HTTP

Introduction

Angular comes with its own HTTP library which we can use to call out to external APIs.

When we make calls to an external server, we want our user to continue to be able to interact with the page. That is, we don't want our page to freeze until the HTTP request returns from the external server. To achieve this effect, our HTTP requests are *asynchronous*.

Dealing with *asynchronous* code is, historically, more tricky than dealing with synchronous code. In JavaScript, there are generally three approaches to dealing with async code:

1. Callbacks
2. Promises
3. Observables

In Angular, the preferred method of dealing with async code is using Observables, and so that's what we'll cover in this chapter.

There's a whole chapter on RxJS and Observables: In this chapter we're going to be using Observables and not explaining them much. If you're just starting to read this book at this chapter, you should know that there's a whole chapter on Observables that goes into RxJS in more detail.

In this chapter we're going to:

1. show a basic example of `Http`
2. create a YouTube search-as-you-type component
3. discuss API details about the `Http` library

Sample Code The complete code for the examples in this chapter can be found in the `http` folder of the sample code. That folder contains a `README.md` which gives instructions for building and running the project.

Try running the code while reading the chapter and feel free play around to get a deeper insight about how it all works.

Using `@angular/http`

HTTP has been split into a separate module in Angular. This means that to use it you need to `import` constants from `@angular/http`. For instance, we might import constants from `@angular/http` like this:

```
import {
  // The NgModule for using @angular/http
  HttpModule,

  // the class constants
  Http,
  Response,
  RequestOptions,
  Headers
} from '@angular/http';
```

`import` from `@angular/http`

In our `app.module.ts` we're going to import `HttpModule` which is a convenience collection of modules.

code/http/src/app/app.module.ts

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

In our `NgModule` we will add `HttpModule` to the list of `imports`. The effect is that we will be able to inject `Http` (and a few other modules) into our components.

code/http/src/app/app.module.ts

```
@NgModule({
  declarations: [
    AppComponent,
    SimpleHttpComponent,
    MoreHttpRequestsComponent,
    YouTubeSearchComponent,
    SearchResultComponent,
    SearchBoxComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule // <-- right here
  ],
  providers: [youTubeSearchInjectables],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Notice that we have custom components in `declarations` as well as a custom `provider`. We'll talk about these later in the chapter!

Now we can inject the `Http` service into our components (or anywhere we use DI).

```
class MyFooComponent {
  constructor(public http: Http) {
  }

  makeRequest(): void {
    // do something with this.http ...
  }
}
```

A Basic Request

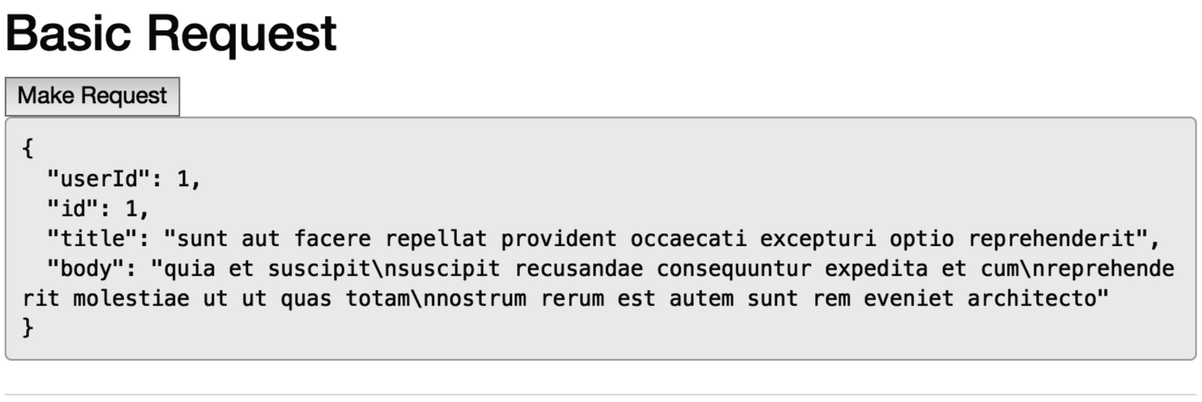
The first thing we're going to do is make a simple `GET` request to the `jsonplaceholder` API[51].

What we're going to do is:

1. Have a `button` that calls `makeRequest`
2. `makeRequest` will call the `http` library to perform a `GET` request on our API
3. When the request returns, we'll update `this.data` with the results of the data, which will be rendered in the view.

Here's a screenshot of our example:

[51] http://jsonplaceholder.typicode.com



```
{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehende
rit molestiae ut ut quas totam\nnostrum rerum est autem sunt rem eveniet architecto"
}
```

Basic Request

Building the `SimpleHttpComponent` Component Definition

The first thing we're going to do is import a few modules and then specify a `selector` for our `@Component`:

code/http/src/app/simple-http/simple-http.component.ts

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

@Component({
  selector: 'app-simple-http',
  templateUrl: './simple-http.component.html'
})
export class SimpleHttpComponent implements OnInit {
  data: Object;
  loading: boolean;

  constructor(private http: Http) {
  }
```

Building the `SimpleHttpComponent template`

Next we build our view:

code/http/src/app/simple-http/simple-http.component.html

```html
<h2>Basic Request</h2>
<button type="button" (click)="makeRequest()">Make Request</button>
<div *ngIf="loading">loading...</div>
<pre>{{data | json}}</pre>
```

Our template has three interesting parts:

1. The `button`
2. The loading indicator
3. The `data`

On the `button` we bind to `(click)` to call the `makeRequest` function in our controller, which we'll define in a minute.

