HTTP – Récupérer & Enregistrer Les données

Dans cet article, nous allons ajouter les fonctionnalités de persistance des données suivantes avec l'aide de *HttpClient* d'Angular

- Le *HeroService* obtient des données de héros avec des requêtes HTTP.
- Les utilisateurs peuvent ajouter, modifier et supprimer des héros et enregistrer ces changements sur HTTP.
- Les utilisateurs peuvent rechercher des héros par leur nom.

Lorsque vous aurez terminé avec cette page, l'application devrait ressembler à cet ***exemple en direct*** / ***téléchargement de l'exemple***.

Activer Les Services HTTP

HttpClient est le mécanisme d'Angular pour communiquer avec un serveur distant via HTTP.

Pour rendre *HttpClient* disponible partout dans l'application,

- ouvrez la racine *AppModule*
- importer le symbole *HttpClientModule* depuis *@angular/common/http*,
- ajoutez-le au tableau *@NgModule.imports*

Simuler Le Serveur Des Données

Cet exemple de tutoriel imite la communication avec un serveur de données distant en utilisant le module***In-memory Web API***.

Après l'installation du module, l'application va effectuer des demandes et recevoir des réponses du *HttpClient* sans savoir que l'API Web en mémoire intercepte ces demandes, les applique à un magasin de données en mémoire et renvoie des réponses simulées.

Cette facilité est une grande commodité pour le tutoriel. Vous n'aurez pas à configurer un serveur pour en savoir plus sur*HttpClient*.

Il peut également s'avérer utile dans les premières étapes du développement de votre propre application lorsque l'API Web du serveur est mal définie ou n'est pas encore implémentée.

Important: le module *In-memory Web API* n'a rien à voir avec HTTP dans Angular.

Si vous lisez ce tutoriel pour en savoir plus sur *HttpClient*, vous pouvez ignorer cette étape. Si vous codez avec ce didacticiel, restez ici et ajoutez *In-memory Web API* maintenant.

Installez le package*In-memory Web API* à partir de *npm*

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

Importez le *HttpClientInMemoryWebApiModule* et la classe*InMemoryDataService*, que vous allez créer dans un instant.

```
src/app/app.module.ts (importations de In-memory Web API)


import { HttpClientInMemoryWebApiModule } from 'angular-in-memory-web-api';
import { InMemoryDataService }  from './in-memory-data.service';
```

Ajoutez le*HttpClientInMemoryWebApiModule*au tableau*@NgModule.imports* après avoir importé*HttpClient*, —tout en le configurant avec*InMemoryDataService*.

```
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 }
)
```

La méthode de configuration *forRoot()* prend une classe*InMemoryDataService*qui initialise la base de données en mémoire.

L'exemple Tour of Heroes crée une telle classe *src/app/in-memory-data.service.ts* qui a le contenu suivant:

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};
  }
}
```

Ce fichier remplace *mock-heroes.ts*, qui peut maintenant être supprimé en toute sécurité.

Lorsque votre serveur est prêt, détachez*In-memory Web API*et les demandes de l'application seront transmises au serveur.

Maintenant, revenons à l'histoire de *HttpClient*.

Héros et HTTP

Importez quelques symboles HTTP dont vous aurez besoin:

src/app/hero.service.ts (importation des symboles HTTP)

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

Injecter *HttpClient* dans le constructeur dans une propriété privée appelée *http*.

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

Continuez à injecter le *MessageService*. Vous l'appelez si souvent que vous l'intégrez dans la méthode du journal *log* privé.

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

