آموزش Angular6ـ آموزش HTTP در Angular6

در این آموزش شما خواهید توانست که با کمک HttpClient موجود در Angular، ویژگی های پایداری داده (data persistence features) زیر را اضافه کنید.

- HeroService داده های hero را همراه با درخواست های HTTP دریافت می کند.
- کاربران می توانند Hero ها را اضافه، ویرایش و حذف کنند و این تغییرات را در HTTP ذخیره کنند.
- کاربران می توانند بر اساس نام hero ها، آن ها را جستجو کنند.

زمانی که کار شما با این بخش تمام شد، نرم افزار شما باید مانند این لینک باشد :

https://angular.io/generated/live-examples/toh-pt6/stackblitz.html

آموزش فعال سازی سرویس های HTTP

HttpClient مکانیزمی از Angular است که برای ایجاد برقراری ارتباط با یک remote server در HTTP استفاده می شود.

برای آنکه بتوانید HttpClient را در سرتاسر نرم افزار خود در دسترس قرار دهید باید:

- Root Appmodule را باز کنید.
- از @angular/common/http ، نشان HttpClientModule را import کنید.
- این نشان را به آرایه @NgModule.imports اضافه کنید.

آموزش شبیه سازی data server

این نمونه آموزشی با استفاده از ماژول In-memory Web API ، برقراری ارتباط با یک remote data server را شبیه سازی می کند.

پس از اینکه این ماژول را نصب کردید، نرم افزارتان بدون اینکه بداند که این ماژول در حال قطع کردن درخواست های ارسالی و دریافتی است، این درخواست ها را به HttpClient ارسال کرده و پاسخ را از آن دریافت می کند، سپس این پاسخ ها را در یک in-memory data store اعمال کرده و پاسخ های شبیه سازی شده را برگشت می دهد.

این تشکیلات کار را برای این آموزش بسیار آسان کرده است. دیگر برای یاد گرفتن HttpClient نیازی به سرپا کردن سرور نیست. همچنین اگر در مراحل اولیه برنامه نویسی هستید یا Web api سرورتان هنوز به خوبی تعریف یا پیاده سازی نشده است این امکانات می تواند کار شما را راحت کند.

نکته مهم: ماژول In-memory Web API هیچ ارتباطی با HTTP در Angular ندارد.

اگر صرفا هدف تان از خواندن این آموزش یادگیری HttpClient است می توانید این مرحله را رد کنید. اما اگر همراه با این آموزش در حال کدنویسی هستید، همینجا بمانید و همین حالا In-memory Web API را اضافه کنید.

پکیج In-memory Web API را از npm نصب کنید.

npm install angular-in-memory-web-api –save

HttpClientInMemoryWebApiModule و کلاس InMemoryDataServiceرا که تا لحظاتی دیگر آن را ایجاد خواهید کرد را import کنید.

src/app/app.module.ts (In-memory Web API imports)

```
import { HttpClientInMemoryWebApiModule } from 'angular-in-memory-web-api';

import { InMemoryDataService }  from './in-memory-data.service';
```

پس از اینکه HttpClient را import کردید،و همزمان با پیکربندی آن با InMemoryDataService، HttpClientInMemoryWebApiModule را به آرایه @NgModule.imports اضافه کنید.

```
HttpClientModule,

// The HttpClientInMemoryWebApiModule module intercepts HTTP requests

// and returns simulated server responses.

// Remove it when a real server is ready to receive requests. HttpClientInMemoryWebApiModule.forRoot(
InMemoryDataService, { dataEncapsulation: false }

)
```

متد پیکربندی forRoot() ، کلاس InMemoryDataService را دریافت می کند و با این کار موتور in-memory database را روشن می کند.

نمونه سTour Heroes ، چنین کلاسی را ایجاد می کند : src/app/in-memory-data.service.ts که شامل کدهای زیر است :

src/app/in-memory-data.service.ts

```
import { InMemoryDbService } from 'angular-in-memory-web-api';



export class InMemoryDataService implements InMemoryDbService {

 createDb() {
```

```
    const heroes = [
     { id: 11, name: 'Mr. 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' }
    ];
    return {heroes};
  }
}
```

این فایل جای mock-heroes.ts را می گیرد که حالا دیگر می توانید با خیال راحت آن را پاک کنید. زمانی که سرورتان حاضر شد، in-memory web api را جدا کنید تا درخواست های نرم افزارتان به سمت سرور بروند. حالا برگردیم سر داستان HttpClient .

آموزش Hero ها و HTTP

برخی از نشان های HTTP که مورد نیازتان است را import کنید :

src/app/hero.service.ts (import HTTP symbols)

```
import { HttpClient, HttpHeaders } from '@angular/common/http';
```

HttpClient را داخل Constructor در ویژگی خصوصی ای به نام http تزریق کنید.

```
constructor( private http: HttpClient, private messageService: MessageService) { }
```

همچنان به تزریق MessageService ادامه دهید. آنقدر زیاد این سرویس را فراخوانی خواهید کرد که بهتر است آن را در متد خصوصی log قرار دهید.

```
/** Log a HeroService message with the MessageService */
 private log(message: string) {
 this.messageService.add(`HeroService: ${message}`);
}
```

heroesUrl را با آدرس منبع hero های (heroes resource) موجود بر روی سرور تعریف کنید.

```
private heroesUrl = 'api/heroes'; // URL to web api
```

آموزش دریافت heroes با HttpClient

HeroService.getHeroes() فعلی برای برگشت دادن آرایه ای از hero های ساختگی از یک Observable<Hero[]> استفاده می کند.

src/app/hero.service.ts (getHeroes with RxJs 'of()')

```
    getHeroes(): Observable<Hero[]> {
     return of(HEROES);
    }
```

جهت استفاده از HttpClient آن را تبدیل کنید.

```
/** GET heroes from the server */
 getHeroes (): Observable<Hero[]> {
 return this.http.get<Hero[]>(this.heroesUrl)
 }
```

مرورگر را refresh کنید. داده های hero ها باید با موفقیت از سرور ساختگی (mock server)، load شوند.

