HTTP CLIENT

Talk to a remote server with the Angular Http Client.

HTTP is the primary protocol for browser/server communication.

> The `WebSocket` protocol is another important communication technology; we won't cover it in this chapter.

Modern browsers support two HTTP-based APIs: XMLHttpRequest (XHR) and JSONP. A few browsers also support Fetch.

The Angular HTTP client library simplifies application programming of the **XHR** and **JSONP** APIs as we'll learn in this chapter covering:

Http client sample overview

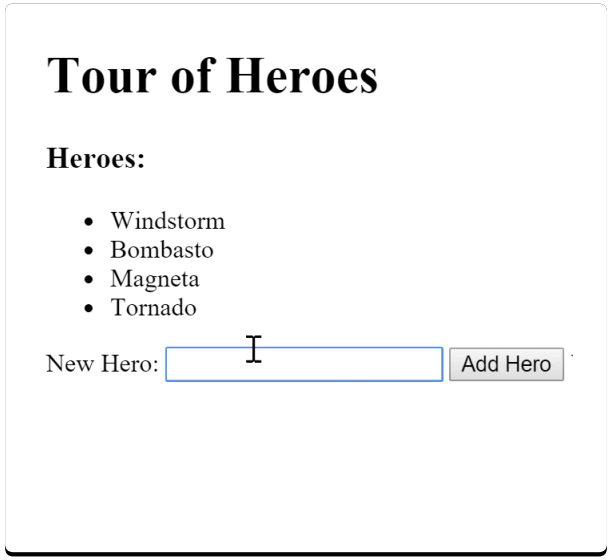
We illustrate these topics with code that you can run live in a browser.

The *Http* Client Demo

We use the Angular `Http` client to communicate via `XMLHttpRequest (XHR)`.

We'll demonstrate with a mini-version of the tutorial's "Tour of Heroes" (ToH) application. This version gets some heroes from the server, displays them in a list, lets us add new heroes, and save them to the server.

It works like this.



It's implemented with two components — a parent `TohComponent` shell and the `HeroListComponent` child. We've seen these kinds of component in many other documentation samples. Let's see how they change to support communication with a server.

> We're overdoing the "separation of concerns" by creating two components for a tiny demo. We're making a point about application structure that is easier to justify when the app grows.

Here is the `TohComponent` shell:

app/toh.component.ts

```
1. import {Component}          from 'angular2/core';
2. import {HTTP_PROVIDERS}     from 'angular2/http';
3.
```

```
import {Hero}              from './hero';
import {HeroListComponent} from './hero-list.component';
import {HeroService}       from './hero.service';

@Component({
  selector: 'my-toh',
  template: `
  <h1>Tour of Heroes</h1>
  <hero-list></hero-list>
  `,
  directives:[HeroListComponent],
  providers: [
    HTTP_PROVIDERS,
    HeroService,
  ]
})
export class TohComponent { }
```

As usual, we import the symbols we need. The newcomer is `HTTP_PROVIDERS`, an array of service providers from the Angular HTTP library. We'll be using that library to access the server. We also import a `HeroService` that we'll look at shortly.

The component specifies both the `` `HTTP_PROVIDERS `` and the `HeroService` in the metadata `providers` array, making them available to the child components of this "Tour of Heroes" application.

> Learn about providers in the Dependency Injection chapter.

This sample only has one child, the `HeroListComponent` shown here in full:

app/toh/hero-list.component.ts

```
import {Component, OnInit} from 'angular2/core';
import {Hero}              from './hero';
import {HeroService}       from './hero.service';

@Component({
  selector: 'hero-list',
  template: `
  <h3>Heroes:</h3>
  <ul>
    <li *ngFor="#hero of heroes">
      {{ hero.name }}
    </li>
  </ul>
  New Hero:
  <input #newHero />
  <button (click)="addHero(newHero.value); newHero.value=''">
    Add Hero
  </button>
  <div class="error" *ngIf="errorMessage">{{errorMessage}}</div>
  `,
  styles: ['.error {color:red;}']
})
export class HeroListComponent implements OnInit {

  constructor (private _heroService: HeroService) {}

