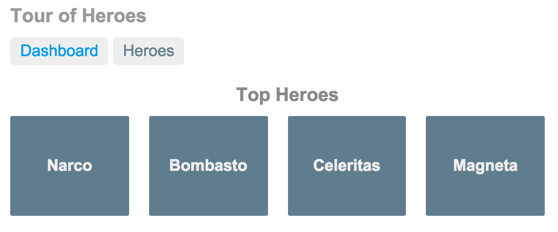
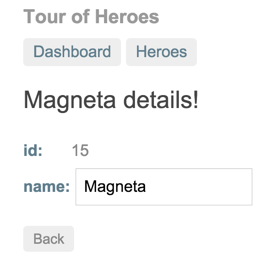
1. use one-way data binding for read-only data.
2. Our final application looks like,



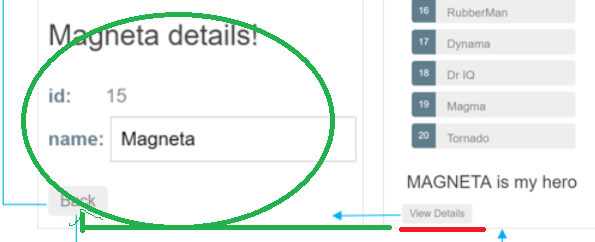
1. You can click the two links above the dashboard ("Dashboard" and "Heroes") to navigate between this Dashboard view and a Heroes view.
2. If you click the dashboard hero "Magneta," the router opens a "**Hero Details**" view where you can change the hero's name.



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.



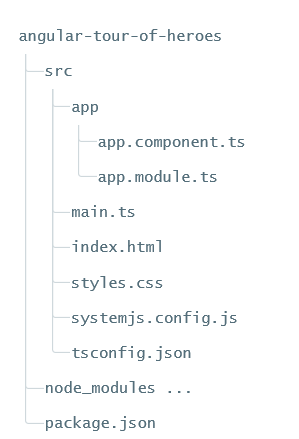
1. You can click the "View Details" button to drill into the editable details of the selected hero.

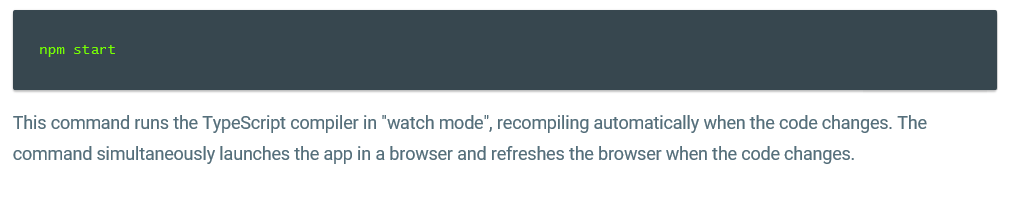
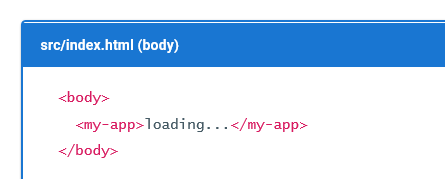


1. The following diagram captures all of the navigation options.

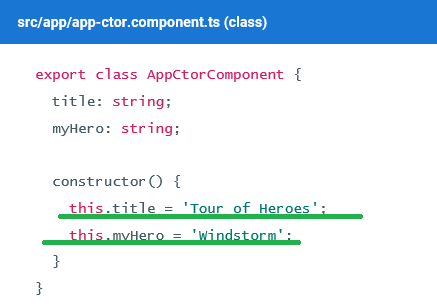


1. The file structure should look like this:

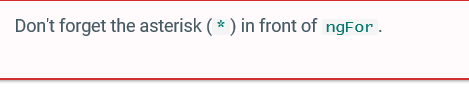


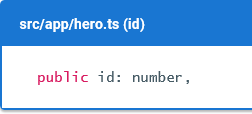
1. 
2. The double curly braces are Angular's interpolation binding syntax.
3. Property binding helps show app data in the UI
4. The template is a multi-line string within ECMAScript 2015 backticks (`). The backtick (`)—which is not the same character as a single quote (')—allows you to compose a string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

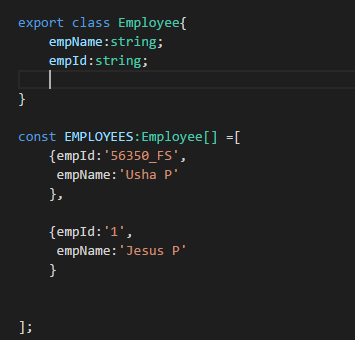
When you bootstrap with the AppComponent class (in main.ts), Angular looks for a <my-app> in the index.html, finds it, instantiates an instance of AppComponent, and renders it inside the <my-app> tag.