حالا با قرار دادن http.get به جای of ، نرم افزارتان بدون نیاز به تغییر دیگر به کار خود ادامه می دهد زیرا مقدار برگشتی هر دو تابع یک Observable<Hero[]> است.

متدهای Http یک مقدار را بر می گردانند.

تمامی متدهای HttpClient ، RxJS observable از چیزی را برگشت می دهند. HTTP پروتکلی مبتنی بر درخواست/پاسخ است. اگر شما درخواستی داشته باشید این پروتکل یک تک پاسخ را برگشت می دهد.

به صورت کلی یک observable به مرور زمان می تواند مقدار متعددی را برگشت دهد. observable های موجود در HttpClient همیشه تنها یک مقدار را برگشت می دهند و پس از تکمیل شدن دیگر این کار را انجام نمی دهند.

تنها در این حالت، فراخوانی HttpClient.get باعث برگشت دادن یک Observable<Hero[]> می شود که همان طور که از اسم آن پیداست observable مربوط به آرایه های hero ها است. در عمل این تابع تنها آرایه ای از hero را برگشت می دهد.

HttpClient.get، داده های پاسخی (response data) را برگشت می دهد.

به صورت پیشفرض، HttpClient.get بدنه پاسخ را به صورت شیء JSON بی نوع (untyped) برگشت می دهد. اگر به دنبال شیء نوع دار (typed) هستید می توانید از <hero [ ]> استفاده کنید. شکل و فرم داده JSON توسط data API سرور مشخص می شود. data API مربوط به Tour of Heroes ، داده hero را به صورت یک آرایه برگشت می دهد.

API های دیگر ممکن است داده هایی که شما داخل یک شیء نیاز دارید را دفن کنند . برای اینکه بتوانید این داده ها را از زیر خاک بیرون بکشید باید با اپراتور RxJs map ، نتیجه observable را پردازش کنید. با وجود اینکه در این بخش به این مبحث نپرداخته ایم، مثالی از map در متد ( )getHeroNo404 در سورس کد نمونه وجود دارد.

آموزش رفع خطاها (Error)

اشتباه همیشه رخ می دهد مخصوصا زمانی که داده های خود را از یک remote server بگیرید. متد ( )HeroService.getHeroes باید خطاها را گرفته و راه حل مناسبی برای آن ها بیندیشد. برای این کار شما باید نتیجه observable را از طریق اپراتور ( ) RxJS catchError، pipe کنید. نشان catchError را همراه با دیگر اپراتورهایی که بعدا به آن ها نیاز خواهید داشت را از rxjs/operators ، import کنید.

```
import { catchError, map, tap } from 'rxjs/operators';
```

حالا نتیجه observable را با متد ()pipe. گسترش دهید و به آن اپراتور ( )catchError را اضافه کنید.

```
getHeroes (): Observable<Hero[]> {

 return this.http.get<Hero[]>(this.heroesUrl)

.pipe(
```

```
 catchError(this.handleError('getHeroes', []))
);
}
```

اپراتور catchError() سد راه observable هایی می شود که fail شده اند. این اپراتور به خطا، error handler ای می دهد که می تواند کارهای مورد نیاز را بر روی آن خطا انجام دهد.

متد handleError() زیر، خطا را گزارش می دهد و سپس نتیجه ای بی ضرر (innocuous) را گزارش می دهد به گونه ای که نرم افزار همچنان بتواند به کار خود ادامه دهد.

آموزش handlerError

errorHandler() توسط خیلی از متدهای HeroService به اشتراک گذاشته خواهد شد به همین دلیل می تواند پاسخگوی بسیاری از نیازهای این متدها باشد.

این تابع به جای حل مستقیم خطا، یک تابع error handler را به catchError ارسال می کند که هم با اسم عملیاتی که fail شده است و هم با یک مقدار برگشتی امن (safe return value) پیکربندی شده است.

```
/**
* 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);
};
}
```

این handler پس از اینکه خطا را به console گزارش داد، پیامی کاربرپسند را ساخته و مقدار برگشتی امنی را به نرم افزار ارسال می کند به گونه ای که نرم افزار از کار نیفتد.

با توجه به اینکه نوع نتیجه observable هر یک از service method ها متفاوت هستند، errorHandler () پارامتر نوع (type parameter) را دریافت می کند به گونه ای که بتواند مقدار امن را به گونه ای برگشت دهد که مورد انتظار نرم افزار است.

آموزش استفاده از اپراتور tap در observable

متدهای HeroService ارتباط محکمی با جریان مقادیر observable برقرار می کنند و (از طریق log()) به بخش پیام ها در پایین صفحه، پیامی ارسال می کنند. این متدها این کار را از طریق اپراتور RxJS tap انجام می دهند. این اپراتور به مقادیر observable نگاهی می اندازد، کاری با این مقادیر انجام می دهد و سپس آن ها را رد می کند. برگشت tap تماسی با خود مقادیر ندارد.

