use an injector to create a new Car earlier in this guide. You *could* create such an injector explicitly:



You won't find code like that in the Tour of Heroes or any of the other documentation samples. You *could* write code that [explicitly creates an injector](https://angular.io/docs/ts/latest/guide/dependency-injection.html#explicit-injector) if you *had* to, but it's not always the best choice. Angular takes care of creating and calling injectors when it creates components for you—whether through HTML markup, as in <hero-list></hero-list>, or after navigating to a component with the [router](https://angular.io/docs/ts/latest/guide/router.html). If you let Angular do its job, you'll enjoy the benefits of automated dependency injection.

**Singleton services**

1. Dependencies are singletons within the scope of an injector🡺 In this guide's example, a single HeroService instance is shared among the HeroesComponent and its HeroListComponent children.
2. **However, Angular DI is a hierarchical injection system, which means that nested injectors can create their own service instances. For more information, see**[**Hierarchical Injectors**](https://angular.io/docs/ts/latest/guide/hierarchical-dependency-injection.html)**.**

**When the service needs a service**

1. The HeroService is very simple. It doesn't have any dependencies of its own.
2. What if it had a dependency?

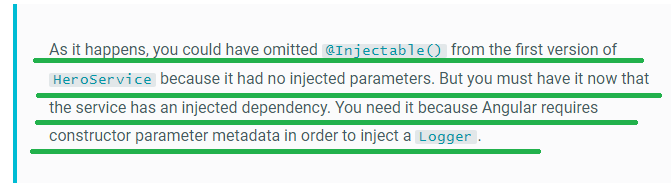
What if it reported its activities through a **logging service?** You'd apply the same constructor injection pattern, adding a constructor that takes a Logger parameter.

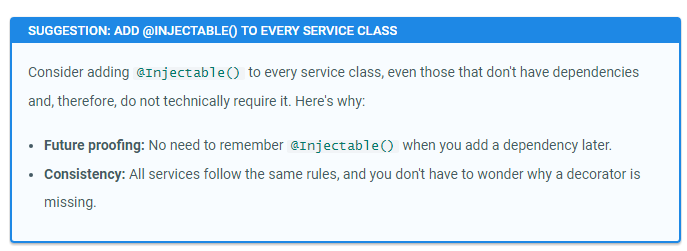


The constructor now asks for an injected instance of a Logger and stores it in a private property called logger. You call that property within the getHeroes() method when anyone asks for heroes.

**Why *@Injectable()*?**

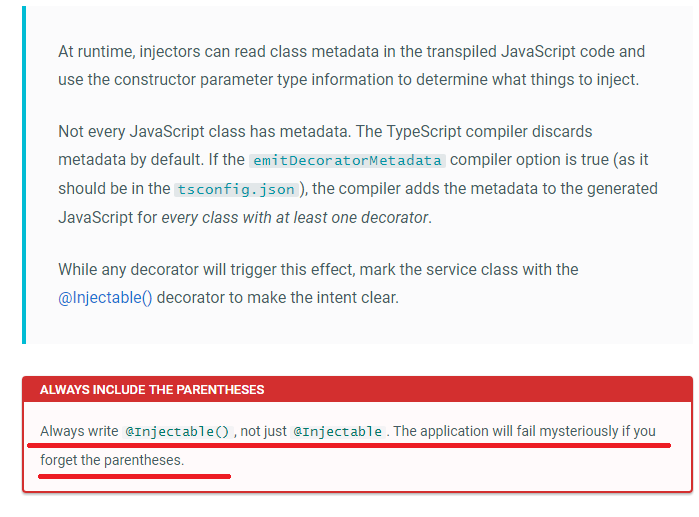
1. [**@Injectable()**](https://angular.io/docs/ts/latest/api/core/index/Injectable-decorator.html) marks a class as available to an injector for instantiation. Generally speaking, an injector reports an error when trying to instantiate a class that is not marked as@Injectable().





**Injectors are also responsible for instantiating components like HeroesComponent. So why doesn’t HeroesComponent have @Injectable()?**

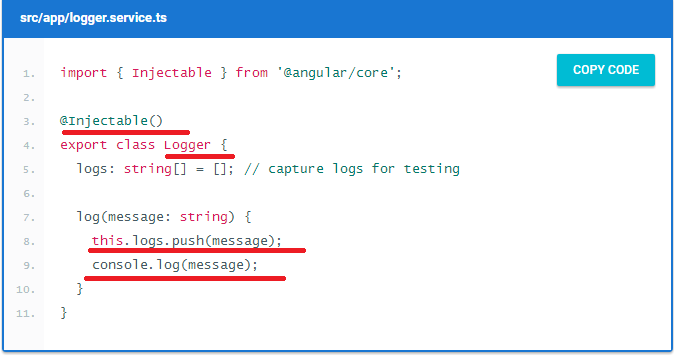
You *can* add it if you really want to. It isn't necessary because the HeroesComponent is already marked with @Component, and this decorator class is a subtype of [@Injectable()](https://angular.io/docs/ts/latest/api/core/index/Injectable-decorator.html). It is in fact @Injectable() decorators that identify a class as a target for instantiation by an injector.