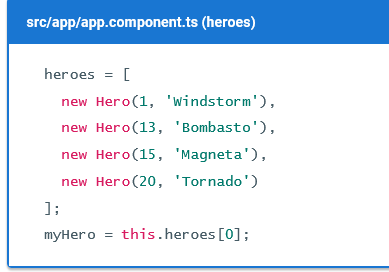
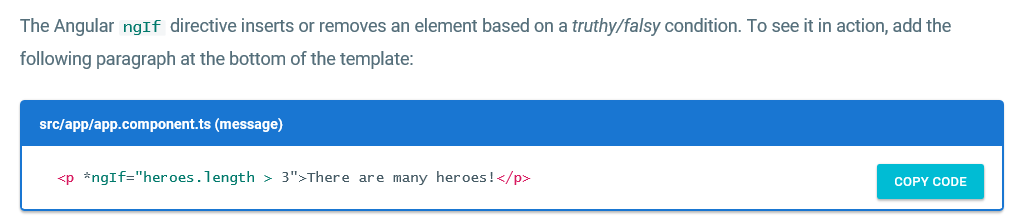
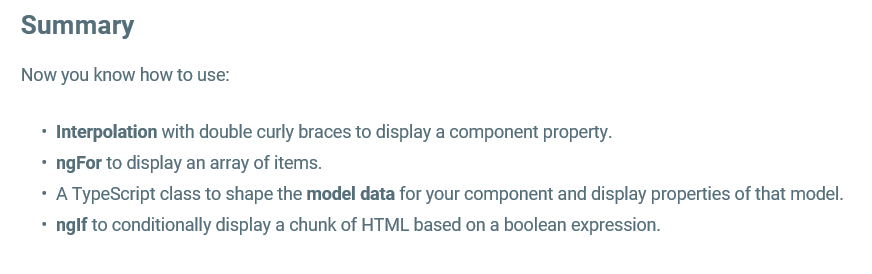
1. Template inline or template file? 🡺 You can store your component's template in one of two places. You can define it inline using the template property, or you can define the template in a separate HTML file and link to it in the component metadata using the @Component decorator's templateUrl property.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

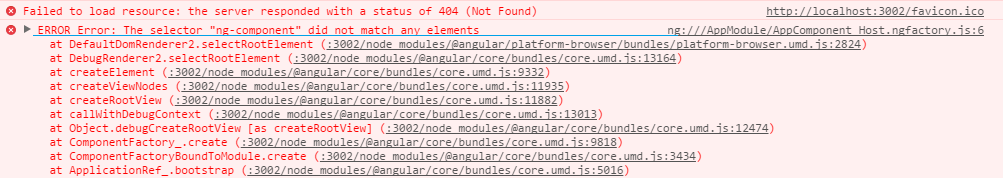




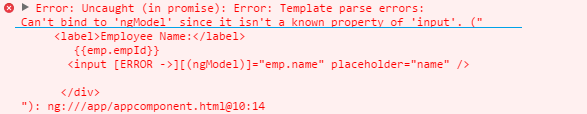
1. 

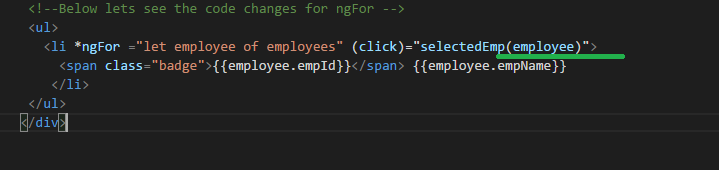
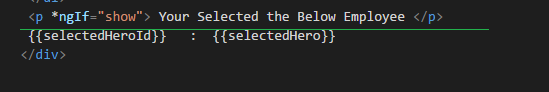


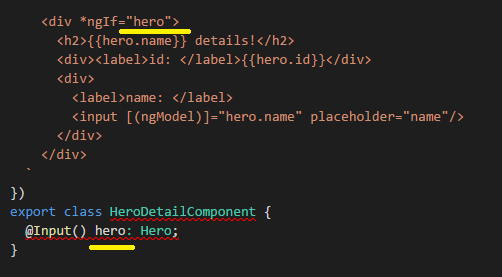
1. 
2. 
3. Don't forget the leading asterisk (\*) in \*ngIf.
4. 
5. Although NgModel is a valid Angular directive, it isn't available by default. It belongs to the optional FormsModule
6. import the FormsModule symbol from the @angular/forms library.
7. @NgModule metadata's imports array, **which contains 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



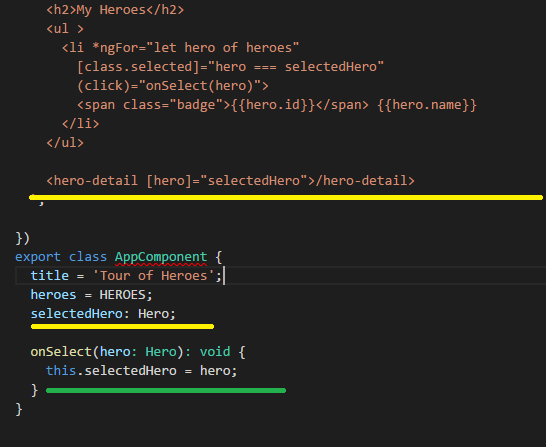
1. Import BrowserModule in the root module and CommonModule in other modules where you want to use common directives
2. CAN I USE THE "const" key word inside the class? 🡺 I was getting some error in the Studio
3. Without the FormModule, the Error in the Console is🡺