```
}
```

Définissez *heroesUrl* avec l'adresse de la ressource heroes sur le serveur.

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

Obtenez des héros avec HttpClient

L'actuel *HeroService.getHeroes()* utilise la fonction *RxJS of()* pour renvoyer un tableau de faux héros en tant que Observable *<Héros []>*.

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


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

Convertir cette méthode pour utiliser *HttpClient*

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

Actualisez le navigateur Les données de héros doivent être chargées avec succès depuis le serveur simulé.

Vous avez échangé *of* pour *http.get* et l'application continue à fonctionner sans aucune autre modification car les deux fonctions renvoient un *Observable<Hero[]>*.

Les Méthodes Http Renvoient Une Valeur

Toutes les méthodes *HttpClient* retournent un RxJS *Observable* de quelque chose.

HTTP est un protocole de requête/réponse. Vous faites une requête, elle renvoie une seule réponse.

En général, un observable peut renvoyer plusieurs valeurs dans le temps. Un observable de *HttpClient* émet toujours une seule valeur, puis se termine, pour ne plus jamais émettre.

Cet appel *HttpClient.get* particulier renvoie un O*bservable<Hero []>*, littéralement « un observable des tableaux de héros ». En pratique, il ne retournera qu'un seul tableau de héros.

HttpClient.get Renvoie Les Données De Réponse

HttpClient.get renvoie le corps de la réponse en tant qu'objet JSON non typé par défaut. L'application du spécificateur de type facultatif, *<Hero[]>*, vous donne un objet résultat typé.

La forme des données JSON est déterminée par l'API de données du serveur. L'API de données Tour of Heroes renvoie les données du héros sous la forme d'un tableau.

D'autres API peuvent enterrer les données que vous souhaitez dans un objet. Vous devrez peut-être extraire ces données en traitant le résultat *Observable* avec l'opérateur de carte RxJS.

Bien que cela ne soit pas abordé ici, il existe un exemple de *map* dans la méthode *getHeroNo404()* incluse dans l'exemple de code source.

La Gestion Des Erreurs

Les choses tournent mal, surtout lorsque vous recevez des données d'un serveur distant. La méthode *HeroService.getHeroes()* doit détecter les erreurs et faire quelque chose d'approprié.

Pour détecter les erreurs, vous « canaliserez » le résultat observable de *http.get()* via un opérateur RxJS *catchError()*.

Importez le symbole *catchError* à partir de *rxjs/operators*, ainsi que d'autres opérateurs dont vous aurez besoin plus tard.

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

Étendez maintenant le résultat observable avec la méthode *.pipe()* et donnez-lui un opérateur *catchError()*.

```
getHeroes (): Observable<Hero[]> {
  return this.http.get<Hero[]>(this.heroesUrl)
    .pipe(
      catchError(this.handleError('getHeroes', []))
    );
}
```

L'opérateur *catchError()* intercepte un Observable qui a échoué. Il passe l'erreur à un gestionnaire d'erreur qui peut faire ce qu'il veut avec l'erreur.

La méthode *handleError()* suivante signale l'erreur et renvoie un résultat inoffensif pour que l'application continue de fonctionner.

handleError

errorHandler() suivant sera partagé par de nombreuses méthodes *HeroService* afin qu'il soit généralisé pour répondre à leurs différents besoins.

Au lieu de gérer l'erreur directement, elle renvoie une fonction de gestionnaire d'erreurs à *catchError* qu'elle a configurée avec le nom de l'opération qui a échoué et une valeur de retour sûre.

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

Après avoir signalé l'erreur à la console, le gestionnaire construit un message convivial et renvoie une valeur sûre à l'application afin qu'elle puisse continuer à fonctionner.

Parce que chaque méthode de service renvoie un type de résultat *Observable* différent, *errorHandler()* prend un paramètre de type afin qu'il puisse renvoyer la valeur sûre comme le type attendu par l'application.

Puiser Dans l'*Observable*

Les méthodes *HeroService* vont puiser dans le flux de valeurs observables et envoyer un message (*via log()*) à la zone de message en bas de la page.

Ils vont le faire avec l'opérateur *tap* RxJS, qui regarde les valeurs observables, fait quelque chose avec ces valeurs, et les transmet. Le rappel de *tap* ne touche pas les valeurs elles-mêmes.

Voici la version finale de *getHeroes* avec le *tap* qui enregistre l'opération.

```
/** 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', []))
    );
}
```

Obtenir un héros par identifiant (id)

La plupart des API Web supportent une requête get by id au format *api/hero/:id* (comme *api/hero/11*). Ajoutez une méthode *HeroService.getHero()* pour effectuer cette requête:

```
src/app/hero.service.ts

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

Il y a trois différences significatives par rapport à *getHeroes()*.