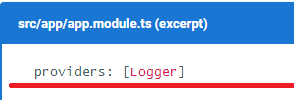


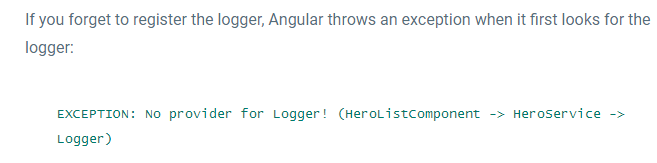
## NOW LETS CREATE AN APPLICATION WITH THE LOGGER SERVICE🡺 Creating and registering a logger service

1. Inject a logger into HeroService in two steps:

* Create the logger service.
* Register it with the application.

1. 
2. You're likely to need the same logger service everywhere in your application, so put it in the project's app folder and register it in the providers array of the application module, AppModule.



1. 

**Injector providers**

1. A provider provides the concrete, runtime version of a dependency value. The injector relies on **providers** to create instances of the services that the injector injects into components and other services🡺 HERE FEW MORE TEXT IS GIVEN SEE THIS LATER

**The *Provider* class and *provide* object literal**

1. WKT🡺 providers: [Logger]
2. This is actually a shorthand expression for a provider registration using a provider object literal with two properties:

[

{

provide: Logger, useClass: Logger

}

]

The first is the [token](https://angular.io/docs/ts/latest/guide/dependency-injection.html#token) that serves as the key for both **locating a dependency value** and registering the provider. **The second is a provider definition object, which you can think of as a recipe for creating the dependency value**. There are many ways to create dependency values just as there are many ways to write a recipe.

### **Alternative class providers**

1. The following code tells the injector to return a BetterLogger when something asks for the Logger.

[

{

provide: Logger,

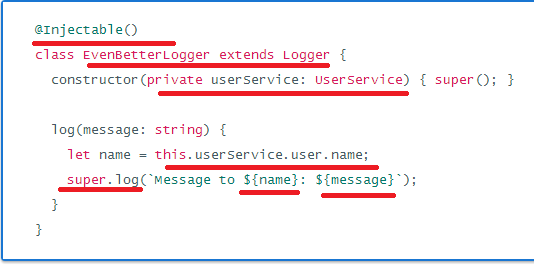
useClass: BetterLogger

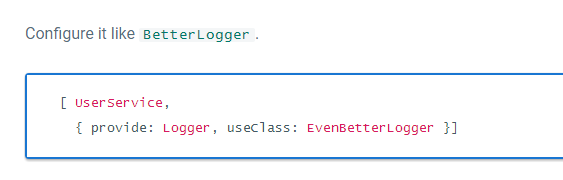
}

]

### **Class provider with dependencies🡺 ms here we see an example for 28th point**

1. Maybe an EvenBetterLogger could display the user name in the log message. This logger gets the user from the injected UserService, which is also injected at the application level.

