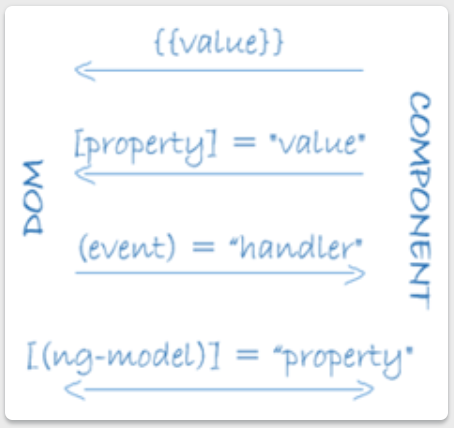
|  |
| --- |
| abc |



# **Links**

|  |  |
| --- | --- |
| Document | <https://angular.io/docs> |
| Online IDE for 1st App | <https://stackblitz.com/angular/odpeknvxnlq?file=src%2Fapp%2Fapp.component.ts> |
| Search API | <https://angular.io/api?type=pipe> |
| Template syntax | [https://angular.io/guide/template-syntax](https://angular.io/guide/template-syntax#the-pipe-operator-) |
| Directive | Attribute Directives: <https://angular.io/guide/attribute-directives>  <input [(ngModel)]="hero.name">  <a [title]="product.name + ' details'"> |
| Structural Directives: <https://angular.io/guide/structural-directives>  \*ngFor  \*ngIf  \*ngSwitch: <https://angular.io/guide/template-syntax#ngSwitch>  \*ngStyle: <https://angular.io/guide/template-syntax#ngStyle>  \*ngClass: <https://angular.io/guide/template-syntax#ngClass> |
| Service / Dependency Injection  (share data) | <https://angular.io/guide/dependency-injection> |
| Glossary (bảng thuật ngữ) | <https://angular.io/guide/glossary> |
| CLI – Command Line Interface | <https://angular.io/cli> |
| Lifecycle hooks | <https://angular.io/guide/lifecycle-hooks> |
| Observe API response  Subcribe data | <https://angular.io/guide/http>  <https://angular.io/tutorial/toh-pt4#observable-data>  <https://rxjs-dev.firebaseapp.com/guide/overview> |
| {{ interpolation binding }} | <https://angular.io/guide/template-syntax#interpolation> |

# **CLI commands**

<https://angular.io/cli>

|  |  |
| --- | --- |
| npm install -g @angular/cli | install the Angular CLI globally |
| **ng new** my-project-name | Create a new Angular CLI workspace project  & initial application |
| cd new my-project-name  ng build –prod | built project |
| ng serve –open  ng serve –o | Run the application  <http://localhost:4200/> |
|  |  |
| **ng generate component** heroes | generate a new ”heroes” **component** |
| **ng generate service** hero | generate a new ”hero” **service** |
| **ng generate module** app-routing **--flat --module=app** | generate a new ”app-routing” **module**  **--flat** puts the file in src/app instead of its own folder.  **--module=app** tells the CLI to register it in the imports array of the AppModule. |

# **Syntax**

|  |  |  |
| --- | --- | --- |
| Property binding [ ] |  | <https://angular.io/guide/template-syntax#property-binding-property> |
| <a [title]="product.name + ' details'">  {{ product.name }}  </a> | <a title="Phone XL details">  Phone XL  </a> |
| <a [routerLink]="['/products', productId]">  {{ product.name }}  </a> | <a href="/products/8">  Phone XL  </a>  <https://angular.io/api/router/RouterLink> |
| [pipe](https://angular.io/guide/pipes) | <b>{{ product.price | currency }}</b> | uses the currency pipe to transform product.price from a number to a currency string.  A pipe is a way you can transform data in your HTML template  <https://angular.io/guide/template-syntax#the-pipe-operator->  <https://angular.io/api?type=pipe>  <https://angular.io/api/common/CurrencyPipe>  <https://angular.io/api/common/DatePipe> |
| <div \*ngFor="let shipping of shippingCosts | async"> | returns the latest value from a stream of data & continues to do so for the life of a given component.  When Angular destroys that component, the async pipe automatically stops  <https://angular.io/api/common/AsyncPipe> |
| ngModel  (data binding) | <input [(ngModel)]="hero.name"> | two-way data binding syntax  app.module.ts  import { [FormsModule](https://angular.io/api/forms/FormsModule) } from '@angular/forms';  imports: [  [BrowserModule](https://angular.io/api/platform-browser/BrowserModule),  [FormsModule](https://angular.io/api/forms/FormsModule)  ], |
| Class binding | [class.some-css-class]="some-condition"  [class.selected]="hero === selectedHero" | <https://angular.io/guide/template-syntax#class-binding> |
| onchange  (on typing) | <input #searchBox id="search-box" (input)="search(searchBox.value)" /> | Every time the user types in the textbox, the binding calls search() with the textbox value |