```

```
27.   errorMessage: string;
28.   heroes:Hero[];
29.
30.   ngOnInit() { this.getHeroes(); }
31.
32.   getHeroes() {
33.     this._heroService.getHeroes()
34.                      .subscribe(
35.                        heroes => this.heroes = heroes,
36.                        error =>  this.errorMessage = <any>error);
37.   }
38.
39.   addHero (name: string) {
40.     if (!name) {return;}
41.     this._heroService.addHero(name)
42.                      .subscribe(
43.                        hero  => this.heroes.push(hero),
44.                        error =>  this.errorMessage = <any>error);
45.   }
46. }
```

The component template displays a list of heroes with the `NgFor` repeater directive.

Beneath the heroes is an input box and an *Add Hero* button where we can enter the names of new heroes and add them to the database. We use a local template variable, `newHero`, to access the value of the input box in the `(click)` event binding. When the user clicks the button, we pass that value to the component's `addHero` method and then clear it to make ready for a new hero name.

Below the button is an optional error message.

The *HeroListComponent* class

We inject the `HeroService` into the constructor. That's the instance of the `HeroService` that we provided in the parent shell `TohComponent`.

Notice that the component **does not talk to the server directly!** The component doesn't know or care how we get the data. Those details it delegates to the `heroService` class (which we'll get to in a moment). This is a golden rule: *always delegate data access to a supporting service class*.

Although the component should request heroes immediately, we do **not** call the service `get` method in the component's constructor. We call it inside the `ngOnInit` lifecycle hook instead and count on Angular to call `ngOnInit` when it instantiates this component.

> This is a "best practice". Components are easier to test and debug when their constructors are simple and all real work (especially calling a remote server) is handled in a separate method.

The service `get` and `addHero` methods return an `Observable` of HTTP Responses to which we `subscribe`, specifying the actions to take if a method succeeds or fails. We'll get to observables and subscription shortly.

With our basic intuitions about the component squared away, we can turn to development of the backend data source and the client-side `HeroService` that talks to it.

Fetch data

In many of our previous samples we faked the interaction with the server by returning mock heroes in a service like this one:

```
import {Hero} from './hero';
import {HEROES} from './mock-heroes';
import {Injectable} from 'angular2/core';

@Injectable()
export class HeroService {
  getHeroes() {
    return Promise.resolve(HEROES);
  }
}
```

In this chapter, we get the heroes from the server using Angular's own HTTP Client service. Here's the new `HeroService`:

app/toh/hero.service.ts

```
import {Injectable}     from 'angular2/core';
import {Http, Response} from 'angular2/http';
import {Hero}           from './hero';
import {Observable}     from 'rxjs/Observable';

@Injectable()
export class HeroService {
  constructor (private http: Http) {}

  private _heroesUrl = 'app/heroes';  // URL to web api
```

```
  getHeroes () {
    return this.http.get(this._heroesUrl)
                    .map(res => <Hero[]> res.json().data)
                    .catch(this.handleError);
  }
  private handleError (error: Response) {
    // in a real world app, we may send the error to some remote logging infrastructure
    // instead of just logging it to the console
    console.error(error);
    return Observable.throw(error.json().error || 'Server error');
  }
}
```

We begin by importing Angular's `Http` client service and inject it into the `HeroService` constructor.

`Http` is not part of the Angular core. It's an optional service in its own `angular2/http` library. Moreover, this library isn't even part of the main Angular script file. It's in its own script file (included in the Angular npm bundle) which we must load in `index.html`.

index.html

```
<script src="node_modules/angular2/bundles/http.dev.js"></script>
```

Look closely at how we call `http.get`

app/toh/hero.service.ts (http.get)

```
return this.http.get(this._heroesUrl)
                .map(res => <Hero[]> res.json().data)
                .catch(this.handleError);
```

We pass the resource URL to `get` and it calls the server which should return heroes.

> It *will* return heroes once we've set up the in-memory web api described in the appendix below.
>
> Alternatively, we can (temporarily) target a JSON file by changing the endpoint URL:
>
> ```
> private _heroesUrl = 'app/heroes.json'; // URL to JSON file
> ```

The return value may surprise us. Many of us would expect a promise. We'd expect to chain a call to `then()` and extract the heroes. Instead we're calling a `map()` method. Clearly this is not a promise.