نسخه نهایی getHeroes با tap را که وظیفه log گرفتن از عملیات را دارد را می توانید در زیر مشاهده کنید:

```
/** GET heroes from the server */
getHeroes (): Observable<Hero[]> {
 return this.http.get<Hero[]>(this.heroesUrl)
 .pipe(
 tap(heroes => this.log('fetched heroes')),
catchError(this.handleError('getHeroes', []))
 );
}
```

آموزش دریافت hero از طریق id

اغلب web API ها، از درخواست دریافت از طریق id به شکل api/hero/:id پشتیبانی می کنند (مانند api/hero/11) .

برای انجام این کار متد HeroService.getHero() را اضافه کنید:

```
/** GET hero by id. Will 404 if id not found */
 getHero(id: number): Observable<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}`))
);
}
```

در کد بالا سه تفاوت مشهود با getHeroes() وجود دارد.

- این کد request URL را با آیدی hero مورد نظر می سازد
- سرور به جای آرایه ای از hero ها، باید با تنها یک hero پاسخ دهد.
- به همین دلیل، getHero به جای برگشت دادن observable آرایه های hero ، یک observable<Hero> (Observable اشیای hero) را برگشت می دهد.

آموزش به روز کردن hero ها

ویرایش اسم hero در hero detail veiw . همزمان با تایپ کردن شما، اسم hero ها در سربرگ بالای صفحه آپدیت می شوند. اما زمانی که بر روی دکمه برگشت به صفحه قبل کلیک کنید، این تغییرات از بین می روند. اگر بخواهید که این تغییرات ماندگار باشند، باید آن ها را در سرور بنویسید .

انتهای قالب hero detail ، با click event binding ، دکمه save را اضافه کنید. پس از اضافه کردن آن متد کامپوننت جدیدی به نام save() ایجاد می شود.

src/app/hero-detail/hero-detail.component.html (save)

```
<button (click)="save()">save</button>
```

متد save() زیر را اضافه کنید. وظیفه این متد ماندگار کردن تغییرات اسم hero ها از طریق متد hero servcie updateHero() است که پس از انجام این کار به view قبلی برمیگردد.

src/app/hero-detail/hero-detail.component.ts (save)

```
save(): void {
  this.heroService.updateHero(this.hero)
    .subscribe(() => this.goBack());
}
```

اضافه کردن HeroService.updateHero()

ساختار کلی متد updateHero() شبیه به getHeroes() است، با این تفاوت که برای ماندگار کردن hero ی تغییر کرده از http.put() استفاده می کند.

src/app/hero.service.ts (update)

```
/** PUT: update the hero on the server */
updateHero (hero: Hero): Observable<any> {
 return this.http.put(this.heroesUrl, hero, httpOptions).pipe(
  tap(_ => this.log(`updated hero id=${hero.id}`)),
  catchError(this.handleError<any>('updateHero'))
 );
}
```

متد HttpClient.put() سه پارامتر را دریافت می کند

- URL
- داده برای آپدیت کردن (در این حالت hero اصلاح شده)
- options

URL بدون تغییر باقی می ماند. Web API مربوط به hero ها با نگاه به آیدی hero ها می دانند کدام hero را به روز کنند.

این web API ، توقع یک هدر خاص در HTTP save request را دارد. این header در httpOptions قرار داشته و با مقداری ثابت در HeroService تعریف شده است.

```
const httpOptions = {
 headers: new HttpHeaders({ 'Content-Type': 'application/json' })
};
```

مرورگر را refresh کرده، نام Hero را تغییر دهید و تغییرات خود را ذخیره کنید، سپس بر روی دکمه go back کلیک کنید. حالا hero در لیستی با اسم های تغییر کرده ظاهر می شود.

آموزش اضافه کردن hero جدید

برای اضافه کردن یک hero جدید، این نرم افزار تنها به نام hero نیاز دارد. می توانید از المان input استفاده کنید به گونه ای که با دکمه add جفت شده باشد.

کد زیر را درست پس از heading در قالب HeroesComponent وارد کنید :

src/app/heroes/heroes.component.html (add)

```
<div>

 <label>Hero name:

  <input #heroName />

 </label>

 <!-- (click) passes input value to add() and then clears the input -->

 <button (click)="add(heroName.value); heroName.value=''">

  add

 </button>

</div>
```

در پاسخ به یک رویداد کلیک (click event) ، click handler مربوط به کامپوننت را فراخوانی کنید و سپس بخش input را خالی کنید به گونه ای که برای اسم بعدی آماده شود.

src/app/heroes/heroes.component.ts (add)

```
add(name: string): void {

 name = name.trim();

 if (!name) { return; }

 this.heroService.addHero({ name } as Hero)

  .subscribe(hero => {

   this.heroes.push(hero);
```

```
    });
  }
```

زمانی که اسم داده شده خالی نباشد، این handler شیئی را از نام hero و مانند hero ایجاد می کند (تنها چیزی که کم دارد id است) و آن را به متد addHero() ارسال می کند.

زمانی که addHero با موفقیت ذخیره شود، برگشت subscribe ، hero جدید را تحویل گرفته و برای نمایش دادن آن، آن را به لیست hero ها ارسال می کند.

در بخش بعد به نوشتن HeroService.addHero() خواهید پرداخت.

آموزش اضافه کردن HeroService.addHero()

متد addHero() زیر را به کلاس HeroService اضافه کنید.

src/app/hero.service.ts (addHero)

```
/** POST: add a new hero to the server */
addHero (hero: Hero): Observable<Hero> {
 return this.http.post<Hero>(this.heroesUrl, hero, httpOptions).pipe(
  tap((hero: Hero) => this.log(`added hero w/ id=${hero.id}`)),
  catchError(this.handleError<Hero>('addHero'))
 );
}
```

این متد از دو جهت با updateHero تفاوت دارد.