|  |  |  |
| --- | --- | --- |
| ngFor |  | <li \*ngFor="let product of products">  {{ product.name }}  </li> |
| ngFor  &  index |  | <div \*ngFor="let product of products; index as productId">  <a [title]="product.name + ' details'" [routerLink]="['/products', productId]">  {{ product.name }}  </a>  </div> |
| ngIf |  | <p \*ngIf="product.description">  {{ product.description }}  </p> |
| click | Event binding ( )  <https://angular.io/guide/template-syntax#event-binding> | <button (click)="share()">  Share  </button> |

# **Import**

* Input (data fr parent)

[Getting Started / Your First App # Input](https://angular.io/start#input)

<https://angular.io/guide/template-syntax#how-to-use-input>

<https://angular.io/guide/template-syntax#input-and-output-properties>

The @[Input](https://angular.io/api/core/Input)() decorator indicates that the property value passes in from the component's parent

|  |
| --- |
| src/app/product-alerts/product-alerts.component.ts  import { [Input](https://angular.io/api/core/Input) } from '@angular/core';  @[Input](https://angular.io/api/core/Input)() product;  src/app/product-alerts/product-alerts.component.html  <p \*[ngIf](https://angular.io/api/common/NgIf)="product.price > 700"> <button>Notify Me</button> </p>  src/app/product-list/product-list.component.html  <app-product-alerts [product]="product"> </app-product-alerts> |

* Output (emit event)

[Getting Started / Your First App # Output](https://angular.io/start#output)

<https://angular.io/guide/template-syntax#how-to-use-output>

<https://angular.io/guide/template-syntax#input-and-output-properties>

Output(emit) an event

|  |
| --- |
| src/app/product-alerts/product-alerts.component.ts  import { [Output](https://angular.io/api/core/Output), [EventEmitter](https://angular.io/api/core/EventEmitter) } from '@angular/core';  @[Output](https://angular.io/api/core/Output)() notify = new [EventEmitter](https://angular.io/api/core/EventEmitter)();  src/app/product-alerts/product-alerts.component.html  <button (click)="notify.emit()">Notify Me</button>  src/app/product-list/product-list.component.ts  onNotify() {  window.alert('You will be notified when the product goes on sale');  }  src/app/product-list/product-list.component.html  <app-product-alerts [product]="product" (notify)="onNotify()"> </app-product-alerts> |

* Routing (URL params)

[Getting Started / Routing](https://angular.io/start/start-routing)

<https://angular.io/api/router/RouterModule>

<https://angular.io/api/router/RouterOutlet>

URL parameters

|  |
| --- |
| src/app/app.module.ts  @NgModule({  imports: [  BrowserModule,  ReactiveFormsModule,  RouterModule.forRoot([  { path: '', component: ProductListComponent },  { path: 'products/:productId', component: ProductDetailsComponent },  ])  ],  src/app/product-list/product-list.component.html  <a [routerLink]="['/products', productId]">  {{ product.name }}  </a>  src/app/product-details/product-details.component.ts  import { [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) } from '@angular/router';  constructor( private route: [ActivatedRoute](https://angular.io/api/router/ActivatedRoute), ) { }  src/app/product-details/product-details.component.ts  ngOnInit() {  this.route.paramMap.subscribe(params => {  this.product = products[+params.get('productId')];  });  } |

* Services (share data)