In fact, the `http.get` method returns an **Observable** of HTTP Responses (`Observable<Response>`) from the RxJS library and `map` is one of the RxJS *operators*.

HTTP GET DELAYED

The `http.get` does **not send the request just yet!** This observable is *cold* which means the request won't go out until something *subscribes* to the observable. That *something* is the HeroListComponent.

RxJS Library

RxJS ("Reactive Extensions") is a 3rd party library, endorsed by Angular, that implements the *asynchronous observable* pattern.

All of our Developer Guide samples have installed the RxJS npm package and loaded the RxJS script in `index.html` because observables are used widely in Angular applications.

index.html

```
<script src="node_modules/rxjs/bundles/Rx.js"></script>
```

We certainly need it now when working with the HTTP client. And we must take a critical extra step to make RxJS observables usable.

Enable RxJS Operators

The RxJS library is quite large. Size matters when we build a production application and deploy it to mobile devices. We should include only those features that we actually need.

Accordingly, Angular exposes a stripped down version of `Observable` in the `rxjs/Observable` module, a version that lacks almost all operators including the ones we'd like to use here such as the `map` method we called above in `getHeroes`.

It's up to us to add the operators we need. We could add each operator, one-by-one, until we had a custom *Observable* implementation tuned precisely to our requirements.

That would be a distraction today. We're learning HTTP, not counting bytes. So we'll make it easy on ourselves and enrich

Observable with the full set of operators. It only takes one `import` statement. It's best to add that statement early when we're bootstrapping the application. :

app/main.ts (import rxjs)

```
// Add all operators to Observable
import 'rxjs/Rx';
```

Map the response object

Let's come back to the `HeroService` and look at the `http.get` call again to see why we needed `map()`

app/toh/hero.service.ts (http.get)

```
return this.http.get(this._heroesUrl)
                .map(res => <Hero[]> res.json().data)
                .catch(this.handleError);
```

The `response` object does not hold our data in a form we can use directly. It takes an additional step — calling `response.json()` — to transform the bytes from the server into a JSON object.

> This is not Angular's own design. The Angular HTTP client follows the ES2015 specification for the response object returned by the `Fetch` function. That spec defines a `json()` method that parses the response body into a JavaScript object.

> We shouldn't expect `json()` to return the heroes array directly. The server we're calling always wraps JSON results in an object with a `data` property. We have to unwrap it to get the heroes. This is conventional web api behavior, driven by security concerns.

Make no assumptions about the server API. Not all servers return an object with a `data` property.

Do not return the response object

Our `getHeroes()` could have returned the `Observable<Response>`.

Bad idea! The point of a data service is to hide the server interaction details from consumers. The component that calls the `HeroService` wants heroes. It has no interest in what we do to get them. It doesn't care where they come from. And it certainly doesn't want to deal with a response object.

Always handle errors

The eagle-eyed reader may have spotted our use of the `catch` operator in conjunction with a `handleError` method. We haven't discussed so far how that actually works. Whenever we deal with I/O we must be prepared for something to go wrong as it surely will.

We should catch errors in the `HeroService` and do something with them. We may also pass an error message back to the component for presentation to the user but only if we can say something the user can understand and act upon.

In this simple app we provide rudimentary error handling in both the service and the component.

We use the Observable `catch` operator on the service level. It takes an error handling function with the failed `Response` object as the argument. Our service handler, `errorHandler`, logs the response to the console, transforms the error into a user-friendly message, and returns the message in a new, failed observable via `Observable.throw`.

app/toh/hero.service.ts

```
getHeroes () {
  return this.http.get(this._heroesUrl)
                  .map(res => <Hero[]> res.json().data)
                  .catch(this.handleError);
}
private handleError (error: Response) {
  // in a real world app, we may send the error to some remote logging infrastructure
  // instead of just logging it to the console
  console.error(error);
  return Observable.throw(error.json().error || 'Server error');
}
```

Subscribe in the *HeroListComponent*

Back in the `HeroListComponent`, where we called `heroService.get`, we supply the `subscribe` function with a second function to handle the error message. It sets an `errorMessage` variable which we've bound conditionally in the template.

app/toh/hero-list.component.ts (getHeroes)

```
getHeroes() {
  this._heroService.getHeroes()
                   .subscribe(
                     heroes => this.heroes = heroes,
                     error =>  this.errorMessage = <any>error);
}
```

> Want to see it fail? Reset the api endpoint in the `HeroService` to a bad value. Remember to restore it!

