**HTTP (AJAX)**

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

**Using @angular/http**

HTTP has been split into a separate module in Angular 2. This means that to use it you need to import

constants from @angular/http.

1 **import** { Http, Response, RequestOptions, Headers } from '@angular/http';

4 **import** {

5 Component

6 } **from** '@angular/core';

7 **import** { bootstrap } **from** '@angular/platform-browser-dynamic';

8 **import** { HTTP\_PROVIDERS } **from** '@angular/http';

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

1 **class** MyFooComponent {

2 constructor(**public** http: Http) {

3 }

4

5 makeRequest(): **void** {

6 *// do something with this.http ...*

7 }

8 }

4 **import** {Component} **from** '@angular/core';

5 **import** {Http, Response} **from** '@angular/http';

6

7 @Component({

8 selector: 'simple-http',

7 @Component({

8 selector: 'simple-http',

9 template: `

10 **<h2>**Basic Request**</h2>**

11 **<button** type="button" (click)="makeRequest()"**>**Make Request**</button>**

12 **<div** \*ngIf="loading"**>**loading...**</div>**

13 **<pre>***{{*data | json*}}***</pre>**

14 `

15 })

Our template has three interesting parts:

1. The button

2. The loading indicator

3. The data

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.

**Building the SimpleHTTPComponent Controller**

We start by defining a new class for our SimpleHTTPComponent:

16 **export class SimpleHTTPComponent** {

17 data: Object;

18 loading: boolean;

20 constructor(public http: Http) {

21 }

Remember that when we use the public keyword in public http: Http TypeScript will

assign http to this.http. It’s a shorthand for:

1 *// other instance variables here*

2 http: Http;

3

4 constructor(http: Http) {

5 **this**.http = http;

6 }

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

23 makeRequest(): void {

24 this.loading = true;

25 this.http.request('http://jsonplaceholder.typicode.com/posts/1')

26 .subscribe((res: Response) => {

27 this.data = res.json();

28 this.loading = false;

29 });

30 }

31 }

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

**Writing a YouTubeSearchComponent**

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 YouTubeService which will manage the API request to YouTube and convert the results to

a stream of SearchResult[]

3. A SearchBox component 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

**Writing a SearchResult**

31 class SearchResult {

32 id: string;

33 title: string;

34 description: string;

35 thumbnailUrl: string;

36 videoUrl: string;

37

38 constructor(obj?: any) {

39 this.id = obj && obj.id || null;

40 this.title = obj && obj.title || null;

41 this.description = obj && obj.description || null;

42 this.thumbnailUrl = obj && obj.thumbnailUrl || null;

43 this.videoUrl = obj && obj.videoUrl ||

44 `https://www.youtube.com/watch?v=*${*this.id*}*`;

45 }

46 }

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.

**Writing the YouTubeService**

**The API**

We’re going to setup two constants for our YouTubeService mapping to our API key and the API

URL:

1 **let** YOUTUBE\_API\_KEY: string = "XXX\_YOUR\_KEY\_HERE\_XXX";

2 **let** YOUTUBE\_API\_URL: string = "https://www.googleapis.com/youtube/v3/search";

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**.

In order to make these values injectable, we use the bind(...).toValue(...) syntax like this:

83 export var youTubeServiceInjectables: Array<any> = [

84 bind(YouTubeService).toClass(YouTubeService),

85 bind(YOUTUBE\_API\_KEY).toValue(YOUTUBE\_API\_KEY),

86 bind(YOUTUBE\_API\_URL).toValue(YOUTUBE\_API\_URL)

87 ];

If you recall, to make something available to be injected throughout our application, we need to make it a dependency at bootstrap. Since we’re exporting youTubeServiceInjectables here we can import it in our app.ts

1 *// http/app.ts*

2 **import** { youTubeServiceInjectables } from "components/YouTubeSearchComponent";

3

4 *// ....*

5 *// further down*

6 bootstrap(HttpApp, [ HTTP\_PROVIDERS, youTubeServiceInjectables ]);

Now we can inject YOUTUBE\_API\_KEY instead of using the variable directly.