[Getting Started / Managing Data #Services](https://angular.io/start/start-data#services)

<https://angular.io/guide/providers>

To share data across components

When you provide the service at the root level, Angular creates a single, shared instance of HeroService and injects into any class that asks for it. Registering the provider in the @Injectable metadata also allows Angular to optimize an app by removing the service if it turns out not to be used after all.

|  |
| --- |
| src/app/cart.service.ts  import { [Injectable](https://angular.io/api/core/Injectable) } from '@angular/core';  @Injectable({  providedIn: 'root'  })  export class CartService {  items = [];  addToCart(product) {  this.items.push(product);  }  getItems() {  return this.items;  }  }  src/app/product-details/product-details.component.ts import { CartService } from '../cart.service';  export class ProductDetailsComponent implements OnInit {  constructor(  private cartService: CartService  ) { }  addToCart(product) {  this.cartService.addToCart(product);  window.alert('Your product has been added to the cart!');  }  }  src/app/product-details/product-details.component.html  <button (click)="addToCart(product)">Buy</button>  src/app/cart/cart.component.ts  import { CartService } from '../cart.service';  export class CartComponent implements OnInit {  items;  constructor(  private cartService: CartService  ) { }  ngOnInit() {  this.items = this.cartService.getItems();  }  }  src/app/cart/cart.component.html  <div class="cart-item" \*ngFor="let item of items">  <span>{{ item.name }}</span>  <span>{{ item.price | currency }}</span>  </div> |

* HttpClient (API JSON)

[Getting Started / Managing Data #HttpClient](https://angular.io/start/start-data#services)

<https://angular.io/guide/http>

<https://angular.io/api/common/http/HttpClient>

Your app can fetch data and interact with external APIs and resources like JSON

|  |
| --- |
| src/assets/shipping.json  [  {  "type": "Overnight",  "price": 25.99  },  {  "type": "2-Day",  "price": 9.99  },  {  "type": "Postal",  "price": 2.99  }  ]  src/app/app.module.ts  import { [HttpClientModule](https://angular.io/api/common/http/HttpClientModule) } from '@angular/common/[http](https://angular.io/api/common/http)';  (at the top of the file with the other imports)  @NgModule({  imports: [  BrowserModule,  HttpClientModule,  ReactiveFormsModule,  RouterModule.forRoot([  { path: '', component: ProductListComponent },  { path: 'products/:productId', component: ProductDetailsComponent },  { path: 'cart', component: CartComponent },  ])  ]  })  src/app/cart.service.ts  import { [HttpClient](https://angular.io/api/common/http/HttpClient) } from '@angular/common/[http](https://angular.io/api/common/http)';  constructor( private [http](https://angular.io/api/common/http): [HttpClient](https://angular.io/api/common/http/HttpClient) ) {}  getShippingPrices() {  return this.http.get('/assets/shipping.json');  }  src/app/shipping/shipping.component.ts  import { CartService } from '../cart.service';  export class ShippingComponent implements OnInit {  shippingCosts;  constructor(  private cartService: CartService  ) {  }  ngOnInit() {  this.shippingCosts = this.cartService.getShippingPrices();  }  }  src/app/shipping/shipping.component.html  <div class="shipping-item" \*ngFor="let shipping of shippingCosts | async">  <span>{{ shipping.type }}</span>  <span>{{ shipping.price | currency }}</span>  </div> |

* Form

[Getting Started / Forms](https://angular.io/start/start-forms#define-the-checkout-form-model)