We want to indicate to the user that our request is loading, so to do that we will show `loading...` if the instance variable `loading` is true, using `ngIf`.

The `data` is an `Object`. A great way to debug objects is to use the `json` pipe as we do here. We've put this in a `pre` tag to give us nice, easy to read formatting.

Building the `SimpleHttpComponent` Controller

We start by defining a new `class` for our `SimpleHttpComponent`:

code/http/src/app/simple-http/simple-http.component.ts

```typescript
export class SimpleHttpComponent implements OnInit {
  data: Object;
  loading: boolean;
```

We have two instance variables: `data` and `loading`. This will be used for our API return value and loading indicator respectively.

Next we define our `constructor`:

code/http/src/app/simple-http/simple-http.component.ts

```typescript
  constructor(private http: Http) {
  }
```

The `constructor` body is empty, but we inject one key module: `Http`.

Remember that when we use the `public` keyword in `public http: Http` TypeScript will assign `http` to `this.http`. It's a shorthand for:

```
    // other instance variables here
    http: Http;

    constructor(http: Http) {
      this.http = http;
    }
```

Now let's make our first HTTP request by implementing the `makeRequest` function:

code/http/src/app/simple-http/simple-http.component.ts

```
  makeRequest(): void {
    this.loading = true;
    this.http.request('http://jsonplaceholder.typicode.com/posts/1')
    .subscribe((res: Response) => {
      this.data = res.json();
      this.loading = false;
    });
  }
```

When we call `makeRequest`, the first thing we do is set `this.loading = true`. This will turn on the loading indicator in our view.

To make an HTTP request is straightforward: we call `this.http.request` and pass the URL to which we want to make a `GET` request.

`http.request` returns an `Observable`. We can subscribe to changes (akin to using `then` from a Promise) using `subscribe`.

code/http/src/app/simple-http/simple-http.component.ts

```
    this.http.request('http://jsonplaceholder.typicode.com/posts/1')
    .subscribe((res: Response) => {
```

When our `http.request` returns (from the server) the stream will emit a `Response` object. We extract the body of the response as an `Object` by using `json` and then we set `this.data` to that `Object`.

Since we have a response, we're not loading anymore so we set `this.loading = false`

`.subscribe` can also handle failures and stream completion by passing a function to the second and third arguments respectively. In a production app it would be a good idea to handle those cases, too. That is, `this.loading` should also be set to `false` if the request fails (i.e. the stream emits an error).

Full `SimpleHttpComponent`

Here's what our `SimpleHttpComponent` looks like altogether:

code/http/src/app/simple-http/simple-http.component.ts

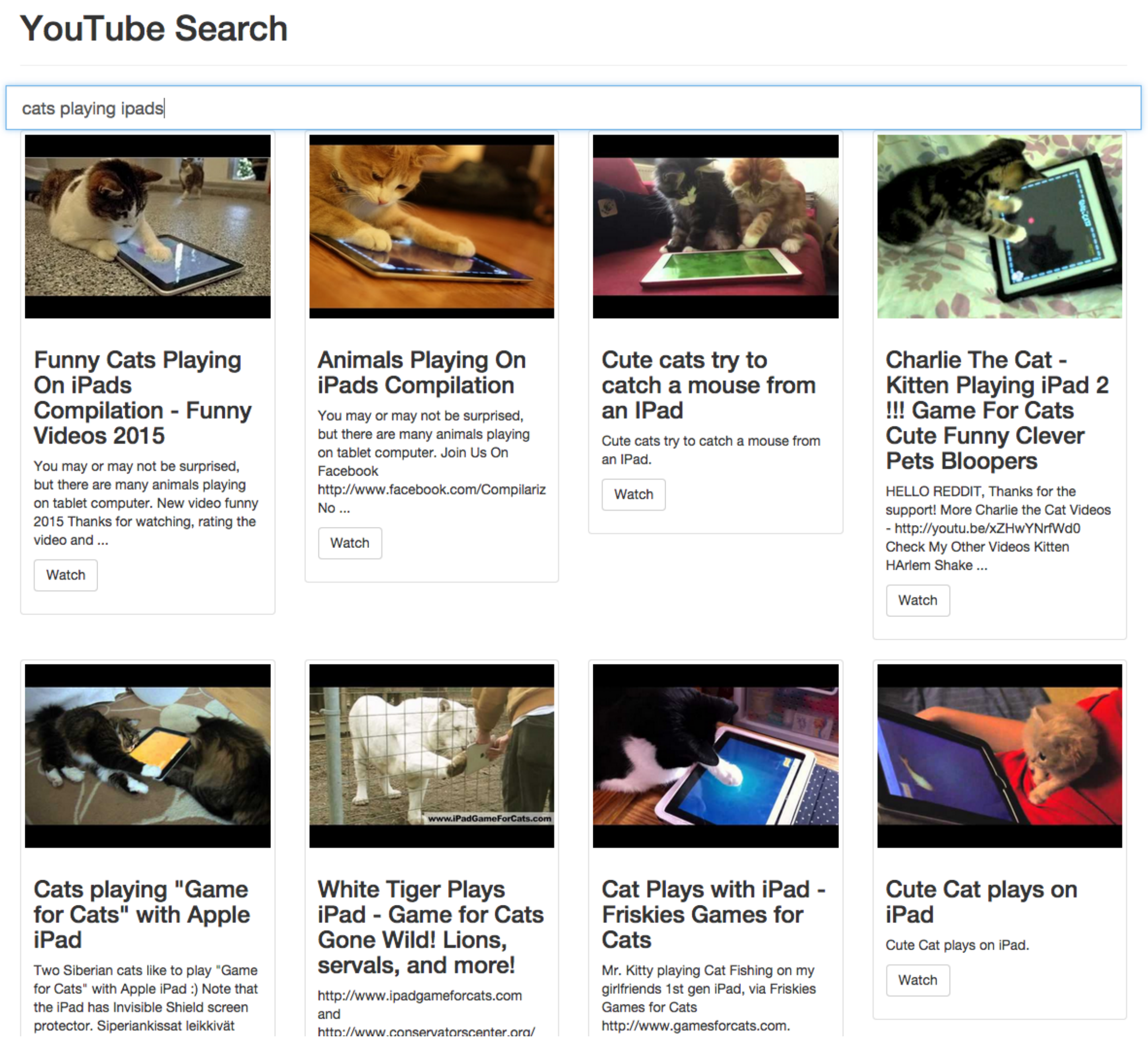
```
import { Component, OnInit } from '@angular/core';
import {Http, Response} from '@angular/http';

@Component({
  selector: 'app-simple-http',
  templateUrl: './simple-http.component.html'
})
export class SimpleHttpComponent implements OnInit {
  data: Object;
  loading: boolean;

  constructor(private http: Http) {
  }

  ngOnInit() {
  }

  makeRequest(): void {
    this.loading = true;
    this.http.request('http://jsonplaceholder.typicode.com/posts/1')
    .subscribe((res: Response) => {
      this.data = res.json();
      this.loading = false;
    });
  }
}
```