Il construit une URL de requête avec l'identifiant du héros désiré.
le serveur devrait répondre avec un seul héros plutôt qu'un ensemble de héros.
par conséquent, getHero renvoie un *Observable<Hero>* (« un observable des objets Hero ») plutôt qu'un observable des tableaux de héros.

Mettre à jour les héros

Modification du nom d'un héros dans la vue détaillée du héros. Au fur et à mesure que vous tapez, le nom du héros met à jour l'en-tête en haut de la page. Mais lorsque vous cliquez sur le bouton « revenir en arrière », les modifications sont perdues.

Si vous souhaitez que les modifications persistent, vous devez les réécrire sur le serveur.

À la fin du modèle de détail de héros, ajoutez un bouton de sauvegarde avec une liaison d'événement de clic qui appelle une nouvelle méthode de composant nommée *save()*.

```
src /app/hero-detail/hero-detail.component.html (enregistrer)


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

Ajoutez la méthode *save()* suivante, qui conserve les modifications de nom de héros en utilisant la méthode *updateHero()* du service *hero*, puis revient à la vue précédente.

```
src/app/hero-detail/hero-detail.component.ts (enregistrer)


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

Ajouter *HeroService.updateHero()*

La structure globale de la méthode *updateHero()* est similaire à celle de *getHeroes()*, mais elle utilise *http.put()* pour conserver le héros modifié sur le serveur.

src/app/hero.service.ts (mise à jour)

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

La méthode *HttpClient.put()* prend trois paramètres
l'URL
les données à mettre à jour (le héros modifié dans ce cas)
options
L'URL est inchangée. L'API web des héros sait quel héros mettre à jour en regardant l'identifiant du héros.
L'API web des héros attend un en-tête spécial dans les requêtes d'enregistrement HTTP. Cet en-tête est dans la constante *httpOptions* définie dans *HeroService*.

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

Actualisez le navigateur, modifiez le nom d'un héros, enregistrez votre modification et cliquez sur le bouton « go back » (revenir en arrière). Le héros apparaît maintenant dans la liste avec le nom changé.

Ajouter un nouveau héros

Pour ajouter un héros, cette application n'a besoin que du nom du héros. Vous pouvez utiliser un élément d'entrée associé à un bouton d'ajout.

Insérez le texte suivant dans le modèle *HeroesComponent*, juste après l'en-tête:

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

```
<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>
```

En réponse à un événement click, appelez le gestionnaire de clic du composant, puis effacez le champ de saisie afin qu'il soit prêt pour un autre nom.

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);
    });
}
```

Lorsque le nom donné n'est pas vide, le gestionnaire crée un objet de type *Hero* à partir du nom (il ne manque que l'identifiant *id*) et le passe à la méthode services *addHero()*.

Lorsque *addHero* enregistre avec succès, le callback *subscribe* reçoit le nouveau héros et le pousse dans la liste des *heroes* pour l'affichage.

Vous allez écrire *HeroService.addHero* dans la section suivante.

Ajouter *HeroService.addHero()*

Ajoutez la méthode *addHero()* suivante à la classe *HeroService*.

```
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'))
  );
}
```

HeroService.addHero() diffère de *updateHero* de deux façons.

il appelle *HttpClient.post()* au lieu de *put()*.
il s'attend à ce que le serveur génère un identifiant pour le nouveau héros, qu'il renvoie dans l' *Observable<Héros>* à l'appelant.
Rafraîchissez le navigateur et ajoutez des héros.

Supprimer Un Héros

Chaque héros de la liste des héros devrait avoir un bouton de suppression.
Ajoutez l'élément de bouton suivant au modèle du composant Héros *HeroesComponent*, après le nom du héros dans l'élément *<li>* répété.

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

Le code HTML de la liste des héros devrait ressembler à ceci:

```
src/app/heroes/heroes.component.html (liste des héros)


<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>
```

Pour positionner le bouton de suppression à l'extrême droite de l'entrée du héros, ajoutez du CSS à *heroes.component.css*. Vous trouverez ce CSS dans le code de révision finale ci-dessous.

Ajoutez le gestionnaire *delete()* au composant.