Create custom form controls and easy validation experiences

<https://angular.io/api/forms/FormsModule>

<https://angular.io/api/forms/FormBuilder>

<https://angular.io/api/forms/ReactiveFormsModule>

|  |
| --- |
| src/app/cart/cart.component.ts  import { [FormBuilder](https://angular.io/api/forms/FormBuilder) } from '@angular/forms';  export class CartComponent implements OnInit {  items;  checkoutForm;  constructor(  private cartService: CartService,  private formBuilder: FormBuilder,  ) {  this.checkoutForm = this.formBuilder.group({  name: '',  address: ''  });  }  onSubmit(customerData) {  // Process checkout data here  this.items = this.cartService.clearCart();  this.checkoutForm.reset();  console.warn('Your order has been submitted', customerData);  }  }  src/app/cart.service.ts  clearCart() {  this.items = [];  return this.items;  }  src/app/cart/cart.component.html  <form [formGroup]="checkoutForm" (ngSubmit)="onSubmit(checkoutForm.value)">  <div>  <label for="name">  Name  </label>  <input id="name" type="text" formControlName="name">  </div>  <div>  <label for="address">  Address  </label>  <input id="address" type="text" formControlName="address">  </div>  <button class="button" type="submit">Purchase</button>  </form> |

* XXX

Getting Started / XXX

UseForXXX

|  |
| --- |
| XXX |

# **NOTE**

* Reserve the constructor() for simple initialization such as wiring constructor parameters to properties. The constructor shouldn't *do anything*. It certainly shouldn't call a function that makes HTTP requests to a remote server as a *real* data service would.
* Instead, call getHeroes() inside the *[ngOnInit lifecycle hook](https://angular.io/guide/lifecycle-hooks)* and let Angular call ngOnInit() at an appropriate time *after* constructing a HeroesComponent instance.
* Angular only binds to **public** component properties

# **Tour of Heros App**

* Links

<https://angular.io/tutorial>

<https://stackblitz.com/angular/bbykkemeqor?file=src%2Fapp%2Fhero.service.ts>

* Visual Layout



* interface

|  |
| --- |
| export interface Hero {      id: number;      name: string;  } |

* Mock data

<https://angular.io/tutorial/toh-pt2#create-mock-heroes>

|  |
| --- |
| import { Hero } from '../hero';    hero: Hero = {      id: 1,      name: 'Windstorm'    };  export const HEROES: Hero[] = [    { id: 11, name: 'Dr Nice' },    { id: 12, name: 'Narco' },    { id: 13, name: 'Bombasto' },    { id: 14, name: 'Celeritas' },    { id: 15, name: 'Magneta' },    { id: 16, name: 'RubberMan' },    { id: 17, name: 'Dynama' },    { id: 18, name: 'Dr IQ' },    { id: 19, name: 'Magma' },    { id: 20, name: 'Tornado' },  ];  export class HeroesComponent implements OnInit {    selectedHero: Hero;    onSelect(hero: Hero): void {      this.selectedHero = hero;    }  } |

* Routing config

<https://angular.io/tutorial/toh-pt5#routes>

|  |
| --- |
| const routes: Routes = [    { path: '', redirectTo: '/dashboard', pathMatch: 'full' }, // auto navigate to dashboard & change '/' path to '/dashboard'    { path: 'dashboard', component: DashboardComponent },    { path: 'heroes', component: HeroesComponent }, // tells the router to match that URL to path: 'heroes' and display the HeroesComponent when the URL is something like localhost:4200/heroes    { path: 'detail/:id', component: HeroDetailComponent }, // :id is a placeholder for a specific hero id  ]; |

* Get id From URL

<https://angular.io/tutorial/toh-pt5#routable-herodetailcomponent>

|  |
| --- |
| import { ActivatedRoute } from '@angular/router';    constructor(      private route: ActivatedRoute, // holds parameter information of URL    ) { }      const id = +this.route.snapshot.paramMap.get('id');      // route parameters are always strings      // (+) operator converts the string to a number (which is what a hero id should be)      // route.snapshot is a static image of the route information shortly after the component was created      // paramMap is a dictionary of route parameter values extracted from the URL. The "id" key returns the id of the hero to fetch |