Peek at results in the console

During development we're often curious about the data returned by the server. Logging to console without disrupting the flow would be nice.

The Observable `do` operator is perfect for the job. It passes the input through to the output while we do something with a useful side-effect such as writing to console. Slip it into the pipeline between `map` and `catch` like this.

app/toh/hero.service.ts

```
    return this.http.get(this._heroesUrl)
                    .map(res => <Hero[]> res.json().data)
                    .do(data => console.log(data)) // eyeball results in the console
                    .catch(this.handleError);
```

Remember to comment it out before going to production!

Send data to the server

So far we've seen how to retrieve data from a remote location using Angular's built-in `Http` service. Let's add the ability to create new heroes and save them in the backend.

We'll create an easy method for the `HeroListComponent` to call, an `addHero` method that takes just the name of a new hero and returns an observable holding the newly-saved hero:

```
addHero (name: string) : Observable<Hero>
```

To implement it, we need to know some details about the server's api for creating heroes.

Our data server follows typical REST guidelines. It expects a `POST` request at the same endpoint where we `GET` heroes. It expects the new hero data to arrive in the body of the request, structured like a `Hero` entity but without the `id` property. The body of the request should look like this:

```
{ "name": "Windstorm" }
```

The server will generate the `id` and return the entire `JSON` representation of the new hero including its generated id. The hero arrives tucked inside a response object with its own `data` property.

Now that we know how the API works, we implement `addHero` like this:

app/toh/hero.service.ts (additional imports)

```
import {Headers, RequestOptions} from 'angular2/http';
```

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

```
addHero (name: string) : Observable<Hero>  {

  let body = JSON.stringify({ name });
  let headers = new Headers({ 'Content-Type': 'application/json' });
  let options = new RequestOptions({ headers: headers });

  return this.http.post(this._heroesUrl, body, options)
                  .map(res =>  <Hero> res.json().data)
                  .catch(this.handleError)
}
```

The second *body* parameter of the `post` method requires a JSON ***string*** so we have to `JSON.stringify` the hero content before sending.

> We may be able to skip the `stringify` step in the near future.

Headers

The server requires a `Content-Type` header for the body of the POST. Headers are one of the RequestOptions. Compose the options object and pass it in as the *third* parameter of the `post` method.

app/toh/hero.service.ts (headers)

```
let headers = new Headers({ 'Content-Type': 'application/json' });
let options = new RequestOptions({ headers: headers });

return this.http.post(this._heroesUrl, body, options)
```

JSON results

As with `get`, we extract the data from the response with `json()` and unwrap the hero via the `data` property.

> Know the shape of the data returned by the server. *This* web api returns the new hero wrapped in an object with a `data` property. A different api might just return the hero in which case we'd omit the `data` de-reference.

Back in the `HeroListComponent`, we see that *its* `addHero` method subscribes to the observable returned by the *service's* `addHero` method. When the data arrive it pushes the new hero object into its `heroes` array for presentation to the user.

app/toh/hero-list.component.ts (addHero)

```
addHero (name: string) {
```

```
  if (!name) {return;}
  this._heroService.addHero(name)
                   .subscribe(
                     hero  => this.heroes.push(hero),
                     error =>  this.errorMessage = <any>error);
}
```

Fall back to Promises

Although the Angular `http` client API returns an `Observable<Response>` we can turn it into a Promise if we prefer. It's easy to do and a promise-based version looks much like the observable-based version in simple cases.

> While promises may be more familiar, observables have many advantages. Don't rush to promises until you give observables a chance.