- این متد به جای put() از HttpClient.post() استفاده می کند.
- این متد از سرور انتظار تولید آیدی برای Hero جدید را دارد که در این حالت observable<Hero> را به فراخواننده بر می گرداند.

مرورگر را refresh کنید و تعدادی hero اضافه کنید.

آموزش پاک کردن یک hero

هر یک از hero ها در لیست باید یک دکمه delete داشته باشد.

المان دکمه زیر را پس از اسم hero در المان تکرار شده <li> به قالب HeroesComponent اضافه کنید.

```
<button class="delete" title="delete hero"
 (click)="delete(hero)">x</button>
```

HTML مربوط به لیست hero ها مانند کد زیر باید بشود :

src/app/heroes/heroes.component.html (list of heroes)

```
<ul class="heroes">
 <li *ngFor="let hero of heroes">
  <a routerLink="/detail/{{hero.id}}">
   <span class="badge">{{hero.id}}</span> {{hero.name}}
  </a>
  <button class="delete" title="delete hero"
   (click)="delete(hero)">x</button>
 </li>
</ul>
```

برای اینکه مکان دکمه delete را در سمت راست ورودی hero قرار دهید، مقداری کد CSS به heroes.component.css اضافه کنید. می توانید این کد را در بازبینی نهایی کد پیدا کنید.

Delete () handler را به کامپوننت اضافه کنید.

src/app/heroes/heroes.component.ts (delete)

```
delete(hero: Hero): void {
 this.heroes = this.heroes.filter(h => h !== hero);
```

```
    this.heroService.deleteHero(hero).subscribe();
  }
```

با وجود اینکه این کامپوننت، پاک کردن Hero را به HeroService محول می کند، اما همچنان آپدیت کردن لیست hero های خود مسئول است. متد ()delete این کامپوننت بلافاصله hero-to-delete را از آن لیست پاک می کند با این فرض که HeroService از سرور موفق بیرون می آید.

کامپوننت کار خاصی با observable برگشتی توسط ()heroService.delete ندارد. به هر حال subscribe کردن راهی است که باید برود.

اگر ()subscribe را از قلم بیندازید ، در این صورت سرویس درخواست delete را به سرور ارسال نمی کند! طبق قانون، یک observable کاری نمی کند تا اینکه چیزی subscribe شود!

می توانید برای امتحان موقتا ()subscribe را پاک کنید سپس بر روی dashboard و Heroes کلیک کنید تا لیست hero ها را دوباره ببینید.

آموزش اضافه کردن ()HeroService.deleteHero

به شیوه زیر متد ()deleteHero را اضافه کنید.

src/app/hero.service.ts (delete)

```
  /** 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, httpOptions).pipe(
      tap(_ => this.log(`deleted hero id=${id}`)),
      catchError(this.handleError<Hero>('deleteHero'))
    );
```

```
}
```

توجه داشته باشید که:

- این کد HttpClient.delete را فراخوانی می کند.
- URL مذکور مجموع heroes resource URL و آیدی hero مد نظر برای پاک کردن است.
- ارسال داده مانند زمانی نیست که با put و post کار می کردید.
- همچنان httpOptions را ارسال می کنید.

مرورگر را refresh کنید و قابلیت جدید delete کردن را امتحان کنید.

آموزش جستجو با اسم

در این تمرین آخر شما خواهید آموخت که اپراتورهای observable را به یکدیگر زنجیر کنید تا بتوانید تعداد درخواست های HTTP مشابه را به حداقل برسانید و مصرف پهنای باند را به حداقل برسانید.

شما قابلیت جستجوی hero ها را به داشبورد اضافه خواهید کرد. پس از اینکه کاربرد اسمی را درون search box تایپ می کند، شما باید درخواست های HTTP تکراری را برای آن hero ها را ایجاد کنید به گونه ای که این عمل بر اساس اسم وارد شده فیلتر شده باشد. هدف شما تنها این است که به اندازه نیاز درخواست HTTP صادر کنید.

آموزش HeroService.searchHeroes

کار خود را با اضافه کردن متد searchHeroes به HeroService شروع کنید.

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(_ => this.log(`found heroes matching "${term}"`)),
  catchError(this.handleError<Hero[]>('searchHeroes', []))
 );
}
```

اگر عبارتی برای جستجو کردن وجود نداشته باشد این متد بلافاصله آرایه ای خالی را برگشت می دهد. دیگر بخش های این متد تا حد زیادی شبیه به getHeroes() هستند. تنها تفاوت مشهود بین این دو URL است که در این حالت در کنار search term از query string استفاده شده است.

آموزش اضافه کردن جستجو به داشبورد

قالب DashboardComponent را باز کرده و المان جستجوی Hero (<app-hero-search>) را به پایین قالب DashboardComponent اضافه کنید.

src/app/dashboard/dashboard.component.html

```
<h3>Top Heroes</h3>
<div class="grid grid-pad">
 <a *ngFor="let hero of heroes" class="col-1-4"
    routerLink="/detail/{{hero.id}}">
  <div class="module hero">
   <h4>{{hero.name}}</h4>
  </div>
 </a>
</div>
<app-hero-search></app-hero-search>
```

این قالب تا حد زیادی شبیه به تکرارکننده *ngFor در قالب HeroesComponent است. متاسفانه اضافه کردن این المان باعث خراب کردن نرم افزار می شود. Angular نمی تواند کامپوننتی پیدا کند که selector آن با <app-hero-search> مطابقت داشته باشد.

HeroSearchComponent هنوز وجود ندارد. این مشکل را خودتان حل کنید.

آموزش ایجاد HeroSearchComponentlink

با CLI یک HeroSearchComponent را ایجاد کنید.