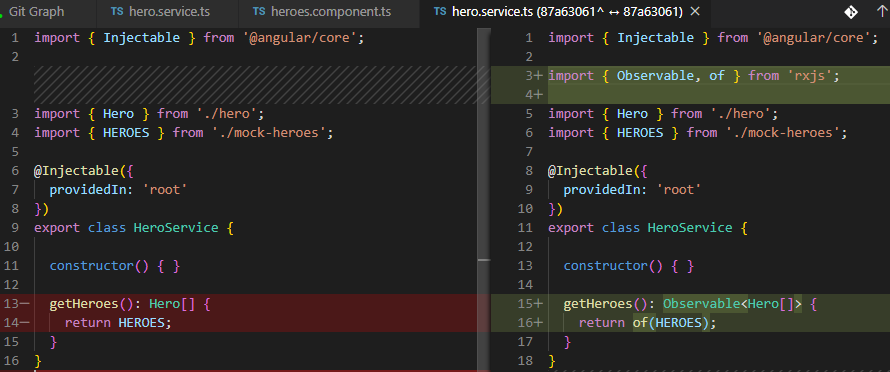
* goBack()

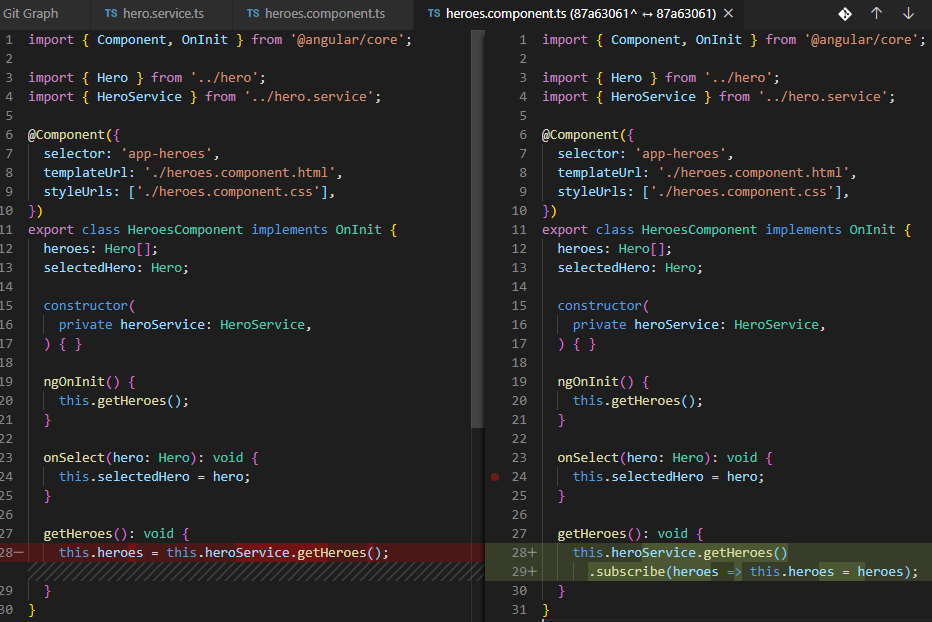
<https://angular.io/tutorial/toh-pt5#find-the-way-back>

|  |
| --- |
| import { Location } from '@angular/common';    constructor(      private location: Location, // service for interacting with the browser; to navigate back to the view that navigated here    ) { }    goBack(): void {      this.location.back();    } |

* Observable data (by RxJS library)

<https://angular.io/tutorial/toh-pt4#observable-data>





* Get data from server (by In-memory Web API module)
  + Get Heroes List

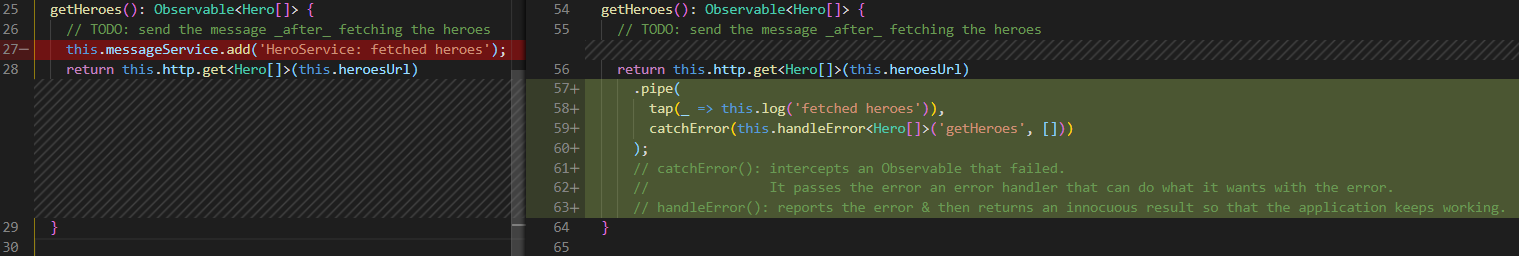
<https://angular.io/tutorial/toh-pt6#heroes-and-http>