Let's rewrite the `HeroService` using promises , highlighting just the parts that are different.

```
1. getHeroes () {
2.   return this.http.get(this._heroesUrl)
3.                   .toPromise()
4.                   .then(res => <Hero[]> res.json().data, this.handleError)
5.                   .then(data => { console.log(data); return data; }); // eyeball results in the
   console
6. }
```

```
 7.
 8. addHero (name: string) : Promise<Hero> {
 9.   let body = JSON.stringify({ name });
10.   let headers = new Headers({ 'Content-Type': 'application/json' });
11.   let options = new RequestOptions({ headers: headers });
12.
13.   return this.http.post(this._heroesUrl, body, options)
14.               .toPromise()
15.               .then(res => <Hero> res.json().data)
16.               .catch(this.handleError);
17. }
18.
19. private handleError (error: any) {
20.   // in a real world app, we may send the error to some remote logging infrastructure
21.   // instead of just logging it to the console
22.   console.error(error);
23.   return Promise.reject(error.message || error.json().error || 'Server error');
24. }
```

Converting from an observable to a promise is as simple as calling `toPromise(success, fail)`.

We move the observable's `map` callback to the first *success* parameter and its `catch` callback to the second *fail* parameter and we're done! Or we can follow the promise `then.catch` pattern as we do in the second `addHero` example.

Our `errorHandler` forwards an error message as a failed promise instead of a failed Observable.

The diagnostic *log to console* is just one more `then` in the promise chain.

We have to adjust the calling component to expect a `Promise` instead of an `Observable`.

```
getHeroes() {
  this._heroService.getHeroes()
                   .then(
                     heroes => this.heroes = heroes,
                     error =>  this.errorMessage = <any>error);
}

addHero (name: string) {
  if (!name) {return;}
  this._heroService.addHero(name)
                   .then(
                     hero  => this.heroes.push(hero),
                     error =>  this.errorMessage = <any>error);
}
```

The only obvious difference is that we call `then` on the returned promise instead of `subscribe`. We give both methods the same functional arguments.

> The less obvious but critical difference is that these two methods return very different results!
>
> The promise-based `then` returns another promise. We can keep chaining more `then` and `catch` calls, getting a new promise each time.
>
> The `subscribe` method returns a `Subscription`. A `Subscription` is not another `Observable`. It's the end of the line for observables. We can't call `map` on it or call `subscribe` again. The `Subscription` object has a different purpose, signified by its primary method, `unsubscribe`.
>
> Learn more about observables to understand the implications and consequences of subscriptions.

Get data with `JSONP`

We just learned how to make `XMLHttpRequests` using Angulars built-in `Http` service. This is the most common approach for server communication. It doesn't work in all scenarios.

For security reasons, web browsers block `XHR` calls to a remote server whose origin is different from the origin of the web page. The *origin* is the combination of URI scheme, hostname and port number. This is called the Same-origin Policy.

> Modern browsers do allow `XHR` requests to servers from a different origin if the server supports the CORS protocol. If the server requires user credentials, we'll enable them in the request headers.

Some servers do not support CORS but do support an older, read-only alternative called JSONP. Wikipedia is one such server.

> This StackOverflow answer covers many details of JSONP.

Search wikipedia

Wikipedia offers a `JSONP` search api. Let's build a simple search that shows suggestions from wikipedia as we type in a text box.

The Angular `Jsonp` service both extends the `Http` service for JSONP and restricts us to `GET` requests. All other HTTP methods throw an error because JSONP is a read-only facility.

As always, we wrap our interaction with an Angular data access client service inside a dedicated service, here called `WikipediaService`.

app/wiki/wikipedia.service.ts

```
1. import {Injectable} from 'angular2/core';
2. import {Jsonp, URLSearchParams} from 'angular2/http';
3.
4. @Injectable()
5. export class WikipediaService {
6.   constructor(private jsonp: Jsonp) {}
7.
```

```
  search (term: string) {

    let wikiUrl = 'http://en.wikipedia.org/w/api.php';

    var params = new URLSearchParams();
    params.set('search', term); // the user's search value
    params.set('action', 'opensearch');
    params.set('format', 'json');
    params.set('callback', 'JSONP_CALLBACK');

    // TODO: Add error handling
    return this.jsonp
               .get(wikiUrl, { search: params })
               .map(request => <string[]> request.json()[1]);
  }
}
```

The constructor expects Angular to inject its `jsonp` service. We register that service with `JSONP_PROVIDERS` in the component below that calls our `WikipediaService`.