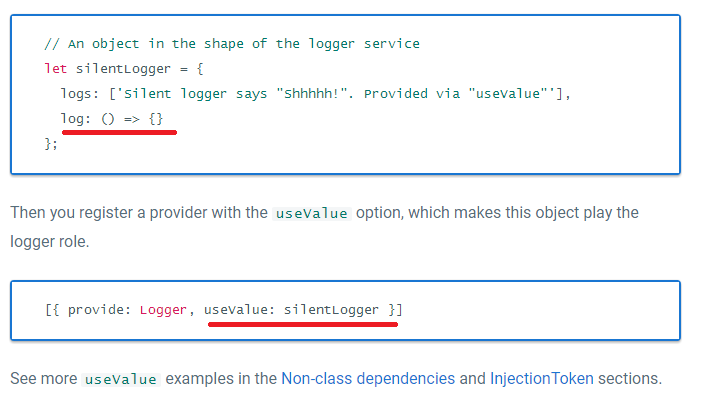




### **Aliased class providers**🡺 LATER

### **Value providers**

1. Sometimes it's easier to provide a ready-made object rather than ask the injector to create it from a class.



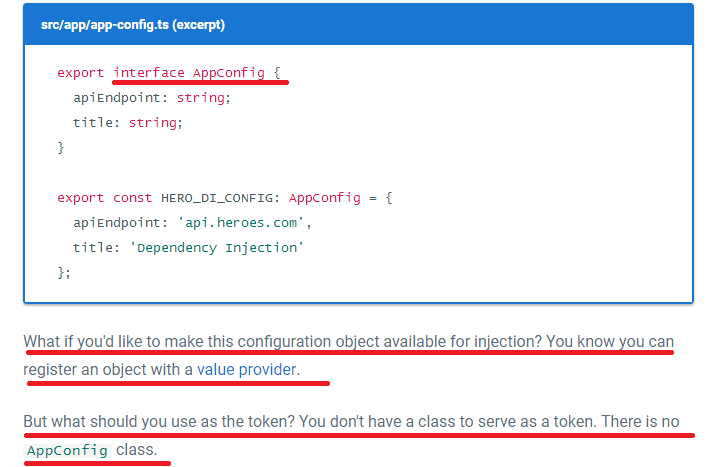
### **Factory providers**

1. Sometimes you need to create the dependent value dynamically, based on information you won't have until the last possible moment. Maybe the information changes repeatedly in the course of the browser session. Suppose also that the injectable service has no independent access to the source of this information🡺 This situation calls for a **factory provider**.
2. For this scenario let’s have the requirement as follows🡺 To illustrate the point, add a new business requirement: the HeroService must hide *secret*heroes from normal users.  Only authorized users should see secret heroes.

**NEEDED MORE INFORMATION ON THIS Concept 🡺 so let’s cover this part later**

**Dependency injection tokens**

### **Non-class dependencies**

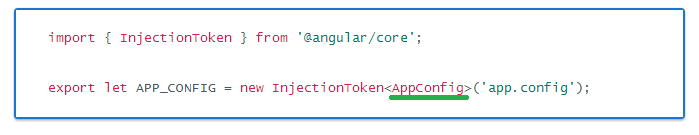
1. What if the dependency value isn't a class? Sometimes the thing you want to inject is a string, function, or object.



1. As a solution let’s see the below topic,

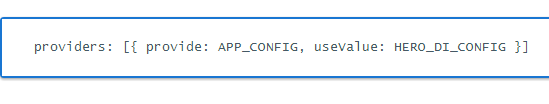
***InjectionToken***

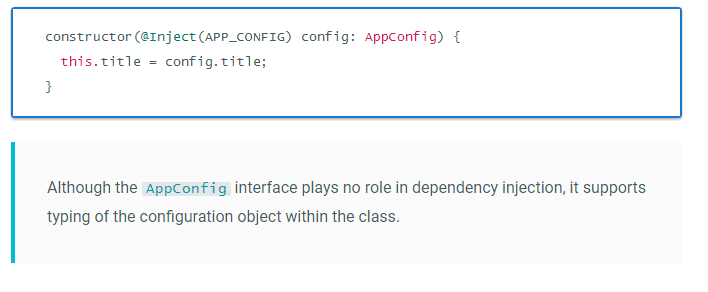
1. One solution to choosing a provider token for non-class dependencies is to define and use an **[InjectionToken](https://angular.io/docs/ts/latest/api/core/index/InjectionToken-class.html)**. The definition of such a token looks like this:



The type parameter, while optional, conveys the dependency's type to developers and tooling. The token description is another developer aid.

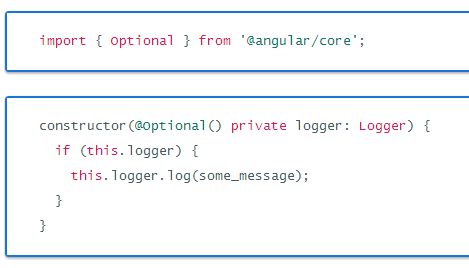
1. Register the dependency provider using the InjectionToken object:



1. Now you can inject the configuration object into any constructor that needs it, with the help of an @Inject decorator:

**Optional dependencies**

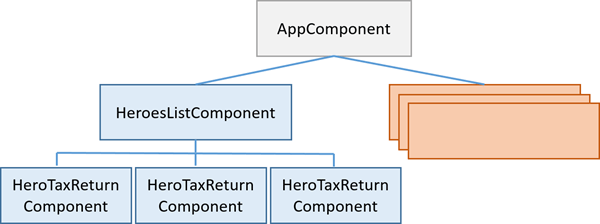
1. EXPLANATION WAS NOT CLEAR
2. You can tell Angular that the dependency is optional by annotating the constructor argument with @Optional():



Note🡺 When using @Optional(), your code must be prepared for a null value. If you don't register a logger somewhere up the line, the injector will set the value of logger to null.

# **HIERARCHICAL DEPENDENCY INJECTORS**

**The injector tree**



**Injector bubbling**

1. When a component requests a dependency, Angular tries to satisfy that dependency with a provider registered in that component's own injector. If the component's injector lacks the provider, it passes the request up to its parent component's injector. If that injector can't satisfy the request, it passes it along to its parent injector. The requests keep bubbling up until Angular finds an injector that can handle the request or runs out of ancestor injectors. If it runs out of ancestors, Angular throws an error.

**Re-providing a service at different levels🡺Later**

**Component injectors🡺Later**

# **REACTIVE FORMS**

# **Introduction to Reactive Forms**

1. Angular offers two form-building technologies: reactive forms and template-driven forms
2. The two technologies belong to the @angular/forms library and share a common set of form control classes.
3. But they diverge markedly in philosophy, programming style, and technique. They even have their own modules: the ReactiveFormsModule and the FormsModule.