* + Log message by RxJS tap()

<https://angular.io/tutorial/toh-pt6#handleerror>

<https://github.com/TonyTran0209/angular-tour-of-heroes/commit/2b4743a2fbcf505a92e78dbc285640d06193f2f2?diff=split>

|  |
| --- |
| import { catchError, map, tap } from 'rxjs/operators';    // After reporting the error to the console,    // the handler constructs a user friendly message    // and returns a safe value to the app so the app can keep working.    /\*\*     \* Handle Http operation that failed.     \* Let the app continue.     \* @param operation - name of the operation that failed     \* @param result - optional value to return as the observable result     \*/    private handleError<T>(operation = 'operation', result?: T) {      return (error: any): Observable<T> => {        // TODO: send the error to remote logging infrastructure        console.error(error); // log to console instead        // TODO: better job of transforming error for user consumption        this.log(`${operation} failed: ${error.message}`);        // Let the app keep running by returning an empty result.        return of(result as T);      };    }    /\*\* Log a HeroService message with the MessageService \*/    private log(message: string) {      this.messageService.add(`HeroService: ${message}`);    } |



|  |
| --- |
| getHeroes(): Observable<Hero[]> {      // TODO: send the message \_after\_ fetching the heroes      return this.http.get<Hero[]>(this.heroesUrl)        .pipe(          tap(\_ => this.log('fetched heroes')),          catchError(this.handleError<Hero[]>('getHeroes', []))        );        // catchError(): intercepts an Observable that failed. It passes the error an error handler that can do what it wants with the error.        // handleError(): reports the error & then returns an innocuous result so that the application keeps working.    } |

* + Get hero by id

<https://angular.io/tutorial/toh-pt6#get-hero-by-id>

<https://github.com/TonyTran0209/angular-tour-of-heroes/commit/5cc71262f82d5bdc83cd485e0fda335895ab476e>