```
src/app/heroes/heroes.component.ts (supprimer)


delete(hero: Hero): void {
```

```
  this.heroes = this.heroes.filter(h => h !== hero);
  this.heroService.deleteHero(hero).subscribe();
}
```

Bien que le composant délègue la suppression du héros au *HeroService*, il reste responsable de la mise à jour de sa propre liste de héros. La méthode *delete()* du composant supprime immédiatement le héro-à-suprimer de cette liste, en anticipant que *HeroService* réussira sur le serveur.

Il n'y a vraiment rien pour le composant à faire avec *Observable* retourné par *heroService.delete()*. Il doit s'abonner de toute façon.

Si vous négligez de ***subscribe()***, le service n'enverra pas la demande de suppression au serveur! En règle générale, un ***observable*** ne fait rien jusqu'à ce que quelque chose souscrit!

Confirmez-le par vous-même en supprimant temporairement le ***subscribe()***, en cliquant sur « Dashboard », puis en cliquant sur « Heroes ». Vous verrez à nouveau la liste complète des héros.

Ajouter *HeroService.deleteHero()*

Ajoutez une méthode*deleteHero()*à*HeroService*comme ceci.

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

Notez que

- ça appelle *HttpClient.delete*.
- l'URL est l'URL de la ressource *heroes* plus l'identifiant *id* du héros à supprimer
- vous n'envoyez pas de données comme vous l'avez fait avec *put* et *post*.
- vous envoyez toujours les *httpOptions*.

Actualisez le navigateur et essayez la nouvelle fonctionnalité de suppression.

Recherche Par Nom

Dans ce dernier exercice, vous apprendrez à enchaîner les opérateurs *Observable* afin de minimiser le nombre de requêtes HTTP similaires et de consommer économiquement la bande passante du réseau.

Vous allez ajouter une fonctionnalité de recherche de héros au tableau de bord. Lorsque l'utilisateur tape un nom dans une zone de recherche, vous effectuez des requêtes HTTP répétées pour les héros filtrés par ce nom. Votre objectif est d'émettre uniquement autant de demandes que nécessaire.

HeroService.searchHeroes

Commencez par ajouter une méthode *searchHeroes* au *HeroService*.

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', []))
  );
}
```

La méthode retourne immédiatement avec un tableau vide s'il n'y a pas de terme de recherche. Le reste ressemble beaucoup à *getHeroes()*. La seule différence significative est l'URL, qui inclut une chaîne de requête avec le terme de recherche.

Ajouter une recherche au tableau de bord

Ouvrez le modèle *DashboardComponent* et ajoutez l'élément de recherche hero, <*app-hero-search*>, en bas du modèle *DashboardComponent*.

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>
```

Ce modèle ressemble beaucoup au répéteur ***ngFo**r du modèle *HeroesComponent*.

Malheureusement, l'ajout de cet élément casse l'application. Angular ne peut pas trouver un composant avec un sélecteur correspondant à <*app-hero-search*>.

Le *HeroSearchComponent* n'existe pas encore. Réparons ça.

Créer HeroSearchComponent

Créez un HeroSearchComponent avec l'interface de ligne de commande CLI.

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

L'interface CLI génère les trois *HeroSearchComponent* et ajoute le composant aux déclarations `*AppModule* '

Remplacez le modèle *HeroSearchComponent* généré par une zone de texte et une liste de résultats de recherche correspondants comme celui-ci.

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>
```

Ajoutez des styles CSS privés à *hero-search.component.css* comme indiqué dans la révision de code finale ci-dessous.
Lorsque l'utilisateur tape dans la zone de recherche, une liaison d'événement de touche appelle la méthode *search()* du composant avec la nouvelle valeur de la zone de recherche.

AsyncPipe

Comme prévu, le **ngFor* répète les objets héros.

Regardez attentivement et vous verrez que le **ngFor* itére sur une liste appelée *heroes$*, et non *heroes*.

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

Le *$* est une convention qui indique que les *heroes$* est un *observable*, pas un tableau.

Le **ngFor* ne peut rien faire avec un *Observable*. Mais il y a aussi un caractère *pipe(|)* suivi de *async*, qui identifie *AsyncPipe* d'Angular.

AsyncPipe s'abonne automatiquement à un *Observable*, donc vous n'aurez pas à le faire dans la classe du composant.