Writing a `YouTubeSearchComponent`

The last example was a minimal way to get the data from an API server into your code. Now let's try to build a more involved example.

In this section, we're going to build a way to search YouTube as you type. When the search returns we'll show a list of video thumbnail results, along with a description and link to each video.

Here's a screenshot of what happens when I search for "cats playing ipads":



YouTube Search

cats playing ipads

Funny Cats Playing On iPads Compilation - Funny Videos 2015

You may or may not be surprised, but there are many animals playing on tablet computer. New video funny 2015 Thanks for watching, rating the video and ...

Watch

Animals Playing On iPads Compilation

You may or may not be surprised, but there are many animals playing on tablet computer. Join Us On Facebook http://www.facebook.com/Compilariz No ...

Watch

Cute cats try to catch a mouse from an IPad

Cute cats try to catch a mouse from an IPad.

Watch

Charlie The Cat - Kitten Playing iPad 2 !!! Game For Cats Cute Funny Clever Pets Bloopers

HELLO REDDIT, Thanks for the support! More Charlie the Cat Videos - http://youtu.be/xZHwYNrfWd0 Check My Other Videos Kitten HArlem Shake ...

Watch

Cats playing "Game for Cats" with Apple iPad

Two Siberian cats like to play "Game for Cats" with Apple iPad :) Note that the iPad has Invisible Shield screen protector. Siperiankissat leikkivät

White Tiger Plays iPad - Game for Cats Gone Wild! Lions, servals, and more!

http://www.ipadgameforcats.com and http://www.conservatorscenter.org/

Cat Plays with iPad - Friskies Games for Cats

Mr. Kitty playing Cat Fishing on my girlfriends 1st gen iPad, via Friskies Games for Cats http://www.gamesforcats.com.

Cute Cat plays on iPad

Cute Cat plays on iPad.

Watch

Can I get my cat to write Angular?

For this example we're going to write several things:

1. A `SearchResult` object that will hold the data we want from each result
2. A `YouTubeSearchService` which will manage the API request to YouTube and convert the results to a stream of `SearchResult[]`
3. A `SearchBoxComponent` which will call out to the `YouTube` service as the user types
4. A `SearchResultComponent` which will render a specific `SearchResult`

5. A `YouTubeSearchComponent` which will encapsulate our whole YouTube searching app and render the list of results

Let's handle each part one at a time.

Patrick Stapleton has an excellent repository named angular2-webpack-starter[52]. This repo has an RxJS example which autocompletes Github repositories. Some of the ideas in this section are inspired from that example. It's a fantastic project with lots of examples and you should check it out.

Writing a `SearchResult`

First let's start with writing a basic `SearchResult` class. This class is just a convenient way to store the specific fields we're interested in from our search results.

code/http/src/app/you-tube-search/search-result.model.ts

```
/**
 * SearchResult is a data-structure that holds an individual
 * record from a YouTube video search
 */
export class SearchResult {
  id: string;
  title: string;
  description: string;
  thumbnailUrl: string;
  videoUrl: string;

  constructor(obj?: any) {
    this.id              = obj && obj.id             || null;
    this.title           = obj && obj.title          || null;
    this.description     = obj && obj.description    || null;
    this.thumbnailUrl    = obj && obj.thumbnailUrl   || null;
    this.videoUrl        = obj && obj.videoUrl       ||
                             `https://www.youtube.com/watch?v=${this.id}`;
  }
}
```

This pattern of taking an `obj?: any` lets us simulate keyword arguments. The idea is that we can create a new `SearchResult` and just pass in an object containing the keys we want to specify.

[52]https://github.com/angular-class/angular2-webpack-starter

The only thing to point out here is that we're constructing the `videoUrl` using a hard-coded URL format. You're welcome to change this to a function which takes more arguments, or use the video `id` directly in your view to build this URL if you need to.