```
ng generate component hero-search
```

در این حالت CLI سه HeroSearchComponent را تولید می کند و این کامپوننت را به اعلان های AppModule اضافه می کند.

قالب HeroSearchComponent تولید شده را جایگزین یک text box و لیستی از نتایج جستجوی مطابق کنید (مانند زیر)

src/app/hero-search/hero-search.component.html

```
<div id="search-component">
  <h4>Hero Search</h4>

  <input #searchBox id="search-box" (keyup)="search(searchBox.value)" />

  <ul class="search-result">
    <li *ngFor="let hero of heroes$ | async" >
      <a routerLink="/detail/{{hero.id}}">
        {{hero.name}}
      </a>
    </li>
  </ul>
</div>
```

همانطور که در بازبینی نهایی کد نشان داده شده است، یک استایل CSS خصوصی را به hero-search.component.css اضافه کنید.

پس از اینکه کاربر در کادر جستجو تایپ می کند، یک keyup event binding متد search() مربوط به کامپوننت را با مقادیر جدید کادر جستجو فراخوانی می کند.

آموزش AsyncPipe

همانطور که می دانید، ngFor* اشیای hero را تکرار می کند. به دقت نگاه کنید تا ببینید که ngFor* در لیستی به نام heroes$ و نه heroes به تکرار می پردازد.

```
<li *ngFor="let hero of heroes$ | async" >
```

علامت $ ، قراردادی است که نشان می دهد heroes$ یک observable هست نه یک آرایه.

ngFor* کاری با observable نمی تواند انجام دهد. در ادامه کاراکتر pipe (|) را به همراه async می توانید مشاهده کنید که بیانگر asyncPipe مربوط به Angular است.

AsyncPipe به صورت خودکار در یک observable ، subscribe می کند تا دیگر شما مجبور به انجام این کار در کلاس کامپوننت نباشید.

آموزش Fix کردن HeroSearchComponent class

مانند زیر کلاس HeroSearchComponent و متادیتای تولید شده را جایگزین کنید.

src/app/hero-search/hero-search.component.ts

```
1.  import { Component, OnInit } from '@angular/core';
2.
3.  import { Observable, Subject } from 'rxjs';
4.
5.  import {
6.    debounceTime, distinctUntilChanged, switchMap
7.  } from 'rxjs/operators';
8.
9.  import { Hero } from '../hero';
10. import { HeroService } from '../hero.service';
11.
12. @Component({
13.  selector: 'app-hero-search',
14.  templateUrl: './hero-search.component.html',
15.  styleUrls: [ './hero-search.component.css' ]
16. })
17. export class HeroSearchComponent implements OnInit {
```

```
18.  heroes$: Observable<Hero[]>;
19.  private searchTerms = new Subject<string>();
20.
21.  constructor(private heroService: HeroService) {}
22.
23.  // Push a search term into the observable stream.
24.  search(term: string): void {
25.   this.searchTerms.next(term);
26.  }
27.
28.  ngOnInit(): void {
29.   this.heroes$ = this.searchTerms.pipe(
30.    // wait 300ms after each keystroke before considering the term
31.    debounceTime(300),
32.
33.    // ignore new term if same as previous term
34.    distinctUntilChanged(),
35.
36.    // switch to new search observable each time the term changes
37.    switchMap((term: string) => this.heroService.searchHeroes(term)),
38.   );
39.  }
40. }
```

به اعلان heroes$ به عنوان یک observable دقت کنید.

```
heroes$: Observable<Hero[]>;
```

این کد در ngOnInit تنظیم (set) می شود.

آموزش The searchTerms RxJS subject

ویژگی searchTerms به عنوان یک RxJS Subject اعلان می شود.

```
private searchTerms = new Subject<string>();

// Push a search term into the observable stream.
```

```
search(term: string): void { this.searchTerms.next(term);
}
```

یک Subject هم منبعی از مقادیر observable و هم خود observable است. subscribe کردن در subject تفاوتی با subscribe کردن در observable ندارد.

همچنین می توانید مانند متد search () با فراخوانی متد next(value) آن، مقادیر را درون این observable فشار دهید.

متد search() از طریق event binding به رخداد keystroke مربوط به کادر جستجو فراخوانی می شود.

```
<input #searchBox id="search-box" (keyup)="search(searchBox.value)" />
```

هر زمان که کاربر در textbox تایپ کند، binding، search () را با مقدار textbox فراخوانی می کند (« یک عبارت جستجو»). SearchTerms تبدیل به یک Observable شده و جریان پایداری از عبارات جستجو را از خود بیرون می دهد.

آموزش زنجیر کردن اپراتورهای RxJS

رد کردن مستقیم عبارات جستجو به searchHeroes() بعد از هر keystroke کاربر باعث ایجاد مقدار زیادی از HTTP request می شود که در ادامه سنگین شدن منابع سرور سایت و هدر رفتن شبکه سلولی اینترنت می شود.

در عوض، متد ngOnInit() از طریق یک سری از اپراتورهای RxJs ، searchTerms observable را pipe می کند که باعث کاهش تعداد فراخوانی ها به searchHeroes() و در نهایت برگشت دادن یک observable نتایج جستجوی مربوط به hero ها در زمان مناسب می شود (هر کدام یک Hero[]) .

در زیر می توانید کد را مشاهده کنید.