### Reactive forms

1. You create and manipulate form control objects directly in the component class
2. As the component class has immediate access to both the data model and the form control structure, you can push data model values into the form controls and pull user-changed values back out. The component can observe changes in form control state and react to those changes.
3. One advantage of working with form control objects directly is that value and validity updates are [always synchronous and under your control](https://angular.io/docs/ts/latest/guide/reactive-forms.html#async-vs-sync).You won't encounter the timing issues that sometimes plague a template-driven form and reactive forms can be easier to unit test.

Template-driven forms

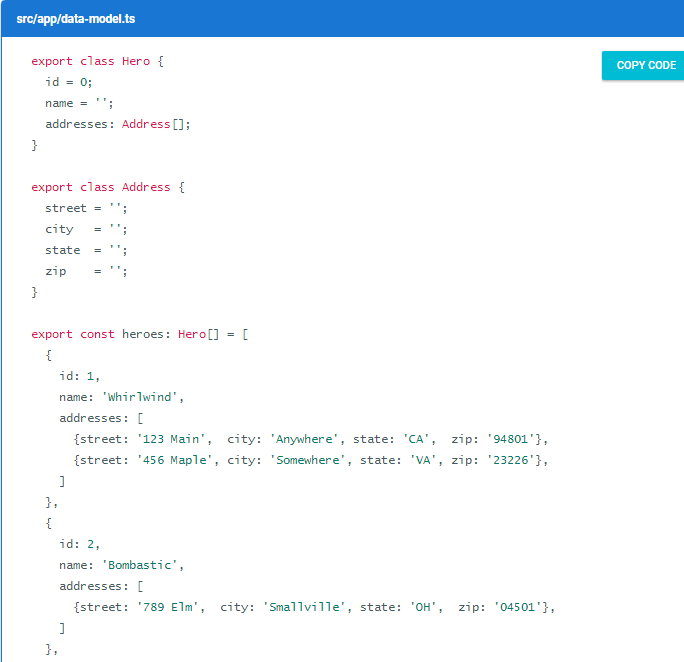
1. You place HTML form controls (such as <input> and <select>) in the component template and bind them to *data model* properties in the component, using directives like ngModel
2. You don't push and pull data values. Angular handles that for you with ngModel. Angular updates the mutable *data model* with user changes as they happen.
3. **While this means less code in the component class,**[**template-driven forms are asynchronous**](https://angular.io/docs/ts/latest/guide/reactive-forms.html#async-vs-sync)**which may complicate development in more advanced scenarios**

### Async vs. sync

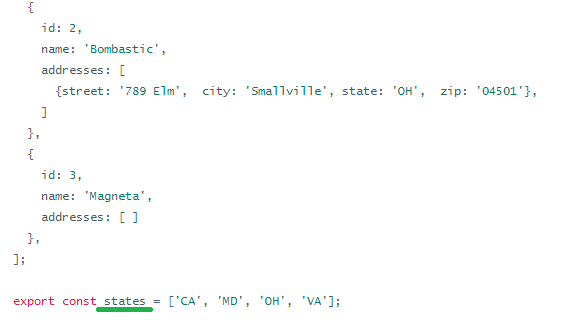
1. Reactive forms are synchronous. Template-driven forms are asynchronous. It's a difference that matters.
2. The asynchrony of template-driven forms also complicates unit testing. You must wrap your test block in async() or fakeAsync() to avoid looking for values in the form that aren't there yet. With reactive forms, everything is available when you expect it to be.
3. Now LETS SEE AN EXAMPLE FOR REACTIVE FORM

Step1🡺 Create a Model class

Part 1

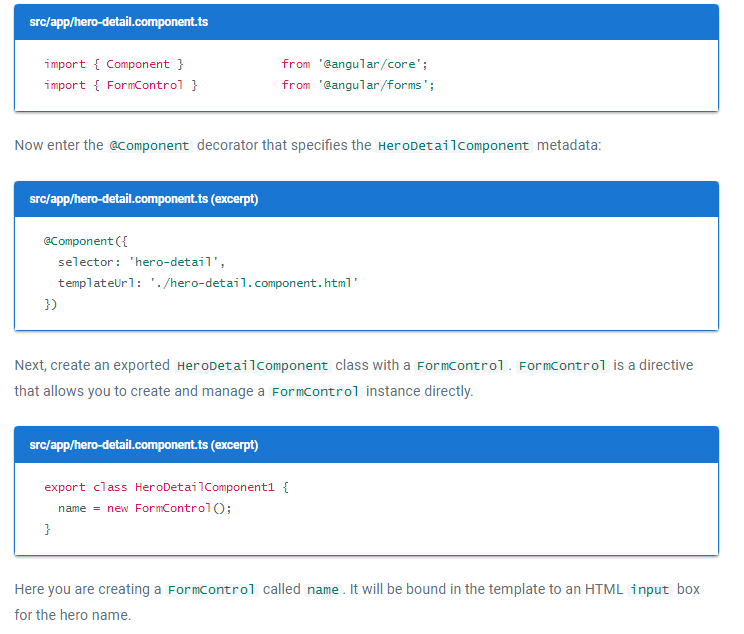


Part 2:



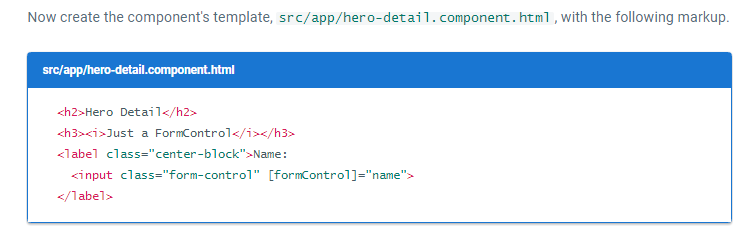
Step 2: Create a reactive forms component

Make a new file called hero-detail.component.ts in the app directory and import these symbols:



**NOTE🡺A FormControl constructor accepts three, optional arguments: the initial data value, an array of validators, and an array of async validators.**

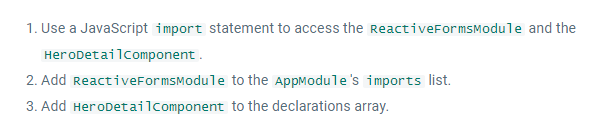
## Step 3🡺 Create the template

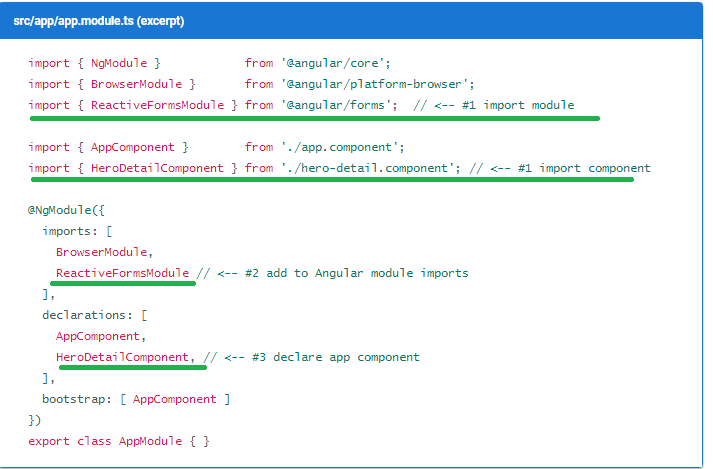


Explanation🡺

To let Angular know that this is the input that you want to associate to the name FormControl in the class, you need [formControl]="name" in the template on the <input>.

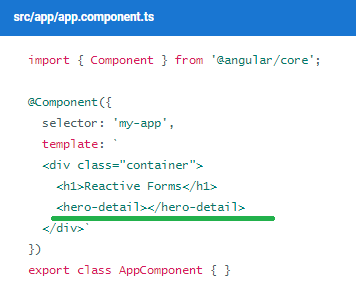
## Step4🡺 Import the *ReactiveFormsModule*

1. The HeroDetailComponent template uses formControlName directive from the ReactiveFormsModule
2. do the following three things in app.module.ts:

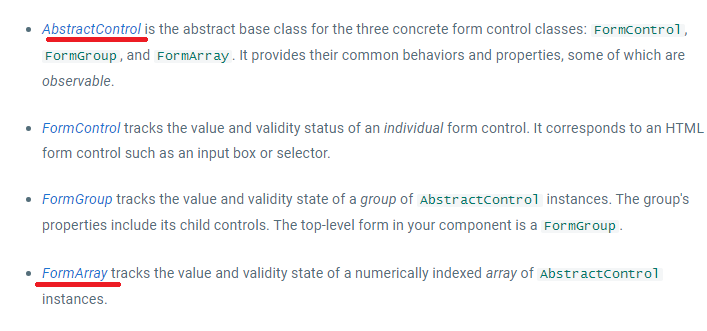


**Step5**🡺**Display the *HeroDetailComponent***

1. Revise the AppComponent template so it displays the HeroDetailComponent.



### **Essential form classes**

1. 