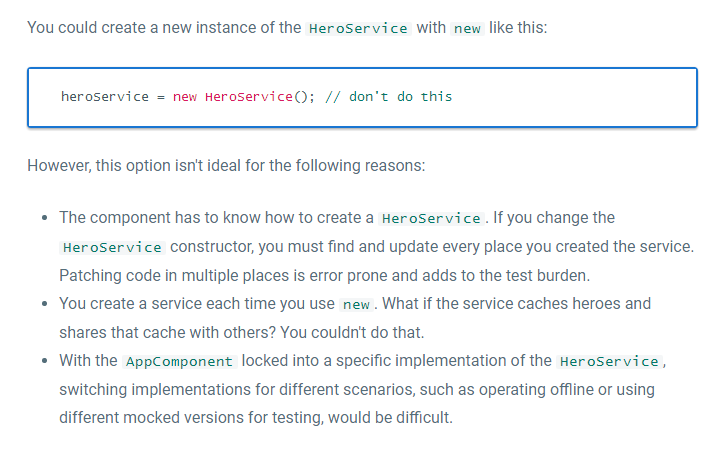


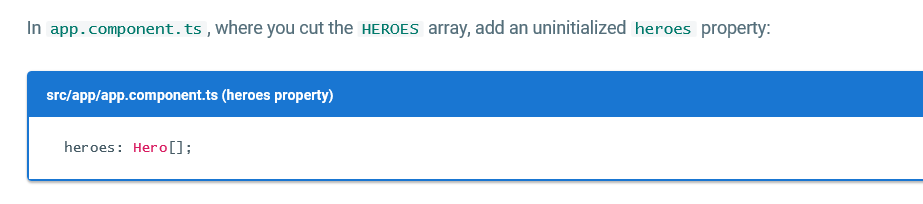
1. 
2. Syntax for \*ngIf is 🡺
3. The @Component decorator provides the Angular metadata for the component.
4. In general, the declarations array contains a list of application components, pipes, and directives that belong to the module.A component must be declared in a module before other components 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



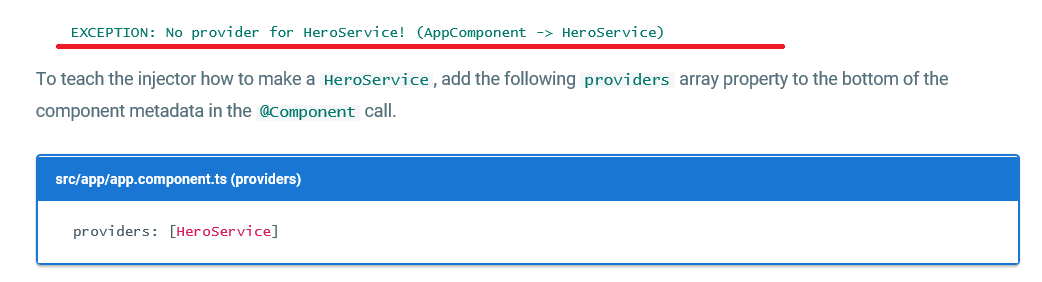
See here it's Quite Different way how the data is sent from parent to child compared to our Video Tutorials



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. 



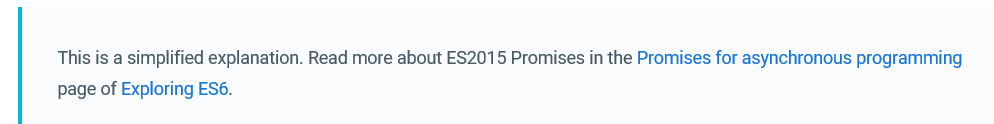
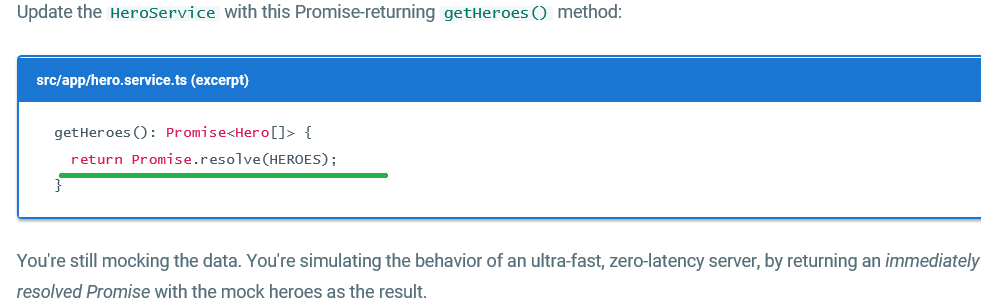


1. 
2. The providers array tells Angular to create a fresh instance of the HeroService when it creates an AppComponent. The AppComponent, as well as its child components, can use that service to get hero data.
3. The ngOnInit lifecycle hook
4. AppComponent should fetch and display hero data with no issues.
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