Search parameters

The Wikipedia 'opensearch' API expects four parameters (key/value pairs) to arrive in the request URL's query string. The keys are `search`, `action`, `format`, and `callback`. The value of the `search` key is the user-supplied search term to find in Wikipedia. The other three are the fixed values "opensearch", "json", and "JSONP_CALLBACK" respectively.

> The `JSONP` technique requires that we pass a callback function name to the server in the query string: `callback=JSONP_CALLBACK`. The server uses that name to build a JavaScript wrapper function in its response which Angular ultimately calls to extract the data. All

of this happens under the hood.

If we're looking for articles with the word "Angular", we could construct the query string by hand and call `jsonp` like this:

```
let queryString =
  `?search=${term}&action=opensearch&format=json&callback=JSONP_CALLBACK`

return this.jsonp
           .get(wikiUrl + queryString)
           .map(request => <string[]> request.json()[1]);
```

In more parameterized examples we might prefer to build the query string with the Angular `URLSearchParams` helper as shown here:

app/wiki/wikipedia.service.ts (search parameters)

```
var params = new URLSearchParams();
params.set('search', term); // the user's search value
params.set('action', 'opensearch');
params.set('format', 'json');
params.set('callback', 'JSONP_CALLBACK');
```

This time we call `jsonp` with *two* arguments: the `wikiUrl` and an options object whose `search` property is the `params` object.

app/wiki/wikipedia.service.ts (call jsonp)

```
// TODO: Add error handling
return this.jsonp
           .get(wikiUrl, { search: params })
           .map(request => <string[]> request.json()[1]);
```

`Jsonp` flattens the `params` object into the same query string we saw earlier before putting the request on the wire.

The WikiComponent

Now that we have a service that can query the Wikipedia API, we turn to the component that takes user input and displays search results.

app/wiki/wiki.component.ts

```
1. import {Component}        from 'angular2/core';
2. import {JSONP_PROVIDERS}  from 'angular2/http';
3. import {Observable}       from 'rxjs/Observable';
4.
5. import {WikipediaService} from './wikipedia.service';
6.
7. @Component({
8.   selector: 'my-wiki',
9.   template: `
10.    <h1>Wikipedia Demo</h1>
11.    <p><i>Fetches after each keystroke</i></p>
12.
```

```
13.     <input #term (keyup)="search(term.value)"/>
14.
15.     <ul>
16.       <li *ngFor="#item of items | async">{{item}}</li>
17.     </ul>
18.   `,
19.   providers:[JSONP_PROVIDERS, WikipediaService]
20. })
21. export class WikiComponent {
22.
23.   constructor (private _wikipediaService: WikipediaService) {}
24.
25.   items: Observable<string[]>;
26.
27.   search (term: string) {
28.     this.items = this._wikipediaService.search(term);
29.   }
30. }
```

The `providers` array in the component metadata specifies the Angular `JSONP_PROVIDERS` collection that supports the `Jsonp` service. We register that collection at the component level to make `Jsonp` injectable in the `WikipediaService`.

The component presents an `<input>` element *search box* to gather search terms from the user. and calls a `search(term)` method after each `keyup` event.

The `search(term)` method delegates to our `WikipediaService` which returns an observable array of string results (`Observable<string[]>`). Instead of subscribing to the observable inside the component as we did in the `HeroListComponent`, we forward the observable result to the template (via `items`) where the async pipe in the `ngFor`

handles the subscription.