Writing the `YouTubeSearchService`

The API

For this example we're going to be using the YouTube v3 search API[53].

In order to use this API you need to have an API key. I've included an API key in the sample code which you can use. However, by the time you read this, you may find it's over the rate limits. If that happens, you'll need to issue your own key.

To issue your own key see this documentation[54]. For the sake of simplicity, I've registered a server key, but you should probably use a browser key if you're going to put your javascript code online.

We're going to setup two constants for our `YouTubeSearchService` mapping to our API key and the API URL:

```
let YOUTUBE_API_KEY: string = "XXX_YOUR_KEY_HERE_XXX";
let YOUTUBE_API_URL: string = "https://www.googleapis.com/youtube/v3/search";
```

Eventually we're going to want to test our app. One of the things we find when testing is that we don't always want to test against production - we often want to test against staging or a development API.

To help with this environment configuration, one of the things we can do is **make these constants injectable**.

Why should we inject these constants instead of just using them in the normal way? Because if we make them injectable we can

1. have code that injects the right constants for a given environment at deploy time and
2. replace the injected value easily at test-time

By injecting these values, we have a lot more flexibility about their values down the line.

In order to make these values injectable, we use the `{ provide: ... , useValue: ... }` syntax like this:

[53]https://developers.google.com/youtube/v3/docs/search/list
[54]https://developers.google.com/youtube/registering_an_application#Create_API_Keys

code/http/src/app/you-tube-search/you-tube-search.injectables.ts

```
import {
  YouTubeSearchService,
  YOUTUBE_API_KEY,
  YOUTUBE_API_URL
} from './you-tube-search.service';

export const youTubeSearchInjectables: Array<any> = [
  {provide: YouTubeSearchService, useClass: YouTubeSearchService},
  {provide: YOUTUBE_API_KEY, useValue: YOUTUBE_API_KEY},
  {provide: YOUTUBE_API_URL, useValue: YOUTUBE_API_URL}
];
```

Here we're specifying that we want to bind `YOUTUBE_API_KEY` "injectably" to the value of `YOUTUBE_-API_KEY`. (Same for `YOUTUBE_API_URL`, and we'll define `YouTubeSearchService` in a minute.)

To get a refresher on the different ways to create 'injectables', checkout the chapter on dependency injection

If you recall, to make something available to be injected throughout our application, we need to put it in `providers` for our `NgModule`. Since we're exporting `youTubeServiceInjectables` here we can use it in our `app.module.ts`

```
// http/app.ts
import { HttpModule } from '@angular/http';
import { youTubeServiceInjectables } from "components/YouTubeSearchComponent";

// ...
// further down
// ...

@NgModule({
  declarations: [
    HttpApp,
    // others ....
  ],
  imports: [ BrowserModule, HttpModule ],
  bootstrap: [ HttpApp ],
  providers: [
    youTubeServiceInjectables // <--- right here
  ]
})
class HttpAppModule {}
```

Now we can inject `YOUTUBE_API_KEY` (from the `youTubeServiceInjectables`) instead of using the variable directly.

`YouTubeSearchService` constructor

We create our `YouTubeSearchService` by making a service `class`:

code/http/src/app/you-tube-search/you-tube-search.service.ts

```
/**
 * YouTubeService connects to the YouTube API
 * See: * https://developers.google.com/youtube/v3/docs/search/list
 */
@Injectable()
export class YouTubeSearchService {
  constructor(private http: Http,
    @Inject(YOUTUBE_API_KEY) private apiKey: string,
    @Inject(YOUTUBE_API_URL) private apiUrl: string) {
    }
```

The `@Injectable` annotation allows us to inject things into this classes `constructor`.

In the `constructor` we inject three things:

1. `Http`
2. `YOUTUBE_API_KEY`
3. `YOUTUBE_API_URL`

Notice that we make instance variables from all three arguments, meaning we can access them as `this.http`, `this.apiKey`, and `this.apiUrl` respectively.

Notice that we explicitly inject using the `@Inject(YOUTUBE_API_KEY)` notation.

`YouTubeSearchService` search

Next let's implement the `search` function. `search` takes a query `string` and returns an `Observable` which will emit a stream of `SearchResult[]`. That is, each item emitted is an *array* of `SearchRe-sults`.

code/http/src/app/you-tube-search/you-tube-search.service.ts

```
  search(query: string): Observable<SearchResult[]> {
    const params: string = [
      `q=${query}`,
      `key=${this.apiKey}`,
      `part=snippet`,
      `type=video`,
      `maxResults=10`
    ].join('&');
    const queryUrl = `${this.apiUrl}?${params}`;
```

We're building the `queryUrl` in a manual way here. We start by simply putting the query params in the `params` variable. (You can find the meaning of each of those values by reading the search API docs[55].)

Then we build the `queryUrl` by concatenating the `apiUrl` and the `params`.

Now that we have a `queryUrl` we can make our request. In this case we are going to use `http.get` instead of `http.request`. While `http.request` can make any kind of request (POST, DELETE< GET, etc.), `http.get` is a shorthand for GET requests:

code/http/src/app/you-tube-search/you-tube-search.service.ts

```
  search(query: string): Observable<SearchResult[]> {
    const params: string = [
      `q=${query}`,
      `key=${this.apiKey}`,
      `part=snippet`,
      `type=video`,
      `maxResults=10`
    ].join('&');
    const queryUrl = `${this.apiUrl}?${params}`;
    return this.http.get(queryUrl)
    .map((response: Response) => {
      return (<any>response.json()).items.map(item => {
        // console.log("raw item", item); // uncomment if you want to debug
        return new SearchResult({
          id: item.id.videoId,
          title: item.snippet.title,
          description: item.snippet.description,
          thumbnailUrl: item.snippet.thumbnails.high.url
        });
```

[55] https://developers.google.com/youtube/v3/docs/search/list

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

Here we take the return value of `http.get` and use `map` to get the `Response` from the request. From that `response` we extract the body as an object using `.json()` and then we iterate over each item and convert it to a `SearchResult`.