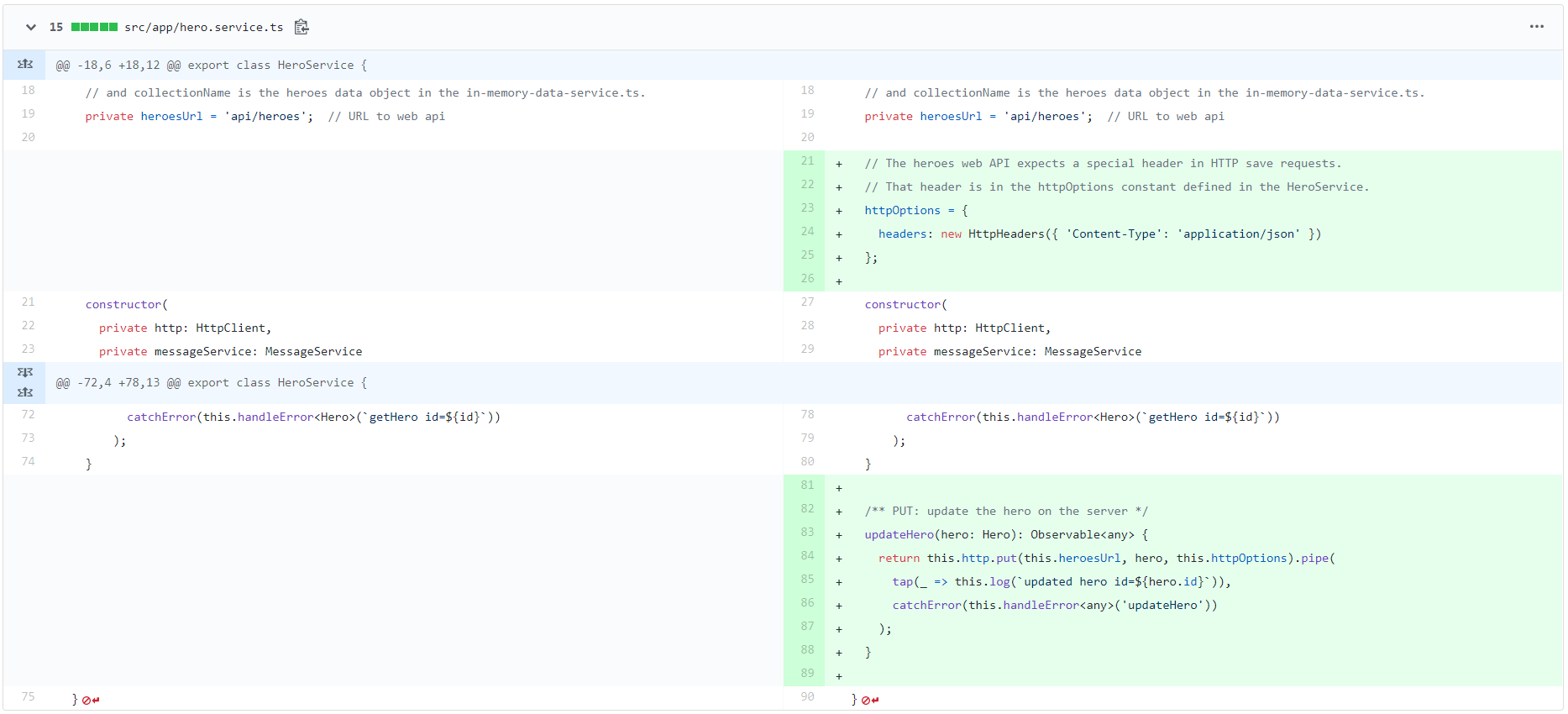
|  |
| --- |
| // GET hero by id. Will 404 if id not found    getHero(id: number): Observable<Hero> {      // TODO: send the message \_after\_ fetching the hero      const url = `${this.heroesUrl}/${id}`;      return this.http.get<Hero>(url)        .pipe(          tap(\_ => this.log(`fetched hero id=${id}`)),          catchError(this.handleError<Hero>(`getHero id=${id}`))        );    } |

* + Update hero by id

<https://angular.io/tutorial/toh-pt6#update-heroes>

<https://github.com/TonyTran0209/angular-tour-of-heroes/commit/245aef04aeb463fb837e1b084f0975f26e6a4923?diff=split>





* + Add a new hero

<https://angular.io/tutorial/toh-pt6#add-a-new-hero>

<https://github.com/TonyTran0209/angular-tour-of-heroes/commit/3d10a4dda7078c17f97b5b1fde2269242d7a2cbb?diff=split>

|  |
| --- |
| src/app/heroes/heroes.component.html  <div>    <label>Hero name:      <input #heroName />    </label>    <!-- (click) passes input value to add() & then clears the input -->    <button (click)="add(heroName.value); heroName.value=''">      add    </button>  </div>  src/app/heroes/heroes.component.ts    add(name: string): void {      name = name.trim();      if (!name) { return; }      this.heroService.addHero({ name } as Hero)        .subscribe(hero => {          this.heroes.push(hero);        });    }  src/app/hero.service.ts    /\*\* POST: add a new hero to the server \*/    addHero(hero: Hero): Observable<Hero> {      return this.http.post<Hero>(this.heroesUrl, hero, this.httpOptions).pipe(        tap((newHero: Hero) => this.log(`added hero w/ id=${newHero.id}`)),        catchError(this.handleError<Hero>('addHero'))      );    } |