```
this.heroes$ = this.searchTerms.pipe(
// wait 300ms after each keystroke before considering the term debounceTime(300),
// ignore new term if same as previous term distinctUntilChanged(),
// switch to new search observable each time the term changes switchMap((term: string) =>
this.heroService.searchHeroes(term)),
);
```

- debounceTime(300) پیش از رد کردن آخرین رشته ، تا زمانی که جریان رویدادهای رشته ای جدید (new string events) به مدت 300 میلی ثانیه متوقف شود ، صبر می کند. در این حالت شما نمی توانید بیشتر از 300 میلی ثانیه درخواست ایجاد کنید.

- distinctUntilChanged() تضمین می کند که تنها در صورتی درخواست ارسال شود که متن فیلتر (filter text) تغییر کرده باشد .
- switchMap() به ازای هر یک از عبارات جستجویی که از پس دو مورد بالا برآمده باشند، service search را فراخوانی می کند. این تابع observable های قبلی جستجو را لغو می کند و تنها آخرین search service observable را برگشت می دهد.

در switchMap operator ، تمامی key event های دارای صلاحیت می توانند باعث فراخوانی متد HttpClient.get() شوند. حتی اگر بین درخواست ها 300 میلی ثانیه فاصله باشد، باز هم باید منتظر تعدادی زیادی از درخواست HTTP باشید که حتی ممکن است بازگشت آن ها به ترتیب حالت ارسالی آن ها صورت نگیرد.

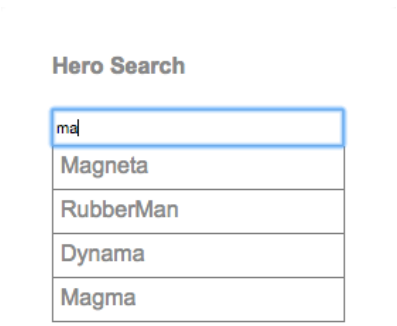
switchMap() وظیفه حفظ ترتیب اصلی درخواست ها و در عین حال برگشت دادن observable هایی را برعهده دارد که توسط آخرین متد HTTP فراخوانی شده باشند. نتایج حاصل از فراخوانی های پیشین لغو و متروک می شوند.

توجه داشته باشید که لغو کردن observable پیشین searchHeroes() باعث لغو کردن درخواست های HTTP در حال انتظار نمی شود. نتایج ناخواسته به راحتی پیش از رسیدن به کد نرم افزارتان دور ریخته می شوند.

یادتان باشد که کلاس کامپوننت در heroes$ observable، subscribe نمی کند. این وظیفه بر عهده AsyncPipe موجود در قالب است.

نتیجه را امتحان کنید

مجددا نرم افزار را اجرا کنید. در داشبورد متنی را به دلخواه وارد کنید. در صورتی که کاراکتری را وارد کنید که با یکی از اسامی hero ها مطابقت داشته باشد، نتیجه ای مانند زیر را باید بگیرید :



بازبینی نهایی کد

برنامه ی شما باید مانند این لینک باشد:

https://angular.io/generated/live-examples/toh-pt6/stackblitz.html

کدهای بحث شده در این صفحه را می توانید در ادامه مشاهده کنید. (تمامی این کدها در پوشه ی src/app/ موجود هستند.)

HeroService, *InMemoryDataService*, *AppModule*

hero.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';

import { Observable, of } from 'rxjs';
import { catchError, map, tap } from 'rxjs/operators';

import { Hero } from './hero';
import { MessageService } from './message.service';

const httpOptions = {
headers: new HttpHeaders({ 'Content-Type': 'application/json' })
};

@Injectable({ providedIn: 'root' })
export class HeroService {

private heroesUrl = 'api/heroes'; // URL to web api

constructor(
private http: HttpClient,
private messageService: MessageService) { }

/** GET heroes from the server */
getHeroes (): Observable<Hero[]> {
return this.http.get<Hero[]>(this.heroesUrl)
.pipe(
tap(heroes => this.log('fetched heroes')),
catchError(this.handleError('getHeroes', []))
);
}
```

```
31.
32. /** GET hero by id. Return `undefined` when id not found */
33. getHeroNo404<Data>(id: number): Observable<Hero> {
34. const url = `${this.heroesUrl}/?id=${id}`;
35. return this.http.get<Hero[]>(url)
36. .pipe(
37. map(heroes => heroes[0]), // returns a {0|1} element array
38. tap(h => {
39. const outcome = h ? `fetched` : `did not find`;
40. this.log(`${outcome} hero id=${id}`);
41. }),
42. catchError(this.handleError<Hero>(`getHero id=${id}`))
43. );
44. }
45.
46. /** GET hero by id. Will 404 if id not found */
47. getHero(id: number): Observable<Hero> {
48. const url = `${this.heroesUrl}/${id}`;
49. return this.http.get<Hero>(url).pipe(
50. tap(_ => this.log(`fetched hero id=${id}`)),
51. catchError(this.handleError<Hero>(`getHero id=${id}`))
52. );
53. }
54.
55. /* GET heroes whose name contains search term */
56. searchHeroes(term: string): Observable<Hero[]> {
57. if (!term.trim()) {
58. // if not search term, return empty hero array.
59. return of([]);
60. }
61. return this.http.get<Hero[]>(`${this.heroesUrl}/?name=${term}`).pipe(
62. tap(_ => this.log(`found heroes matching "${term}"`)),
63. catchError(this.handleError<Hero[]>('searchHeroes', []))
64. );
65. }
66.
```

```
67. //////// Save methods //////////
68.
69. /** POST: add a new hero to the server */
70. addHero (hero: Hero): Observable<Hero> {
71. return this.http.post<Hero>(this.heroesUrl, hero, httpOptions).pipe(
72. tap((hero: Hero) => this.log(`added hero w/ id=${hero.id}`)),
73. catchError(this.handleError<Hero>('addHero'))
74. );
75. }
76.
77. /** DELETE: delete the hero from the server */
78. deleteHero (hero: Hero | number): Observable<Hero> {
79. const id = typeof hero === 'number' ? hero : hero.id;
80. const url = `${this.heroesUrl}/${id}`;
81.
82. return this.http.delete<Hero>(url, httpOptions).pipe(
83. tap(_ => this.log(`deleted hero id=${id}`)),
84. catchError(this.handleError<Hero>('deleteHero'))
85. );
86. }
87.
88. /** PUT: update the hero on the server */
89. updateHero (hero: Hero): Observable<any> {
90. return this.http.put(this.heroesUrl, hero, httpOptions).pipe(
91. tap(_ => this.log(`updated hero id=${hero.id}`)),
92. catchError(this.handleError<any>('updateHero'))
93. );
94. }
95.
96. /**
97. * Handle Http operation that failed.
98. * Let the app continue.
99. * @param operation - name of the operation that failed
100.          * @param result - optional value to return as the observable result
101.          */
102.          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}`);
        }
        }
```