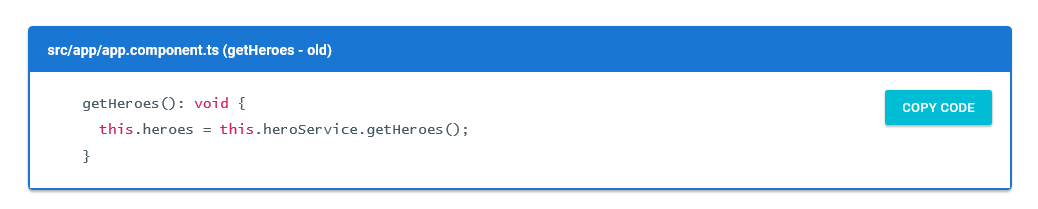
1. A Promise essentially promises to call back when the results are ready. You ask an asynchronous service to do some work and give it a callback function. The service does that work and eventually calls the function with the results or an error.
2. 
3. 

Act on the Promise

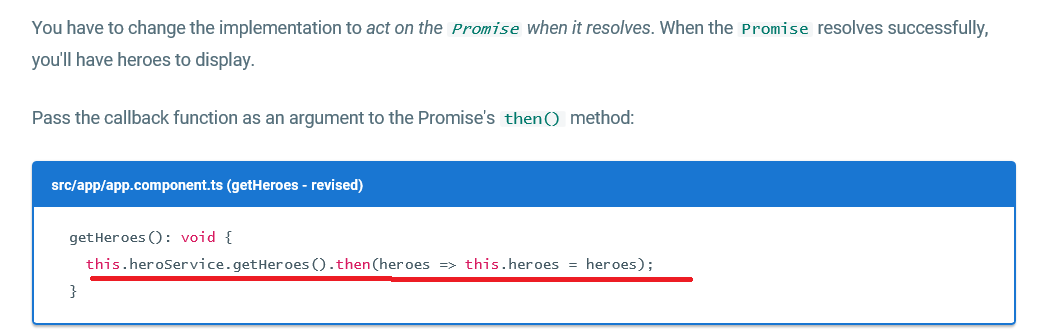
1. As a result of the change to HeroService, this.heroes is now set to a Promise rather than an array of heroes.

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.

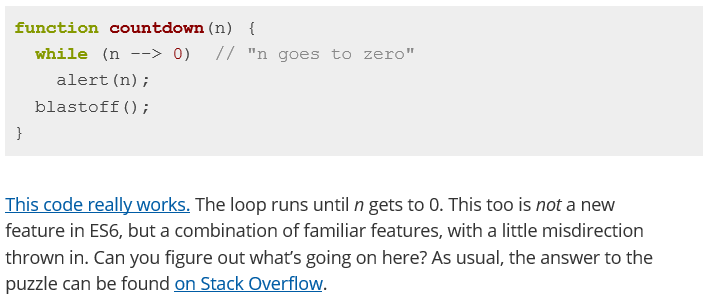
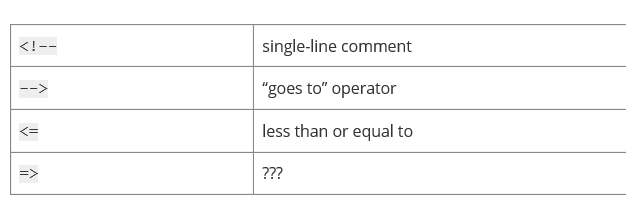
Ms🡺 lets see few concept with respect to Arrow functions,

* 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

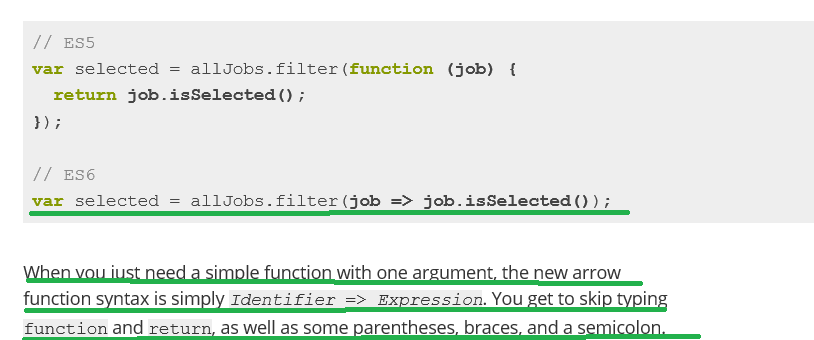


1. Arrow function in Es6🡺 <https://hacks.mozilla.org/2015/06/es6-in-depth-arrow-functions/>
   1. -->

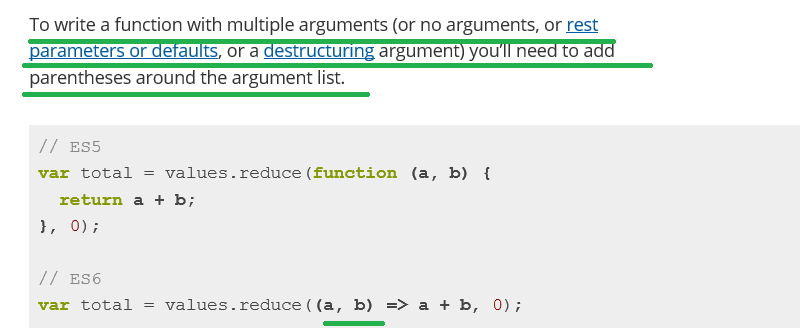
* This arrow indicates a comment *only* when it appears at the start of a line. That’s because in other contexts, --> is an operator in JS, the “goes to” operator!
* 
* 
* What happened to =>? Today, we find out. First, let’s talk a bit about functions.