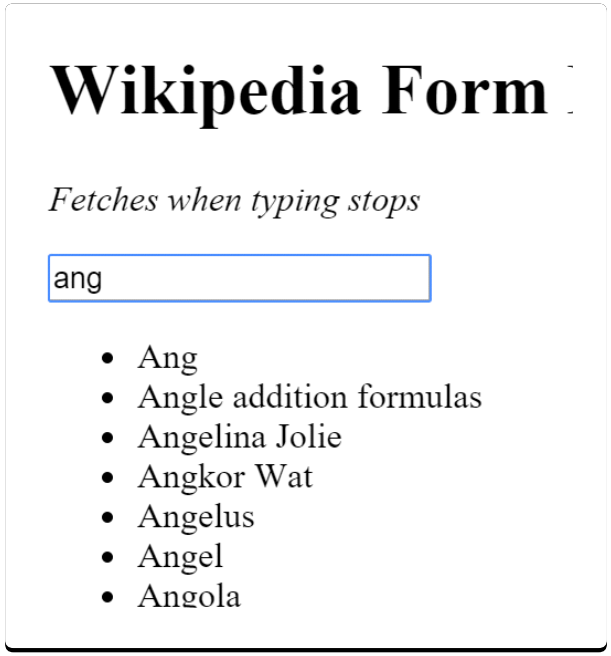
> We often use the async pipe in read-only components where the component has no need to interact with the data. We couldn't use the pipe in the `HeroListComponent` because the "add hero" feature pushes newly created heroes into the list.

Our wasteful app

Our wikipedia search makes too many calls to the server. It is inefficient and potentially expensive on mobile devices with limited data plans.

1. Wait for the user to stop typing

At the moment we call the server after every key stroke. The app should only make requests when the user *stops typing* . Here's how it *should* work — and *will* work — when we're done refactoring:



2. Search when the search term changes

Suppose the user enters the word *angular* in the search box and pauses for a while. The application issues a search request for *Angular*.

Then the user backspaces over the last three letters, *lar*, and immediately re-types *lar* before pausing once more. The search term is still "angular". The app shouldn't make another request.

3. Cope with out-of-order responses

The user enters *angular*, pauses, clears the search box, and enters *http*. The application issues two search requests, one for *angular* and one for *http*.

Which response will arrive first? We can't be sure. A load balancer could dispatch the requests to two different servers with

different response times. The results from the first *angular* request might arrive after the later *http* results. The user will be confused if we display the *angular* results to the *http* query.

When there are multiple requests in-flight, the app should present the responses in the original request order. That won't happen if *angular* results arrive last.

More fun with Observables

We can address these problems and improve our app with the help of some nifty observable operators.

We could make our changes to the `WikipediaService`. But we sense that our concerns are driven by the user experience so we update the component class instead.

app/wiki/wiki-smart.component.ts

```
1. import {Component}        from 'angular2/core';
2. import {JSONP_PROVIDERS}  from 'angular2/http';
3. import {Observable}       from 'rxjs/Observable';
4. import {Subject}          from 'rxjs/Subject';
5.
6. import {WikipediaService} from './wikipedia.service';
7.
8. @Component({
9.   selector: 'my-wiki-smart',
10.  template: `
11.    <h1>Smarter Wikipedia Demo</h1>
12.    <p><i>Fetches when typing stops</i></p>
13.
14.    <input #term (keyup)="search(term.value)"/>
```

```
15.
16.     <ul>
17.       <li *ngFor="#item of items | async">{{item}}</li>
18.     </ul>
19.   `,
20.   providers:[JSONP_PROVIDERS, WikipediaService]
21. })
22. export class WikiSmartComponent {
23.
24.   constructor (private _wikipediaService: WikipediaService) { }
25.
26.   private _searchTermStream = new Subject<string>();
27.
28.   search(term:string) { this._searchTermStream.next(term); }
29.
30.   items:Observable<string[]> = this._searchTermStream
31.     .debounceTime(300)
32.     .distinctUntilChanged()
33.     .switchMap((term:string) => this._wikipediaService.search(term));
34. }
```

We made no changes to the template or metadata, confining them all to the component class. Let's review those changes.

Create a stream of search terms

We're binding to the search box `keyup` event and calling the component's `search` method after each keystroke.

We turn these events into an observable stream of search terms using a `Subject` which we import from the RxJS observable library:

```
import {Subject}          from 'rxjs/Subject';
```

Each search term is a string, so we create a new `Subject` of type `string` called `_searchTermStream`. After every keystroke, the `search` method adds the search box value to that stream via the subject's `next` method.