in-memory-data.service.ts

```
import { InMemoryDbService } from 'angular-in-memory-web-api';

export class InMemoryDataService implements InMemoryDbService {
createDb() {
const heroes = [
{ id: 11, name: 'Mr. 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' }
```

```
16. ];
17. return {heroes};
18. }
19. }
```

app.module.ts

```
1.  import { NgModule } from '@angular/core';
2.  import { BrowserModule } from '@angular/platform-browser';
3.  import { FormsModule } from '@angular/forms';
4.  import { HttpClientModule } from '@angular/common/http';
5.
6.  import { HttpClientInMemoryWebApiModule } from 'angular-in-memory-web-api';
7.  import { InMemoryDataService } from './in-memory-data.service';
8.
9.  import { AppRoutingModule } from './app-routing.module';
10.
11. import { AppComponent } from './app.component';
12. import { DashboardComponent } from './dashboard/dashboard.component';
13. import { HeroDetailComponent } from './hero-detail/hero-detail.component';
14. import { HeroesComponent } from './heroes/heroes.component';
15. import { HeroSearchComponent } from './hero-search/hero-search.component';
16. import { MessagesComponent } from './messages/messages.component';
17.
18. @NgModule({
19. imports: [
20. BrowserModule,
21. FormsModule,
22. AppRoutingModule,
23. HttpClientModule,
24.
25. // The HttpClientInMemoryWebApiModule module intercepts HTTP requests
26. // and returns simulated server responses.
27. // Remove it when a real server is ready to receive requests.
28. HttpClientInMemoryWebApiModule.forRoot(
29. InMemoryDataService, { dataEncapsulation: false }
```

```
)
],
declarations: [
AppComponent,
DashboardComponent,
HeroesComponent,
HeroDetailComponent,
MessagesComponent,
HeroSearchComponent
],
bootstrap: [ AppComponent ]
})
export class AppModule { }
```

HeroesComponent

heroes/heroes.component.html

```
<h2>My Heroes</h2>

<div>
<label>Hero name:
<input #heroName />
</label>
<!-- (click) passes input value to add() and then clears the input -->
<button (click)="add(heroName.value); heroName.value=''">
add
</button>
</div>

<ul class="heroes">
<li *ngFor="let hero of heroes">
<a routerLink="/detail/{{hero.id}}">
<span class="badge">{{hero.id}}</span> {{hero.name}}
</a>
<button class="delete" title="delete hero"
(click)="delete(hero)">x</button>
```

```
</li>
</ul>
```

heroes/heroes.component.ts

```
import { Component, OnInit } from '@angular/core';

import { Hero } from '../hero';
import { HeroService } from '../hero.service';

@Component({
selector: 'app-heroes',
templateUrl: './heroes.component.html',
styleUrls: ['./heroes.component.css']
})
export class HeroesComponent implements OnInit {
heroes: Hero[];

constructor(private heroService: HeroService) { }

ngOnInit() {
this.getHeroes();
}

getHeroes(): void {
this.heroService.getHeroes()
.subscribe(heroes => this.heroes = heroes);
}

add(name: string): void {
name = name.trim();
if (!name) { return; }
this.heroService.addHero({ name } as Hero)
.subscribe(hero => {
this.heroes.push(hero);
});
```

```
}

delete(hero: Hero): void {
this.heroes = this.heroes.filter(h => h !== hero);
this.heroService.deleteHero(hero).subscribe();
}

}
```

heroes/heroes.component.css

```
/* HeroesComponent's private CSS styles */
.heroes {
margin: 0 0 2em 0;
list-style-type: none;
padding: 0;
width: 15em;
}
.heroes li {
position: relative;
cursor: pointer;
background-color: #EEE;
margin: .5em;
padding: .3em 0;
height: 1.6em;
border-radius: 4px;
}

.heroes li:hover {
color: #607D8B;
background-color: #DDD;
left: .1em;
}

.heroes a {
color: #888;
```

```
text-decoration: none;
position: relative;
display: block;
width: 250px;
}

.heroes a:hover {
color:#607D8B;
}

.heroes .badge {
display: inline-block;
font-size: small;
color: white;
padding: 0.8em 0.7em 0 0.7em;
background-color: #607D8B;
line-height: 1em;
position: relative;
left: -1px;
top: -4px;
height: 1.8em;
min-width: 16px;
text-align: right;
margin-right: .8em;
border-radius: 4px 0 0 4px;
}

button {
background-color: #eee;
border: none;
padding: 5px 10px;
border-radius: 4px;
cursor: pointer;
cursor: hand;
font-family: Arial;
}
```

```
62.
63. button:hover {
64. background-color: #cfd8dc;
65. }
66.
67. button.delete {
68. position: relative;
69. left: 194px;
70. top: -32px;
71. background-color: gray !important;
72. color: white;
73. }
```