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

Example 1:



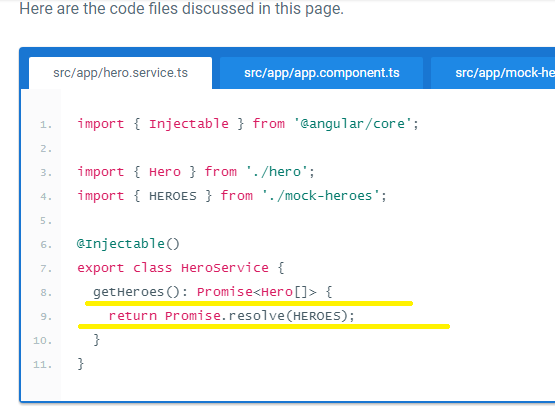
Example 2:



Example 3:

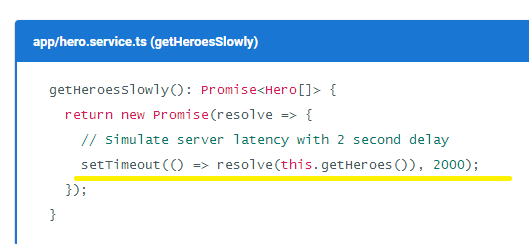


Now let us continue with our Angular Tutorials

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🡺

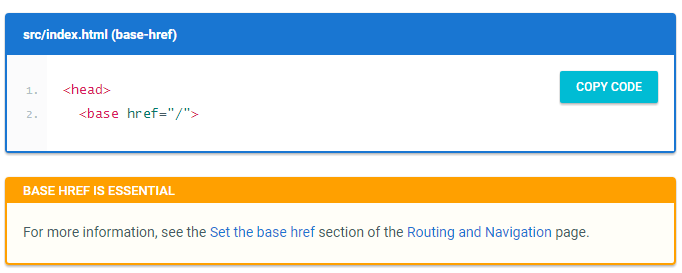


In this part I will just cover the important concept

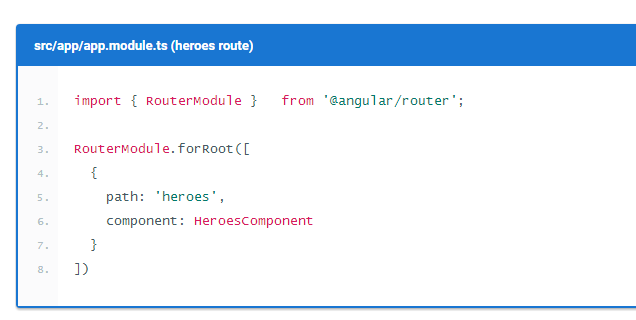
1. Add HeroesComponent to the declarations array of AppModule so Angular recognizes the <my-heroes> tags. 🡺
2. **Add HeroService to the providers array of AppModule because you'll need it in every other view**.
3. The Angular router is an external, optional Angular NgModule called RouterModule. The router 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

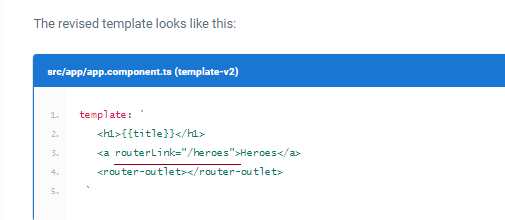
1. Routes tell the router which views to display when a user clicks a link or pastes a URL into the browser address bar.
2. 
3. The forRoot() method supplies the Router service providers and directives needed for routing, and performs the initial navigation based on the current browser URL.Router outlet

### **Router outlet**

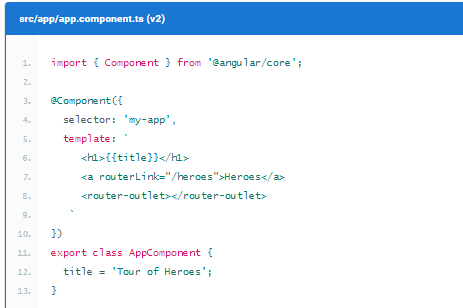
1. However, you have to tell the router where to display the component. To do this, you can add a <router-outlet> element at the end of the template. RouterOutlet is one of the directives provided by the RouterModule. The router displays each component immediately below the <router-outlet> as users navigate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