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

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

Listen for search terms

Earlier, we passed each search term directly to the service and bound the template to the service results. Now we listen to the *stream of terms*, manipulating the stream before it reaches the `WikipediaService`.

```
items:Observable<string[]> = this._searchTermStream
  .debounceTime(300)
  .distinctUntilChanged()
  .switchMap((term:string) => this._wikipediaService.search(term));
```

We wait for the user to stop typing for at least 300 milliseconds (debounce). Only changed search values make it through to the service (distinctUntilChanged).

The `WikipediaService` returns a separate observable of string arrays (`Observable<string[]>`) for each request. We

could have multiple requests *in flight*, all awaiting the server's reply, which means multiple *observables-of-strings* could arrive at any moment in any order.

The switchMap (formerly known as `flatMapLatest`) returns a new observable that combines these `WikipediaService` observables, re-arranges them in their original request order, and delivers to subscribers only the most recent search results.

The displayed list of search results stays in sync with the user's sequence of search terms.

Appendix: Tour of Heroes in-memory server

If we only cared to retrieve data, we could tell Angular to get the heroes from a `heroes.json` file like this one:

app/heroes.json

```
{
  "data": [
    { "id": "1", "name": "Windstorm" },
    { "id": "2", "name": "Bombasto" },
    { "id": "3", "name": "Magneta" },
    { "id": "4", "name": "Tornado" }
  ]
}
```

> We wrap the heroes array in an object with a `data` property for the same reason that a data server does: to mitigate the security risk

posed by top-level JSON arrays.

We'd set the endpoint to the JSON file like this:

```
private _heroesUrl = 'app/heroes.json'; // URL to JSON file
```

The *get heroes* scenario would work. But we want to *save* data too. We can't save changes to a JSON file. We need a web api server.

We didn't want the hassle of setting up and maintaining a real server for this chapter. So we turned to an *in-memory web api simulator* instead. You too can use it in your own development while waiting for a real server to arrive.

First, install it with `npm`:

```
npm install a2-in-memory-web-api --save
```

Then load the script in the `index.html` below angular:

index.html

```
<script src="node_modules/a2-in-memory-web-api/web-api.js"></script>
```

The *in-memory web api* gets its data from a class with a `createDb()` method that returns a "database" object whose keys

are collection names ("heroes") and whose values are arrays of objects in those collections.

Here's the class we created for this sample by copy-and-pasting the JSON data:

app/hero-data.ts

```
export class HeroData {
  createDb() {
    let heroes = [
      { "id": "1", "name": "Windstorm" },
      { "id": "2", "name": "Bombasto" },
      { "id": "3", "name": "Magneta" },
      { "id": "4", "name": "Tornado" }
    ];
    return {heroes};
  }
}
```

We update the `HeroService` endpoint to the location of the web api data.

```
private _heroesUrl = 'app/heroes';  // URL to web api
```

Finally, we tell Angular itself to direct its http requests to the *in-memory web api* rather than externally to a remote server.

This redirection is easy because Angular's `http` delegates the client/server communication tasks to a helper service called the `XHRBackend`.

To enable our server simulation, we replace the default `XHRBackend` service with the *in-memory web api service* using standard Angular provider registration in the `TohComponent`. We initialize the *in-memory web api* with mock hero data at the same time.

Here are the pertinent details, excerpted from `TohComponent`, starting with the imports:

toh.component.ts (web api imports)

```
import {provide}           from 'angular2/core';
import {XHRBackend}        from 'angular2/http';

// in-memory web api imports
import {InMemoryBackendService,
        SEED_DATA}         from 'a2-in-memory-web-api/core';
import {HeroData}          from '../hero-data';
```

Then we add the following two provider definitions to the `providers` array in component metadata:

toh.component.ts (web api providers)

```
// in-memory web api providers
provide(XHRBackend, { useClass: InMemoryBackendService }), // in-mem server
provide(SEED_DATA,  { useClass: HeroData }) // in-mem server data
```

See the full source code in the live example.

Next Step

Lifecycle Hooks