Corrigez la classe HeroSearchComponent

Remplacez la classe *HeroSearchComponent* générée et les métadonnées comme suit.

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[]>;
  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)),
    );
  }
```

```
}
```

Notez la déclaration des *heroes$* comme un *Observable.*

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

Vous le définissez dans *ngOnInit()*. Avant cela, concentrez-vous sur la définition de *searchTerms*.

Le sujet de recherche RxJS

La propriété *searchTerms* est déclarée comme un RxJS *Subject*.

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

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

Un sujet *Subject* est à la fois une source de valeurs *Observables* et un *Observable* lui-même. Vous pouvez vous abonner à un sujet *Subject* comme vous le feriez pour tout *Observable*.

Vous pouvez également insérer des valeurs dans *Observable* en appelant sa méthode *next(value)* comme le fait la méthode *search()*.

La méthode *search()* est appelée via un événement lié à l'événement de frappe *keystroke* de la zone de texte.

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

Chaque fois que l'utilisateur tape dans la zone de texte, la liaison appelle *search()* avec la valeur *textbox*, un « terme de recherche ». *SearchTerms* devient un *Observable* émettant un flux constant de termes de recherche.

Enchaînement des opérateurs RxJS

Passer un nouveau terme de recherche directement à *searchHeroes()* après chaque frappe de l'utilisateur créerait une quantité excessive de requêtes HTTP, taxant les ressources du serveur et brûlant à travers le plan de données du réseau cellulaire.

Au lieu de cela, la méthode *ngOnInit()* redirige les *searchTerms Observables* à travers une séquence d'opérateurs RxJS qui réduit le nombre d'appels à *searchHeroes()*, renvoyant finalement un résultat de recherche de héros en temps opportun (chacun un *Hero[]*).

Voici le code

```
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) attend que le flux de nouveaux événements de chaîne marque une pause de 300 millisecondes avant de transmettre la dernière chaîne. Vous ne ferez jamais de demandes plus fréquemment que 300ms.
Le paramètre *distinctUntilChanged()* garantit qu'une requête est envoyée uniquement si le texte du filtre a changé.
switchMap() appelle le service de recherche pour chaque terme de recherche qui le fait passer par *debounce* et *distinctUntilChanged*. Il annule et se jètte les Observables de recherche précédents, renvoyant uniquement le dernier service de recherche observable.

Avec l'opérateur ***switchMap***, chaque événement clé qualifié peut déclencher un appel de méthode ***HttpClient.get()***. Même avec une pause de 300 ms entre les requêtes, vous pouvez avoir plusieurs requêtes HTTP en vol et elles peuvent ne pas revenir dans l'ordre envoyé.

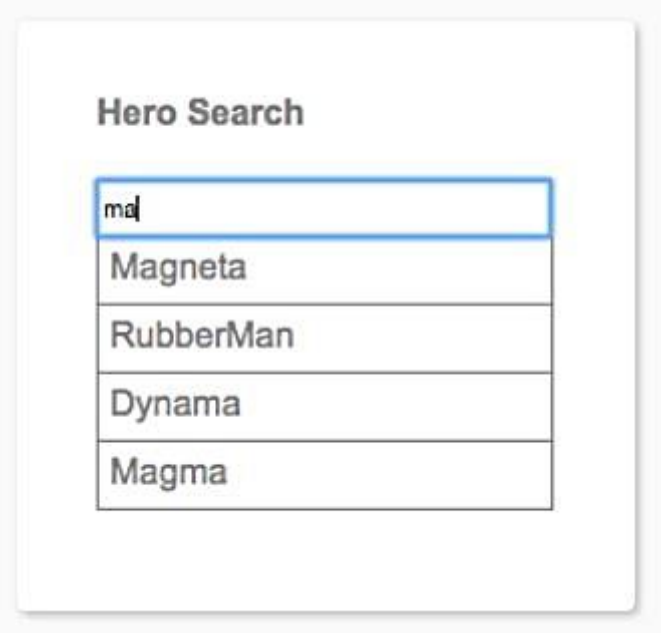
switchMap() préserve l'ordre de requête d'origine tout en renvoyant uniquement l'observable de l'appel de méthode HTTP le plus récent. Les résultats des appels précédents sont annulés et rejetés.