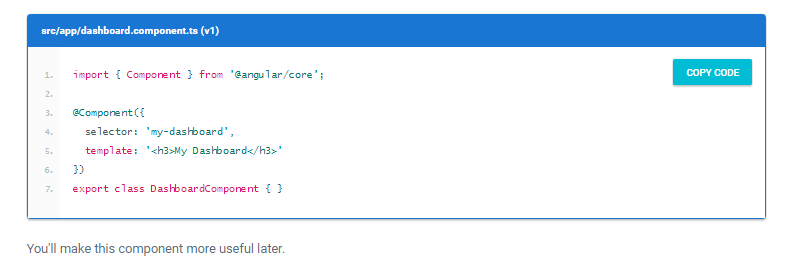
### **Router links**

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

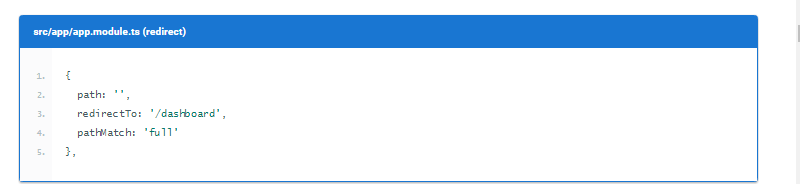




🡺And do the necessary changes in app.module.ts

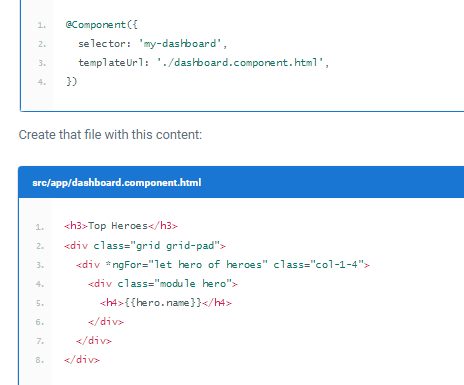
### **Add a redirect route**

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:

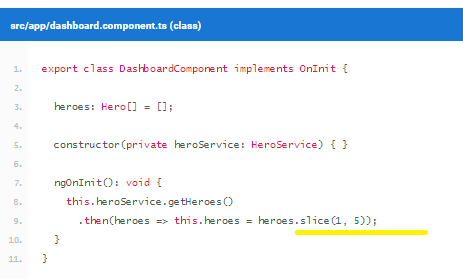


1. The <nav> tags don't do anything yet, but they'll be useful later when you style the links.

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

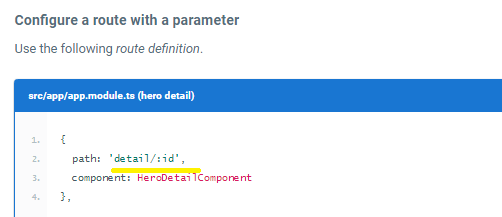


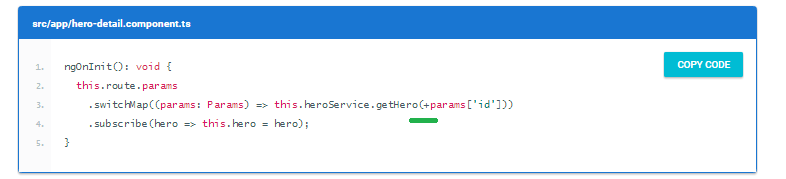
Now let's have a look at the body contents of the dashboard component,



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.

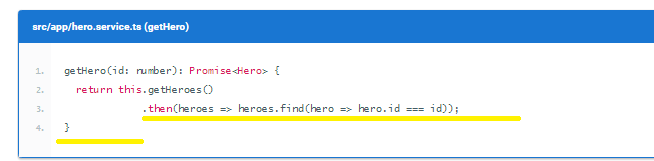
**The hero id is a number. Route parameters are always strings. So the route parameter value is converted to a number with the JavaScript (+) operator.**

### 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.

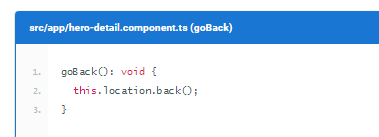


### **Find the way back**

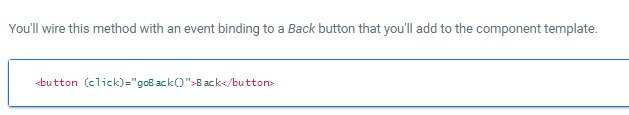
Now let us see the Reason for Location service

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

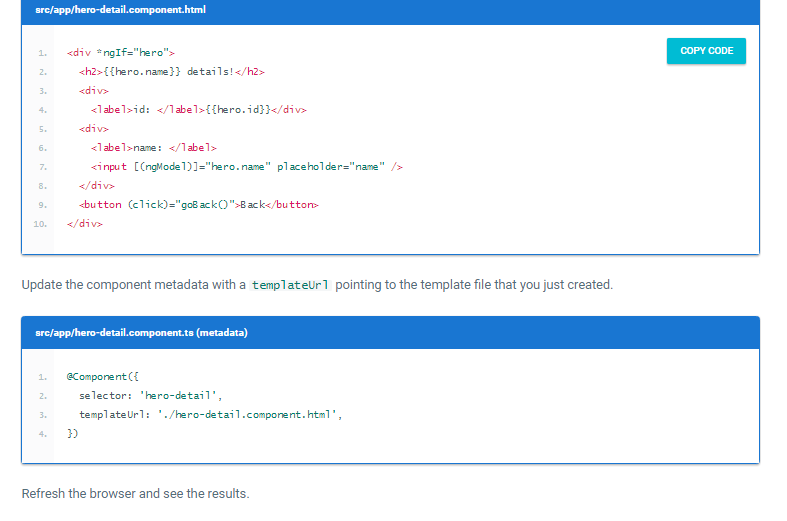
To navigate somewhere else, users can click one of the two links in the AppComponent or click the browser's back button. Now add a third option, a goBack() method that navigates backward one step in the browser's history stack using the Locationservice you injected previously.