If you'd like to see what the raw `item` looks like, just uncomment the `console.log` and inspect it in your browsers developer console.

Notice that we're calling `(<any>response.json()).items`. What's going on here? We're telling TypeScript that we're not interested in doing strict type checking.

When working with a JSON API, we don't generally have typing definitions for the API responses, and so TypeScript won't know that the `Object` returned even has an `items` key, so the compiler will complain.

We could call `response.json()["items"]` and then cast that to an `Array` etc., but in this case (and in creating the `SearchResult`, it's just cleaner to use an `any` type, at the expense of strict type checking

`YouTubeSearchService` Full Listing

Here's the full listing of our `YouTubeSearchService`.

In this chapter we are adding some style using the CSS framework Bootstrap[56]

[56]http://getbootstrap.com

code/http/src/app/you-tube-search/you-tube-search.service.ts

```
/**
 * YouTubeService connects to the YouTube API
 * See: * https://developers.google.com/youtube/v3/docs/search/list
 */
@Injectable()
export class YouTubeSearchService {
  constructor(private http: Http,
    @Inject(YOUTUBE_API_KEY) private apiKey: string,
    @Inject(YOUTUBE_API_URL) private apiUrl: string) {
    }

    search(query: string): Observable<SearchResult[]> {
      const params: string = [
        `q=${query}`,
        `key=${this.apiKey}`,
        `part=snippet`,
        `type=video`,
        `maxResults=10`
      ].join('&');
      const queryUrl = `${this.apiUrl}?${params}`;
      return this.http.get(queryUrl)
      .map((response: Response) => {
        return (<any>response.json()).items.map(item => {
          // console.log("raw item", item); // uncomment if you want to debug
          return new SearchResult({
            id: item.id.videoId,
            title: item.snippet.title,
            description: item.snippet.description,
            thumbnailUrl: item.snippet.thumbnails.high.url
          });
        });
      });
    }
  }
```

Writing the `SearchBoxComponent`

The `SearchBoxComponent` plays a key role in our app: it is the mediator between our UI and the `YouTubeSearchService`.

The `SearchBoxComponent` will :

1. Watch for `keyup` on an `input` and submit a search to the `YouTubeSearchService`
2. Emit a `loading` event when we're loading (or not)
3. Emit a `results` event when we have new results

`SearchBoxComponent @Component` Definition

Let's define our `SearchBoxComponent @Component`:

code/http/src/app/you-tube-search/search-box.component.ts

```
@Component({
  selector: 'app-search-box',
  template: `
    <input type="text" class="form-control" placeholder="Search" autofocus>
  `
})
export class SearchBoxComponent implements OnInit {
  @Output() loading: EventEmitter<boolean> = new EventEmitter<boolean>();
  @Output() results: EventEmitter<SearchResult[]> = new EventEmitter<SearchResul\
t[]>();

  constructor(private youtube: YouTubeSearchService,
              private el: ElementRef) {
  }
```

The `selector` we've seen many times before: this allows us to create a `<app-search-box>` tag.

The two `@Outputs` specify that events will be emitted from this component. That is, we can use the `(output)="callback()"` syntax in our view to listen to events on this component. For example, here's how we will use the `app-search-box` tag in our view later on:

```
<app-search-box
 (loading)="loading = $event"
 (results)="updateResults($event)"
  ></app-search-box>
```

In this example, when the `SearchBoxComponent` emits a `loading` event, we will set the variable `loading` in the parent context. Likewise, when the `SearchBoxComponent` emits a `results` event, we will call the `updateResults()` function, with the value, in the parent's context.

In the `@Component` class we're specifying the properties of the events with the names `loading` and `results`. In this example, each event will have a corresponding `EventEmitter` as an *instance variable of the controller class*. We'll implement that in a few minutes.

For now, remember that `@Component` is like the public API for our component, so here we're just specifying the name of the events, and we'll worry about implementing the `EventEmitters` later.

`SearchBoxComponent` `template` Definition

Our template is straightforward. We have one `input` tag:

code/http/src/app/you-tube-search/search-box.component.ts

```
  template: `
    <input type="text" class="form-control" placeholder="Search" autofocus>
  `
```

`SearchBoxComponent` Controller Definition

Our `SearchBoxComponent` controller is a new class:

code/http/src/app/you-tube-search/search-box.component.ts

```
export class SearchBoxComponent implements OnInit {
  @Output() loading: EventEmitter<boolean> = new EventEmitter<boolean>();
  @Output() results: EventEmitter<SearchResult[]> = new EventEmitter<SearchResul\
t[]>();
```

We say that this class `implements OnInit` because we want to use the `ngOnInit` lifecycle callback. If a class `implements OnInit` then the `ngOnInit` function will be called after the first change detection check.

`ngOnInit` is a good place to do initialization (vs. the `constructor`) because inputs set on a component are not available in the `constructor`.

Here we create the `EventEmitters` for both `loading` and the `results`. `loading` will emit a `boolean` when this search is loading and `results` will emit an array of `SearchResults` when the search is finished.