Notez que l'annulation d'un précédent ***searchHeroes()*** Observable n'abandonne pas réellement une requête HTTP en attente. Les résultats indésirables sont simplement supprimés avant qu'ils n'atteignent le code de votre application.

Rappelez-vous que la classe de composant ne s'abonne pas aux observables des *heroes$*. C'est le travail de l'*AsyncPipe* dans le modèle.

Essayons cela

Exécutez l'application à nouveau. Dans le tableau de bord, entrez du texte dans la zone de recherche. Si vous entrez des caractères qui correspondent à des noms de héros existants, vous verrez quelque chose comme ça.



Code Final: Revue

Votre application devrait ressembler à cet ***exemple en direct***/***télécgargement de l'exemple*** pour ceux et celles qui le veulent.

Voici les fichiers de code discutés sur cette page (ils sont tous dans le dossier *src/app/*)

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', []))
      );
  }

  /** GET hero by id. Return `undefined` when id not found */
  getHeroNo404<Data>(id: number): Observable<Hero> {
    const url = `${this.heroesUrl}/?id=${id}`;
    return this.http.get<Hero[]>(url)
      .pipe(
        map(heroes => heroes[0]), // returns a {0|1} element array
        tap(h => {
          const outcome = h ? `fetched` : `did not find`;
          this.log(`${outcome} hero id=${id}`);
        }),
        catchError(this.handleError<Hero>(`getHero id=${id}`))
      );
  }

  /** 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}`))
    );
  }

  /* 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', []))
    );
  }

  //////// Save methods //////////

  /** 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'))
    );
  }

  /** 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'))
    );
```

```
  }

  /** 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'))
    );
  }

  /**
   * 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);
  }
}
```

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};
  }
}
```

app.module.ts

```
import { NgModule }       from '@angular/core';
import { BrowserModule }  from '@angular/platform-browser';
import { FormsModule }    from '@angular/forms';
import { HttpClientModule }    from '@angular/common/http';

import { HttpClientInMemoryWebApiModule } from 'angular-in-memory-web-api';
import { InMemoryDataService }  from './in-memory-data.service';

import { AppRoutingModule }     from './app-routing.module';

import { AppComponent }         from './app.component';
```

```
import { DashboardComponent }   from './dashboard/dashboard.component';
import { HeroDetailComponent }  from './hero-detail/hero-detail.component';
import { HeroesComponent }      from './heroes/heroes.component';
import { HeroSearchComponent }  from './hero-search/hero-search.component';
import { MessagesComponent }    from './messages/messages.component';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    AppRoutingModule,
    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 }
    )
  ],
  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;
}

button:hover {
  background-color: #cfd8dc;
}

button.delete {
  position: relative;
  left: 194px;
  top: -32px;
  background-color: gray !important;
  color: white;
}
```

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

```
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)
      .subscribe(() => this.goBack());
  }
}
```

HeroSearchComponent

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>
```

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[]>;
  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;
  border-right: 1px solid gray;
  width:195px;
  height: 16px;
  padding: 5px;
  background-color: white;
  cursor: pointer;
  list-style-type: none;
}

.search-result li:hover {
  background-color: #607D8B;
}

.search-result li a {
  color: #888;
  display: block;
  text-decoration: none;
}

.search-result li a:hover {
  color: white;
}
.search-result li a:active {
  color: white;
}
#search-box {
  width: 200px;
  height: 20px;
}


ul.search-result {
  margin-top: 0;
  padding-left: 0;
}
```

Récapitulons!

Vous êtes à la fin de votre voyage et vous avez déjà accompli beaucoup de choses.

Dans cette section nous avons pu:

- ajouter les dépendances nécessaires pour utiliser HTTP dans l'application.
- refactoriser *HeroService* pour charger des héros à partir d'une API web.
- étendre *HeroService* pour prendre en charge les méthodes *post()*, *put()* et *delete()*.
- metre à jour les composants pour autoriser l'ajout, la modification et la suppression des héros.

- configurer une API Web en mémoire.
- apprendre à utiliser des observables