* 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.

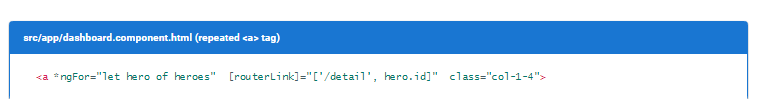


Now lets the Complete code



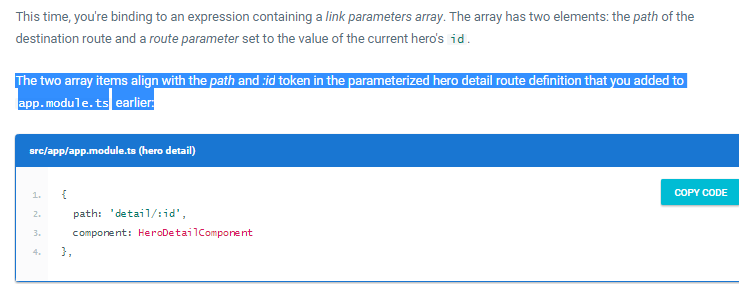
## Select a dashboard hero

* 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:

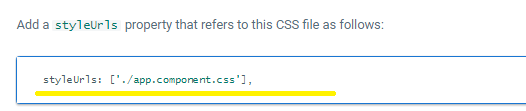


**This time, you're binding to an expression containing a *link parameters array*. The array has two elements: the *path* of the destination route and a *route parameter* set to the value of the current hero's id.**

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**

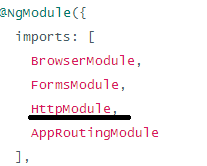
* Here we will see the logic to Convert the service and components to use Angular's HTTP service

**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**

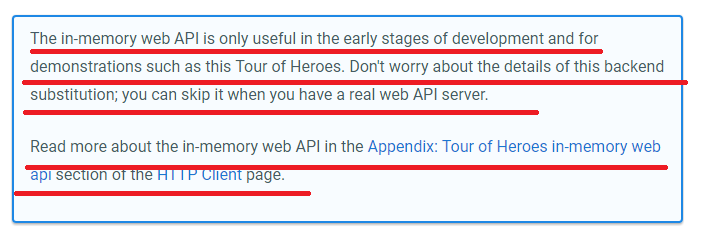
* Our app will depend on the Angular http service, which itself depends on other supporting services.
* The HttpModule from the @angular/http library holds providers for a complete set of HTTP services.
* To allow access to these services from anywhere in the app, add HttpModule to the imports list of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



## Simulate the web API

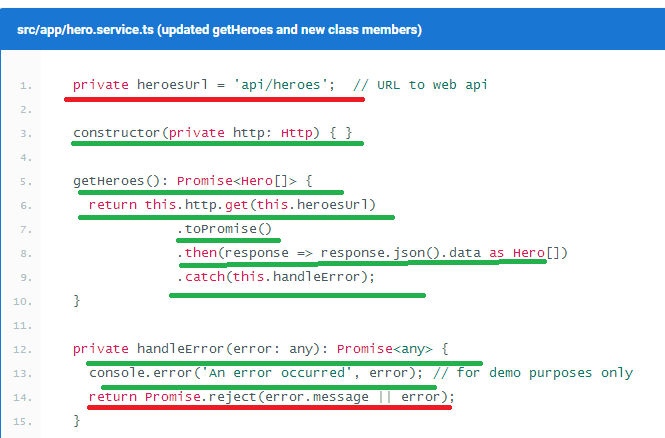
* We recommend registering app-wide services in the root AppModule *providers*.
* Until you have a web server that can handle requests for hero data, the HTTP client will fetch and save data from a mock service, the in-memory web API.
* **Rather than require a real API server, this example simulates communication with the remote server by adding the**[**InMemoryWebApiModule**](https://github.com/angular/in-memory-web-api)**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

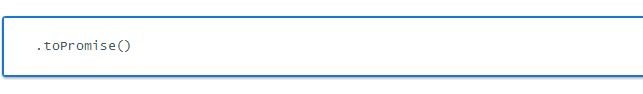


And also the following code changes is done in the above file, that’s is the “import parts”

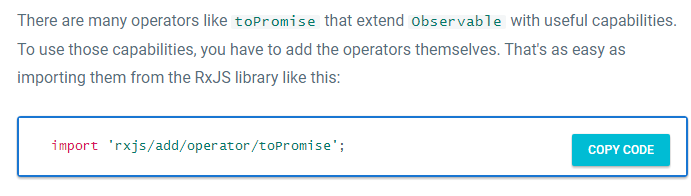
1. import 'rxjs/add/operator/**toPromise'**;