`SearchBoxComponent` Controller Definition `constructor`

Let's talk about the `SearchBoxComponent constructor`:

code/http/src/app/you-tube-search/search-box.component.ts

```
  constructor(private youtube: YouTubeSearchService,
              private el: ElementRef) {
  }
```

In our `constructor` we inject :

1. Our `YouTubeSearchService` and
2. The element `el` that this component is attached to. `el` is an object of type `ElementRef`, which is an Angular wrapper around a native element.

We set both injections as instance variables.

`SearchBoxComponent` Controller Definition `ngOnInit`

On this input box we want to watch for `keyup` events. The thing is, if we simply did a search after every `keyup` that wouldn't work very well. There are three things we can do to improve the user experience:

1. Filter out any empty or short queries
2. "debounce" the input, that is, don't search on every character but only after the user has stopped typing after a short amount of time
3. discard any old searches, if the user has made a new search

We could manually bind to `keyup` and call a function on each `keyup` event and then implement filtering and debouncing from there. However, there is a better way: turn the `keyup` events into an observable stream.

RxJS provides a way to listen to events on an element using `Rx.Observable.fromEvent`. We can use it like so:

code/http/src/app/you-tube-search/search-box.component.ts

```
  ngOnInit(): void {
    // convert the `keyup` event into an observable stream
    Observable.fromEvent(this.el.nativeElement, 'keyup')
```

Notice that in `fromEvent`:

- the first argument is `this.el.nativeElement` (the native DOM element this component is attached to)
- the second argument is the string `'keyup'`, which is the name of the event we want to turn into a stream

We can now perform some RxJS magic over this stream to turn it into `SearchResults`. Let's walk through step by step.

Given the stream of `keyup` events we can chain on more methods. In the next few paragraphs we're going to chain several functions on to our stream which will transform the stream. Then at the end we'll show the whole example together.

First, let's extract the value of the input tag:

```
.map((e: any) => e.target.value) // extract the value of the input
```

Above says, map over each `keyup` event, then find the event target (`e.target`, that is, our `input` element) and extract the `value` of that element. This means our stream is now a stream of strings.

Next:

```
.filter((text: string) => text.length > 1)
```

This `filter` means the stream will not emit any search strings for which the length is less than one. You could set this to a higher number if you want to ignore short searches.

```
.debounceTime(250)
```

`debounceTime` means we will throttle requests that come in faster than 250ms. That is, we won't search on every keystroke, but rather after the user has paused a small amount.

```
.do(() => this.loading.emit(true))         // enable loading
```

Using `do` on a stream is a way to perform a function mid-stream for each event, but it does not change anything in the stream. The idea here is that we've got our search, it has enough characters, and we've debounced, so now we're about to search, so we turn on `loading`.

`this.loading` is an `EventEmitter`. We "turn on" `loading` by emitting `true` as the next event. We emit something on an `EventEmitter` by calling `next`. Writing `this.loading.emit(true)` means, emit a `true` event on the `loading` `EventEmitter`. When we listen to the `loading` event on this component, the `$event` value will now be `true` (we'll look more closely at using `$event` below).

```
.map((query: string) => this.youtube.search(query))
.switch()
```

We use `.map` to call perform a search for each query that is emitted. By using `switch` we're, essentially, saying "ignore all search events but the most recent". That is, if a new search comes in, we want to use the most recent and discard the rest.

> Reactive experts will note that I'm handwaving here. `switch` has a more specific technical definition which you can read about in the RxJS docs here[57].

For each `query` that comes in, we're going to perform a `search` on our `YouTubeSearchService`.

Putting the chain together we have this:

[57] https://github.com/Reactive-Extensions/RxJS/blob/master/doc/api/core/operators/switch.md

code/http/src/app/you-tube-search/search-box.component.ts

```
  ngOnInit(): void {
    // convert the `keyup` event into an observable stream
    Observable.fromEvent(this.el.nativeElement, 'keyup')
      .map((e: any) => e.target.value) // extract the value of the input
      .filter((text: string) => text.length > 1) // filter out if empty
      .debounceTime(250)                         // only once every 250ms
      .do(() => this.loading.emit(true))         // enable loading
      // search, discarding old events if new input comes in
      .map((query: string) => this.youtube.search(query))
      .switch()
      // act on the return of the search
      .subscribe(
```

The API of RxJS can be a little intimidating because the API surface area is large. That said, we've implemented a sophisticated event-handling stream in very few lines of code!

Because we are calling out to our `YouTubeSearchService` our stream is now a stream of `SearchResult[]`. We can `subscribe` to this stream and perform actions accordingly.

`subscribe` takes three arguments: onSuccess, onError, onCompletion.

code/http/src/app/you-tube-search/search-box.component.ts

```
      .subscribe(
        (results: SearchResult[]) => { // on sucesss
          this.loading.emit(false);
          this.results.emit(results);
        },
        (err: any) => { // on error
          console.log(err);
          this.loading.emit(false);
        },
        () => { // on completion
          this.loading.emit(false);
        }
      );
  }
```

The first argument specifies what we want to do when the stream emits a regular event. Here we emit an event on both of our `EventEmitters`:

1. We call `this.loading.emit(false)`, indicating we've stopped loading

2. We call `this.results.emit(results)`, which will emit an event containing the list of results

The second argument specifies what should happen when the stream has an error event. Here we set `this.loading.emit(false)` and log out the error.

The third argument specifies what should happen when the stream completes. Here we also emit that we're done loading.