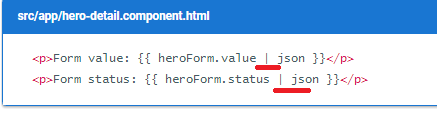
**Add a FormGroup**

1. Usually, if you have multiple *FormControls*, you'll want to register them within a parent FormGroup. This is simple to do. To add a FormGroup, add it to the imports section of hero-detail.component.ts:





Now following code is the interpolation for 🡺To see the form model, add the following line after the closing form tag in the hero-detail.component.html:



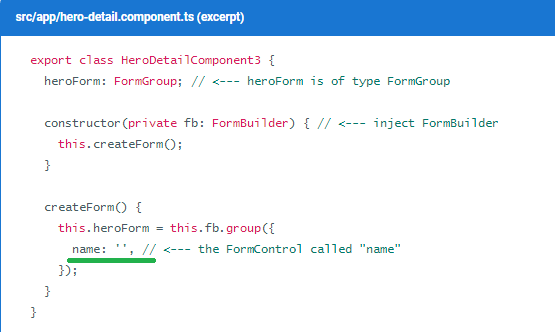
Explanation🡺

The heroForm.value returns the *form model*. Piping it through the JsonPipe renders the model as JSON in the browser:

**Introduction to *FormBuilder***

1. To use FormBuilder, you need to import it

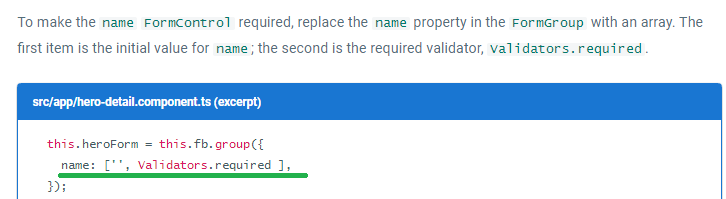
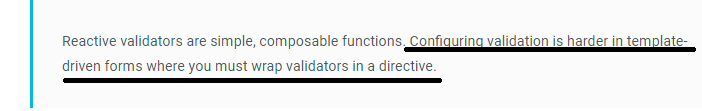


1. 

Explanation🡺

FormBuilder.group is a factory method that creates a FormGroup.   FormBuilder.group takes an object whose keys and values are FormControl names and their definitions. In this example, the name control is defined by its initial data value, an empty string.

### Validators.required

1. First, import the Validators🡺import { FormBuilder, FormGroup, Validators } from '@angular/forms';
2. 
3. 
4. 
5. Now let’s see the complete example with all the form fields,

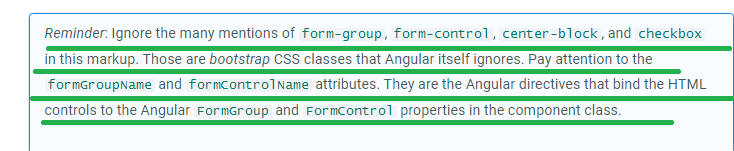


Html🡺

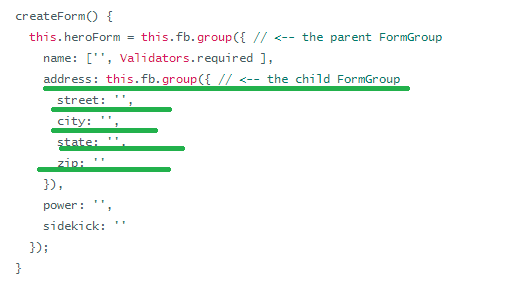


Explanation:

You must bind the option's value property with [value]="state". If you do not bind the value, the select shows the first option from the data model.

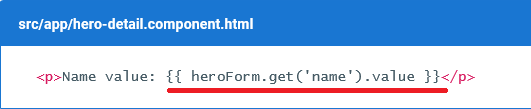
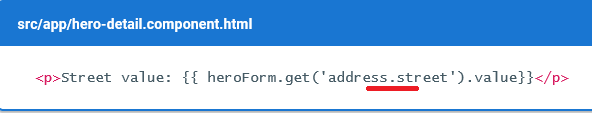
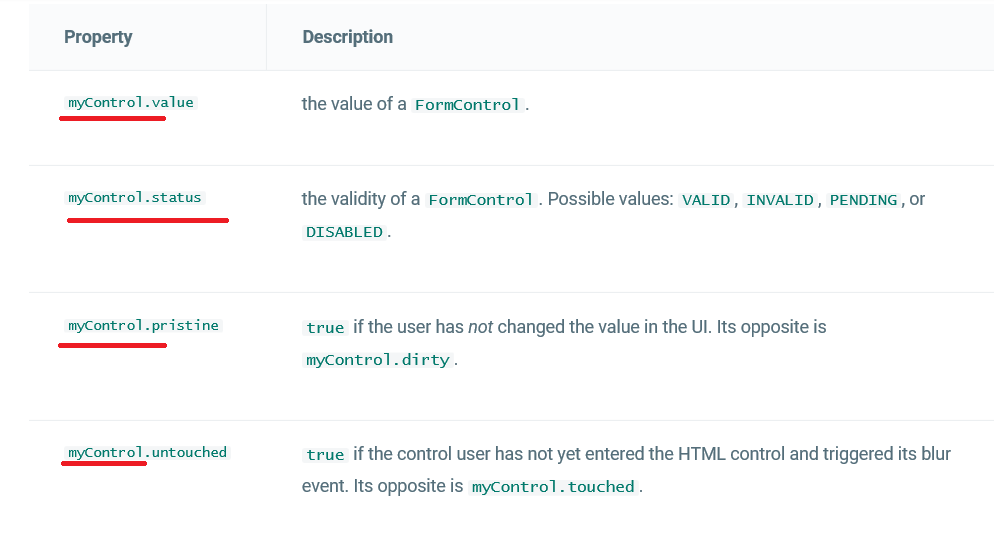