## 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.

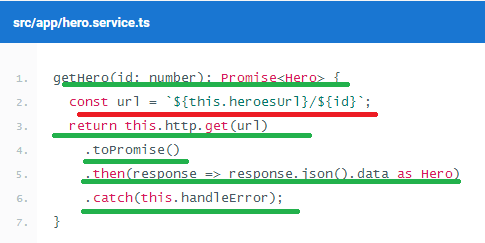
### **Error Handling**

* 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:

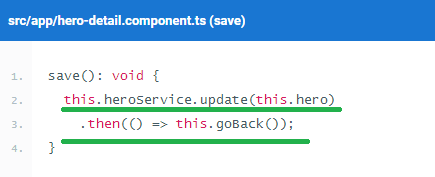


Note heroesUrl is defined in the above screen shot

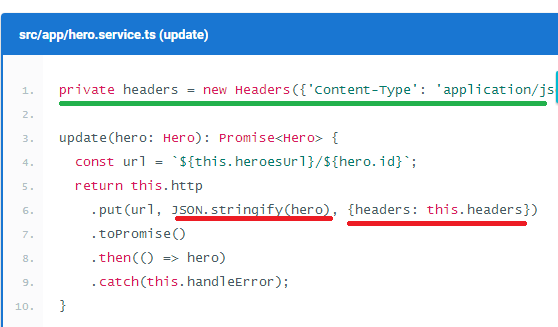
* **The data in the response is a single hero object rather than an array**.

### **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.**
* 

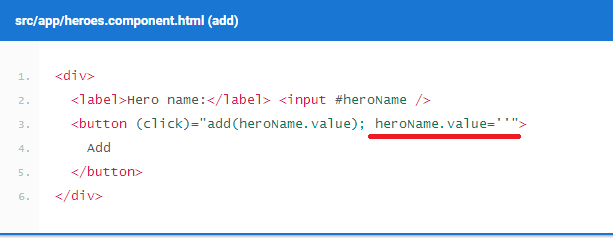
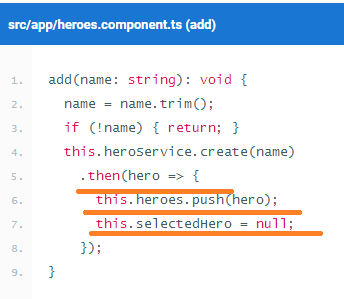
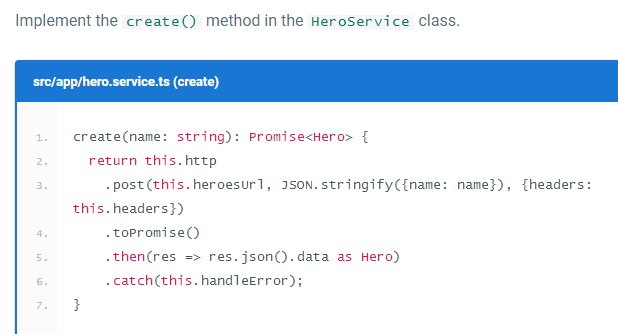
Explanation

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

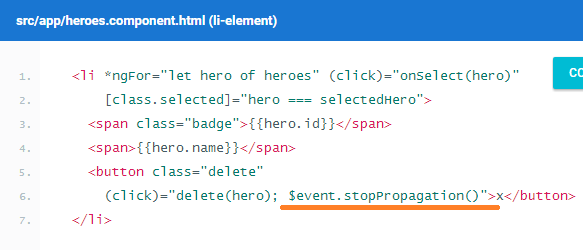
🡺 The put() body is the JSON string encoding of the hero, obtained by calling JSON.stringify

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.
* 
* 
* 

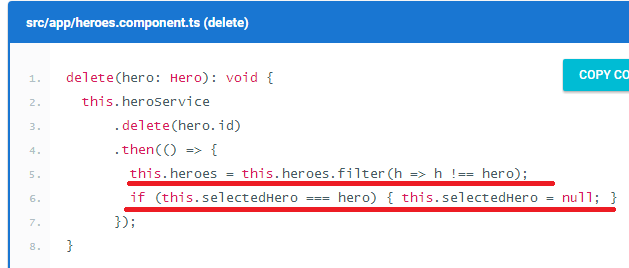
### **Add the ability to delete a hero**

* 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**

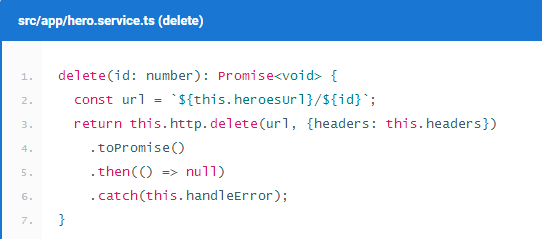
**In addition to calling the component's delete() method, the delete button's click handler code stops the propagation of the click event—you don't want the <li> click handler to be triggered because doing so would select the hero that the user will delete.**

* 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).
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