`SearchBoxComponent`: Full Listing

All together, here's the full listing of our `SearchBoxComponent` Component:

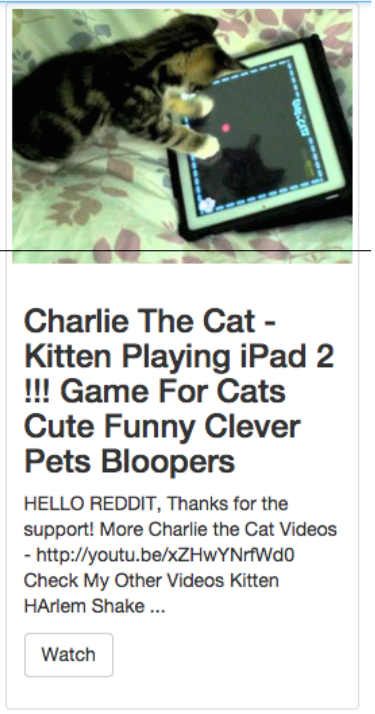
code/http/src/app/you-tube-search/search-box.component.ts

```
@Component({
  selector: 'app-search-box',
  template: `
    <input type="text" class="form-control" placeholder="Search" autofocus>
  `
})
export class SearchBoxComponent implements OnInit {
  @Output() loading: EventEmitter<boolean> = new EventEmitter<boolean>();
  @Output() results: EventEmitter<SearchResult[]> = new EventEmitter<SearchResul\
t[]>();

  constructor(private youtube: YouTubeSearchService,
              private el: ElementRef) {
  }

  ngOnInit(): void {
    // convert the `keyup` event into an observable stream
    Observable.fromEvent(this.el.nativeElement, 'keyup')
      .map((e: any) => e.target.value) // extract the value of the input
      .filter((text: string) => text.length > 1) // filter out if empty
      .debounceTime(250)                         // only once every 250ms
      .do(() => this.loading.emit(true))         // enable loading
      // search, discarding old events if new input comes in
      .map((query: string) => this.youtube.search(query))
      .switch()
      // act on the return of the search
      .subscribe(
        (results: SearchResult[]) => { // on sucesss
          this.loading.emit(false);
          this.results.emit(results);
        },
        (err: any) => { // on error
          console.log(err);
          this.loading.emit(false);
        },
        () => { // on completion
          this.loading.emit(false);
        }
      );
  }
}
```

Writing `SearchResultComponent`

The `SearchBoxComponent` was fairly complicated . Let's handle a **much** easier component now: the `SearchResultComponent`. The `SearchResultComponent`'s job is to render a single `SearchResult`.

Given what we've already covered there aren't any new ideas here, so let's take it all at once:


Charlie The Cat - Kitten Playing iPad 2 !!! Game For Cats Cute Funny Clever Pets Bloopers

HELLO REDDIT, Thanks for the support! More Charlie the Cat Videos - http://youtu.be/xZHwYNrfWd0 Check My Other Videos Kitten HArlem Shake ...

Single Search Result Component

code/http/src/app/you-tube-search/search-result.component.ts

```
import {
  Component,
  OnInit,
  Input
} from '@angular/core';
import { SearchResult } from './search-result.model';


@Component({
  selector: 'app-search-result',
  templateUrl: './search-result.component.html'
})
export class SearchResultComponent implements OnInit {
  @Input() result: SearchResult;

  constructor() { }

  ngOnInit() {
  }

}
```

A few things:

The `@Component` takes a single input `result`, on which we will put the `SearchResult` assigned to this component.

The template shows the title, description, and thumbnail of the video and then links to the video via a button.

code/http/src/app/you-tube-search/search-result.component.html

```html
<div class="col-sm-6 col-md-3">
   <div class="thumbnail">
     <img src="{{result.thumbnailUrl}}">
     <div class="caption">
       <h3>{{result.title}}</h3>
       <p>{{result.description}}</p>
       <p><a href="{{result.videoUrl}}"
             class="btn btn-default" role="button">
             Watch</a></p>
     </div>
   </div>
 </div>
```

The `SearchResultComponent` simply stores the `SearchResult` in the instance variable `result`.

Writing `YouTubeSearchComponent`

The last component we have to implement is the `YouTubeSearchComponent`. This is the component that ties everything together.

`YouTubeSearchComponent @Component`

code/http/src/app/you-tube-search/you-tube-search.component.ts

```ts
@Component({
  selector: 'app-you-tube-search',
  templateUrl: './you-tube-search.component.html'
})
export class YouTubeSearchComponent implements OnInit {
  results: SearchResult[];
  loading: boolean;
```

Our `@Component` decorator is straightforward: use the `selector` `app-you-tube-search`.

`YouTubeSearchComponent` Controller

Before we look at the template, let's take a look at the `YouTubeSearchComponent` controller:

code/http/src/app/you-tube-search/you-tube-search.component.ts

```
export class YouTubeSearchComponent implements OnInit {
  results: SearchResult[];
  loading: boolean;

  constructor() { }
  ngOnInit() { }

  updateResults(results: SearchResult[]): void {
    this.results = results;
    // console.log("results:", this.results); // uncomment to take a look
  }
}
```

This component holds one instance variable: `results` which is an array of `SearchResults`.

We also define one function: `updateResults`. `updateResults` simply takes whatever new `SearchResult[]` it's given and sets `this.results` to the new value.

We'll use both `results` and `updateResults` in our `template`.