Nested FormGroups

1. 
2. Now let’s see the equivalent change in html



Inspect FormControl Properties

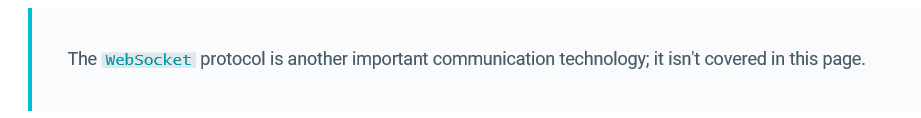
1. You can inspect(examine) an individual FormControl within a form by extracting it with the **.get()** method, You can inspect an individual FormControl within a form by extracting it with the .get() method
2. 
3. To get the state of a FormControl that’s inside a FormGroup, use dot notation to path to the control🡺
4. 

**The data model and the form model🡺 later**

### **setValue,** patchValue, When to set form model values (ngOnChanges), reset the form flags, Use FormArray to present an array of FormGroups, From address to secret lairs, Get the FormArray, Display the FormArray, Add a new lair to the FormArray, Remove a lair, Observe control changes, Save form data and etc.. we will see this later once clear with the basictopic

HTTP Client

1. [HTTP](https://tools.ietf.org/html/rfc2616) is the primary protocol for browser/server communication.

Note🡺 

1. Modern browsers support two HTTP-based APIs: [XMLHttpRequest (XHR)](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest) and [JSONP](https://en.wikipedia.org/wiki/JSONP). A few browsers also support [Fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API).
2. The Angular HTTP library simplifies application programming with the **XHR** and **JSONP** APIs.

Demos

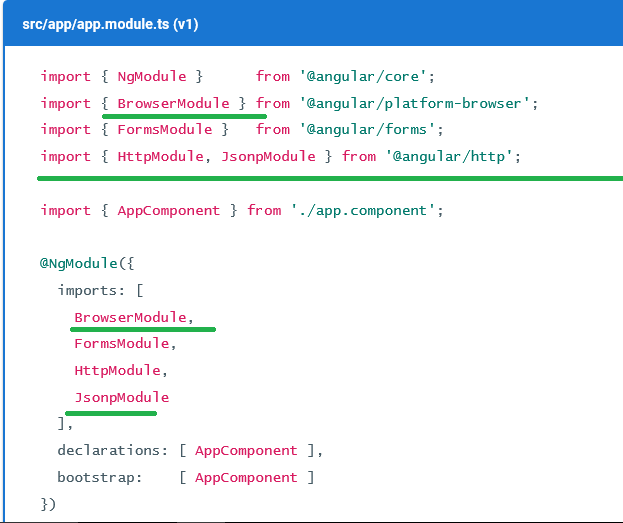
1. This page describes server communication with the help of the following demos:

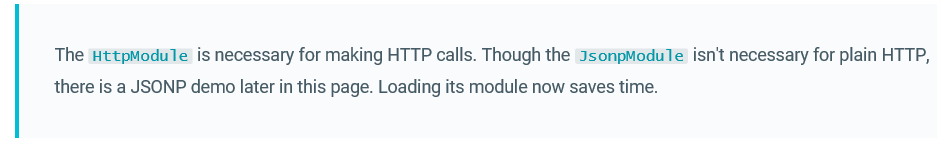
* [The Tour of Heroes HTTP client demo](https://angular.io/docs/ts/latest/guide/server-communication.html#http-client).
* [Fall back to Promises](https://angular.io/docs/ts/latest/guide/server-communication.html#promises).
* [Cross-Origin Requests: Wikipedia example](https://angular.io/docs/ts/latest/guide/server-communication.html#cors).
* [More fun with Observables](https://angular.io/docs/ts/latest/guide/server-communication.html#more-observables).

1. 

# Providing HTTP services

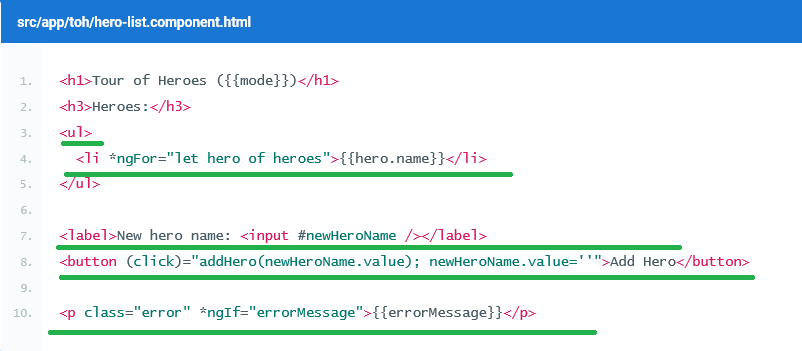
1. First, configure the application to use server communication facilities.
2. The Angular Http client communicates with the server using a familiar HTTP request/response protocol. The Http client is one of a family of services in the Angular HTTP library.
3. When importing from the @angular/http module, SystemJS knows how to load services from the Angular HTTP library because the systemjs.config.js file maps to that module name.
4. Before you can use the Http client, you need to register it as a service provider with the dependency injection system
5. Register providers by importing other NgModules to the root NgModule in app.module.ts.