* + Delete hero by id

<https://angular.io/tutorial/toh-pt6#delete-a-hero>

<https://github.com/TonyTran0209/angular-tour-of-heroes/commit/7640e720f3e9b06286b694f837d65e1cb1e18cf4?diff=split>

|  |
| --- |
| src/app/heroes/heroes.component.ts    delete(hero: Hero): void {      this.heroes = this.heroes.filter(h => h !== hero);      this.heroService.deleteHero(hero).subscribe();      // There's nothing for the component to do with the Observable returned by heroService.delete() but it must subscribe() anyway.      // If you don't use subscribe(), the service will not send the delete request to the server.      // As a rule, an Observable does nothing until something subscribes.    }  src/app/hero.service.ts    /\*\* DELETE: delete the hero from the server \*/    deleteHero(hero: Hero | number): Observable<Hero> {      const id = typeof hero === 'number' ? hero : hero.id;      const url = `${this.heroesUrl}/${id}`;      return this.http.delete<Hero>(url, this.httpOptions).pipe(        tap(\_ => this.log(`deleted hero id=${id}`)),        catchError(this.handleError<Hero>('deleteHero'))      );    } |

* + Search Hero by name

<https://angular.io/tutorial/toh-pt6#search-by-name>

<https://github.com/TonyTran0209/angular-tour-of-heroes/commit/80e937938ecefc6889cd8054aa86ba4b9ffd25d9?diff=split>

<https://www.learnrxjs.io/learn-rxjs/operators/transformation/switchmap>

|  |
| --- |
| src/app/hero-search/hero-search.component.html  <div id="search-component">    <h4><label for="search-box">Hero Search</label></h4>    <!-- As the user types in the search box, an input event binding calls the component's search() method with the new search box value. -->    <!-- Every time the user types in the textbox, the binding calls search() with the textbox value, a "search term".    The searchTerms becomes an Observable emitting a steady stream of search terms. -->    <input #searchBox id="search-box" (input)="search(searchBox.value)" />    <ul class="search-result">      <!-- The $ is a convention that indicates heroes$ is an Observable, not an array -->      <!-- async pipe identifies Angular's AsyncPipe & subscribes to an Observable automatically, so you won't have to do so in the component class -->      <li \*ngFor="let hero of heroes$ | async">        <a routerLink="/detail/{{ hero.id }}">          {{ hero.name }}        </a>      </li>    </ul>  </div>  src/app/hero-search/hero-search.component.ts  import { Component, OnInit } from '@angular/core';  import { Observable, Subject } from 'rxjs';  import {     debounceTime, distinctUntilChanged, switchMap   } from 'rxjs/operators';  import { Hero } from '../hero';  import { HeroService } from '../hero.service';  @Component({    selector: 'app-hero-search',    templateUrl: './hero-search.component.html',    styleUrls: ['./hero-search.component.css']  })  export class HeroSearchComponent implements OnInit {    heroes$: Observable<Hero[]>; // The $ is a convention that indicates heroes$ is an Observable, not an array    private searchTerms = new Subject<string>(); // is an RxJS Subject    constructor(private heroService: HeroService) {}    // Push a search term into the observable stream.    /\* A Subject is both a source of observable values & an Observable itself.    You can subscribe to a Subject as you would any Observable.    You can also push values into that Observable by calling its next(value) method as the search() method does. \*/    search(term: string): void {      this.searchTerms.next(term);    }    ngOnInit(): void {      this.heroes$ = this.searchTerms.pipe(        // wait until the flow of new string events pauses for 300ms before passing along the latest string        // You'll never make requests more frequently than 300ms        debounceTime(300),        // ensures that a request is sent only if the filter text changed        distinctUntilChanged(),        /\* switchMap() calls the search service for each search term that makes it through debounce() and distinctUntilChanged().        It cancels & discards previous search observables,        returning only the latest search service observable \*/        // https://www.learnrxjs.io/learn-rxjs/operators/transformation/switchmap        switchMap((term: string) => this.heroService.searchHeroes(term)),      );    }    }  src/app/hero.service.ts    /\* GET heroes whose name contains search term \*/    searchHeroes(term: string): Observable<Hero[]> {      if (!term.trim()) {        // if not search term, return empty hero array.        return of([]);      }      return this.http.get<Hero[]>(`${this.heroesUrl}/?name=${term}`).pipe(        tap(x => x.length ?          this.log(`found heroes matching "${term}"`) :          this.log(`no heroes matching "${term}"`)),        catchError(this.handleError<Hero[]>('searchHeroes', []))      );    } |

* XXX

|  |
| --- |
| abc |