`YouTubeSearchComponent template`

Our view needs to do three things:

1. Show the loading indicator, if we're loading
2. Listen to events on the `search-box`
3. Show the search results

Next lets look at our template. Let's build some basic structure and show the loading gif next to the header:

code/http/src/app/you-tube-search/you-tube-search.component.html

```html
<div class='container'>
    <div class="page-header">
      <h1>YouTube Search
        <img
          style="float: right;"
          *ngIf="loading"
          src='assets/images/loading.gif' />
      </h1>
    </div>
```

We only want to show this loading image if `loading` is true, so we use `ngIf` to implement that functionality.

Next, let's look at the markup where we use our `search-box`:

code/http/src/app/you-tube-search/you-tube-search.component.html

```html
    <div class="row">
      <div class="input-group input-group-lg col-md-12">
        <app-search-box
           (loading)="loading = $event"
           (results)="updateResults($event)"
            ></app-search-box>
      </div>
```

The interesting part here is how we bind to the `loading` and `results` outputs. Notice, that we use the `(output)="action()"` syntax here.

For the `loading` output, we run the expression `loading = $event`. `$event` will be substituted with the value of the event that is emitted from the `EventEmitter`. That is, in our `SearchBoxComponent`, when we call `this.loading.emit(true)` then `$event` will be `true`.

Similarly, for the `results` output, we call the `updateResults()` function whenever a new set of results are emitted. This has the effect of updating our components `results` instance variable.

Lastly, we want to take the list of `results` in this component and render a `search-result` for each one:

code/http/src/app/you-tube-search/you-tube-search.component.html

```
    <div class="row">
      <app-search-result
        *ngFor="let result of results"
        [result]="result">
      </app-search-result>
    </div>
</div>
```

`YouTubeSearchComponent` Full Listing

Here's the full listing for the `YouTubeSearchComponent`:

code/http/src/app/you-tube-search/you-tube-search.component.ts

```
@Component({
  selector: 'app-you-tube-search',
  templateUrl: './you-tube-search.component.html'
})
export class YouTubeSearchComponent implements OnInit {
  results: SearchResult[];
  loading: boolean;

  constructor() { }
  ngOnInit() { }

  updateResults(results: SearchResult[]): void {
    this.results = results;
    // console.log("results:", this.results); // uncomment to take a look
  }
}
```

and the template:

code/http/src/app/you-tube-search/you-tube-search.component.html

```html
<div class='container'>
    <div class="page-header">
      <h1>YouTube Search
        <img
          style="float: right;"
          *ngIf="loading"
          src='assets/images/loading.gif' />
      </h1>
    </div>

    <div class="row">
      <div class="input-group input-group-lg col-md-12">
        <app-search-box
           (loading)="loading = $event"
           (results)="updateResults($event)"
            ></app-search-box>
      </div>
    </div>

    <div class="row">
      <app-search-result
        *ngFor="let result of results"
        [result]="result">
      </app-search-result>
    </div>
</div>
```

There we have it! A functional search-as-you-type implemented for YouTube video search! Try running it from the code examples if you haven't already.

`@angular/http` API

Of course, all of the HTTP requests we've made so far have simply been `GET` requests. It's important that we know how we can make other requests too.

Making a `POST` request

Making `POST` request with `@angular/http` is very much like making a `GET` request except that we have one additional parameter: a body.

jsonplaceholder API[58] also provides a convent URL for testing our POST requests, so let's use it for a `POST`:

code/http/src/app/more-http-requests/more-http-requests.component.ts

```
  makePost(): void {
    this.loading = true;
    this.http.post(
      'http://jsonplaceholder.typicode.com/posts',
      JSON.stringify({
        body: 'bar',
        title: 'foo',
        userId: 1
      }))
      .subscribe((res: Response) => {
        this.data = res.json();
        this.loading = false;
      });
  }
```

Notice in the second argument we're taking an `Object` and converting it to a JSON string using `JSON.stringify`.

`PUT` / `PATCH` / `DELETE` / `HEAD`

There are a few other fairly common HTTP requests and we call them in much the same way.

- `http.put` and `http.patch` map to `PUT` and `PATCH` respectively and both take a URL and a body
- `http.delete` and `http.head` map to `DELETE` and `HEAD` respectively and both take a URL (no body)

Here's how we might make a `DELETE` request:

[58] http://jsonplaceholder.typicode.com

code/http/src/app/more-http-requests/more-http-requests.component.ts

```
  makeDelete(): void {
    this.loading = true;
    this.http.delete('http://jsonplaceholder.typicode.com/posts/1')
      .subscribe((res: Response) => {
        this.data = res.json();
        this.loading = false;
      });
  }
```

`RequestOptions`

All of the http methods we've covered so far also take an optional last argument: `RequestOptions`. The `RequestOptions` object encapsulates:

- method
- headers
- body
- mode
- credentials
- cache
- url
- search

Let's say we want to craft a `GET` request that uses a special `X-API-TOKEN` header. We can create a request with this header like so:

code/http/src/app/more-http-requests/more-http-requests.component.ts

```
  makeHeaders(): void {
    const headers: Headers = new Headers();
    headers.append('X-API-TOKEN', 'ng-book');

    const opts: RequestOptions = new RequestOptions();
    opts.headers = headers;

    this.http.get('http://jsonplaceholder.typicode.com/posts/1', opts)
      .subscribe((res: Response) => {
        this.data = res.json();
      });
  }
```

Summary

`@angular/http` is flexible and suitable for a wide variety of APIs.

One of the great things about `@angular/http` is that it has support for mocking the backend which is very useful in testing. To learn about testing HTTP, flip on over to the testing chapter.