1. 

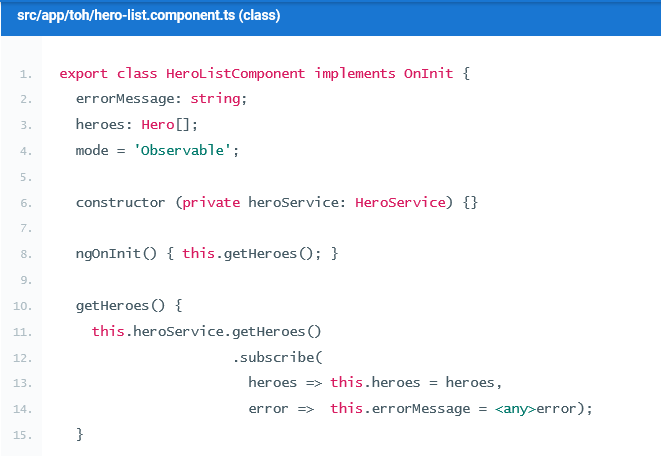
**The Tour of Heroes HTTP client demo**

1. The first demo is a mini-version of the [tutorial](https://angular.io/docs/ts/latest/tutorial)'s "Tour of Heroes" (ToH) application. This version gets some heroes from the server, displays them in a list, lets the user add new heroes, and saves them to the server. The app uses the Angular Http client to communicate via **XMLHttpRequest (XHR)**.
2. For Final output 🡺 see the Tutorial
3. This demo has a single component, the HeroListComponent
4. Now let’s see the html file

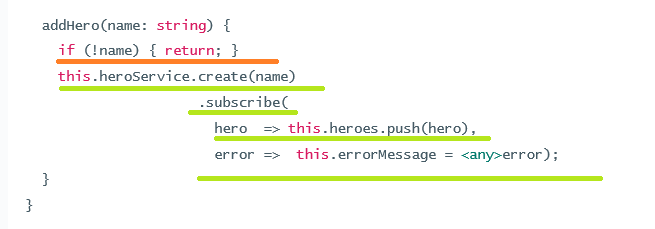


1. Now let’s see the component file

Part1



Part2



Explanation🡺

Angular [injects](https://angular.io/docs/ts/latest/guide/dependency-injection.html) a HeroService into the constructor and the component calls that service to fetch and save data. The component **does not talk directly to the Angular Http client**. The component doesn't know or care how it gets the data. It delegates to the HeroService.This is a golden rule: **always delegate data access to a supporting service class**.

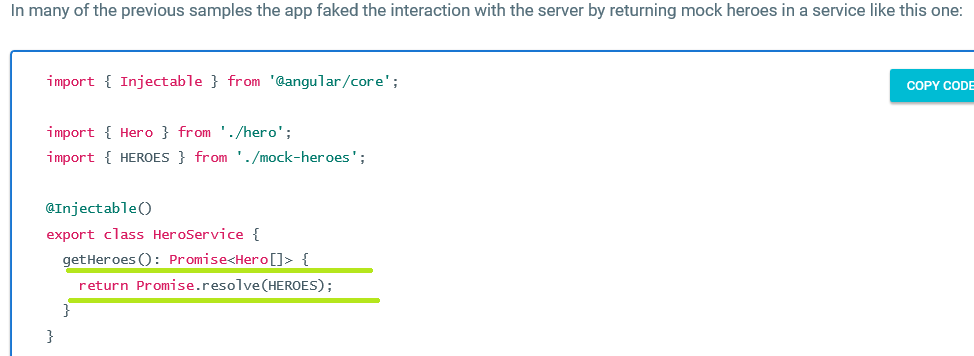
Although at runtime the component requests heroes immediately after creation, you **don't** call the service's get method in the component's constructor. Instead, call it inside the ngOnInit [lifecycle hook](https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html) and rely on Angular to call ngOnInit when it instantiates this component



The service's getHeroes() and create() methods return an Observable of hero data that the Angular Http client fetched from the server.Think of an Observable as a stream of events published by some source. To listen for events in this stream, ***subscribe*** to the Observable. These subscriptions specify the actions to take when the web request produces a success event (with the hero data in the event payload) or a fail event (with the error in the payload).

1. Now let’s start to see the service class

**Fetch data with *http.get()***



**You can revise that HeroService to get the heroes from the server using the Angular Http client service:**

**Part1**



Part2