HeroDetailComponent

hero-detail/hero-detail.component.html

```
<div *ngIf="hero">
 <h2>{{hero.name | uppercase}} Details</h2>
<div><span>id: </span>{{hero.id}}
</div>
 <div>
<label>name:
<input [(ngModel)]="hero.name" placeholder="name"/>
 </label>
 </div>
 <button (click)="goBack()">go back</button>
<button (click)="save()">save</button>
 </div>
```

hero-detail/hero-detail.component.ts

```
1. import { Component, OnInit, Input } from '@angular/core';
```

```
import { ActivatedRoute } from '@angular/router';
import { Location } from '@angular/common';

import { Hero } from '../hero';
import { HeroService } from '../hero.service';

@Component({
selector: 'app-hero-detail',
templateUrl: './hero-detail.component.html',
styleUrls: [ './hero-detail.component.css' ]
})
export class HeroDetailComponent implements OnInit {
@Input() hero: Hero;

constructor(
private route: ActivatedRoute,
private heroService: HeroService,
private location: Location
) {}

ngOnInit(): void {
this.getHero();
}

getHero(): void {
const id = +this.route.snapshot.paramMap.get('id');
this.heroService.getHero(id)
.subscribe(hero => this.hero = hero);
}

goBack(): void {
this.location.back();
}

save(): void {
this.heroService.updateHero(this.hero)
```

```
38. .subscribe(() => this.goBack());
39. }
40. }
```

HeroSearchComponent

hero-search/hero-search.component.html

```
1. <div id="search-component">
2. <h4>Hero Search</h4>
3.
4. <input #searchBox id="search-box" (keyup)="search(searchBox.value)" />
5.
6. <ul class="search-result">
7. <li *ngFor="let hero of heroes$ | async" >
8. <a routerLink="/detail/{{hero.id}}">
9. {{hero.name}}
10. </a>
11. </li>
12. </ul>
13. </div>
```

hero-search/hero-search.component.ts

```
1. import { Component, OnInit } from '@angular/core';
2.
3. import { Observable, Subject } from 'rxjs';
4.
5. import {
6. debounceTime, distinctUntilChanged, switchMap
7. } from 'rxjs/operators';
8.
9. import { Hero } from '../hero';
10. import { HeroService } from '../hero.service';
11.
```

```
@Component({
selector: 'app-hero-search',
templateUrl: './hero-search.component.html',
styleUrls: [ './hero-search.component.css' ]
})
export class HeroSearchComponent implements OnInit {
heroes$: Observable<Hero[]>;
private searchTerms = new Subject<string>();

constructor(private heroService: HeroService) {}

// Push a search term into the observable stream.
search(term: string): void {
this.searchTerms.next(term);
}

ngOnInit(): void {
this.heroes$ = this.searchTerms.pipe(
// wait 300ms after each keystroke before considering the term
debounceTime(300),

// ignore new term if same as previous term
distinctUntilChanged(),

// switch to new search observable each time the term changes
switchMap((term: string) => this.heroService.searchHeroes(term)),
);
}
}
```

hero-search/hero-search.component.css

```
/* HeroSearch private styles */
.search-result li {
border-bottom: 1px solid gray;
border-left: 1px solid gray;
```

```
5.  border-right: 1px solid gray;
6.  width:195px;
7.  height: 16px;
8.  padding: 5px;
9.  background-color: white;
10. cursor: pointer;
11. list-style-type: none;
12. }
13.
14. .search-result li:hover {
15. background-color: #607D8B;
16. }
17.
18. .search-result li a {
19. color: #888;
20. display: block;
21. text-decoration: none;
22. }
23.
24. .search-result li a:hover {
25. color: white;
26. }
27. .search-result li a:active {
28. color: white;
29. }
30. #search-box {
31. width: 200px;
32. height: 20px;
33. }
34.
35.
36. ul.search-result {
37. margin-top: 0;
38. padding-left: 0;
39. }
```

خلاصه

حالا به انتهای راه رسیده اید و تا به این لحظه چیزهای زیادی آموخته اید مانند:

- برای استفاده از HTTP در نرم افزار، وابستگی های مورد نیاز را اضافه کردید.
- برای بارگیری hero ها از یک web API، HeroService را ریفاکتور کردید.
- برای پشتیبانی از متدهای ()post، ()put و ()delete، HeroService را گسترش دادید.
- برای اضافه کردن، ویرایش کردن و پاک کردن hero ها کامپوننت ها را آپدیت کردید.
- in-memory web API را پیکربندی کردید.
- نحوه ی استفاده از ovservable ها را آموختید.

این چکیده ای بود از آموزش Tour of Heroes در این نقطه شما آماده اید تا در بخش اصول بنیادین Angular (fundamentals) در رابطه با Angular اطلاعات بیشتری کسب کنید.

در این بخش کار خود را می توانید از بخش معماری آغاز کنید.