**YouTubeService constructor**

We create our YouTubeService by making a class and annotating it as @Injectable:

48 */\*\**

49 *\* YouTubeService connects to the YouTube API*

50 *\* See: \* https://developers.google.com/youtube/v3/docs/search/list*

51 *\*/*

52 @Injectable()

53 export class YouTubeService {

54 constructor(public http: Http,

55 @Inject(YOUTUBE\_API\_KEY) private apiKey: string,

56 @Inject(YOUTUBE\_API\_URL) private apiUrl: string) {

57 }

In the constructor we inject three things:

1. Http

2. YOUTUBE\_API\_KEY

3. YOUTUBE\_API\_URL

Next let’s implement the search function. search takes a query string and returns an Observable which will emit a stream of SearchResult[].

59 search(query: string): Observable**<SearchResult**[]**>** {

60 let params: string = [

61 `q=*${*query*}*`,

62 `key=*${*this.apiKey*}*`,

63 `part=snippet`,

64 `type=video`,

65 `maxResults=10`

66 ].join('**&');**

67 let queryUrl: string = `*${*this.apiUrl*}*?*${*params*}*`;

We’re building the queryUrl in a manual way here. We start by simply putting the query params in the params variable.

Now that we have a queryUrl we can make our request:

68 return this.http.get(queryUrl)

69 .map((response: Response) => {

70 return (<any>response.json()).items.map(item => {

71 // console.log("raw item", item); // uncomment if you want to debug

72 return new SearchResult({

73 id: item.id.videoId,

74 title: item.snippet.title,

75 description: item.snippet.description,

76 thumbnailUrl: item.snippet.thumbnails.high.url

77 });

78 });

79 });

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.

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.

48 /\*\*

49 \* YouTubeService connects to the YouTube API

50 \* See: \* https://developers.google.com/youtube/v3/docs/search/list

51 \*/

52 @Injectable()

53 export class YouTubeService {

54 constructor(public http: Http,

55 @Inject(YOUTUBE\_API\_KEY) private apiKey: string,

56 @Inject(YOUTUBE\_API\_URL) private apiUrl: string) {

57 }

58

59 search(query: string): Observable**<SearchResult**[]**>** {

60 let params: string = [

61 `q=*${*query*}*`,

62 `key=*${*this.apiKey*}*`,

63 `part=snippet`,

64 `type=video`,

65 `maxResults=10`

66 ].join('**&');**

67 let queryUrl: string = `*${*this.apiUrl*}*?*${*params*}*`;

68 return this.http.get(queryUrl)

69 .map((response: Response) => {

70 return (**<any>**response.json()).items.map(item => {

71 // console.log("raw item", item); // uncomment if you want to debug

72 return new SearchResult({

73 id: item.id.videoId,

74 title: item.snippet.title,

75 description: item.snippet.description,

76 thumbnailUrl: item.snippet.thumbnails.high.url

77 });

78 });

79 });

80 }

81 }

**Writing the SearchBox**

The SearchBox will:

1. Watch for keyup on an input and submit a search to the YouTubeService

2. Emit a loading event when we’re loading (or not)

3. Emit a results event when we have new results

89 */\*\**

90 *\* SearchBox displays the search box and emits events based on the results*

91 *\*/*

92

93 @Component({

94 outputs: ['loading', 'results'],

95 selector: 'search-box',

The selector we’ve seen many times before: this allows us to create a <search-box> tag.The outputs key specifies events that will be emitted from this component. That is, we can use the (ouput)="callback()" syntax in our view to listen to events on this component. For example, here’s how we will use the search-box tag in our view later on:

1 <**search-box**

2 (results)="updateResults($event)"

3 (loading)="loading = $event"

4 ></**search-box**>

In this example, when the SearchBox component emits a loading event, we will set the variable

loading in the parent context. Likewise, when the SearchBox emits a results event, we will call

the updateResults() function

**SearchBox Controller Definition**

Our SearchBox controller is a new class:

100 **class SearchBox implements OnInit** {

101 loading: EventEmitter<boolean> = new EventEmitter<boolean>();

102 results: EventEmitter<SearchResult*[]*> = new EventEmitter<SearchResult*[]*>();

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.

**SearchBox Controller Definition constructor**

Let’s talk about the SearchBox constructor:

104 constructor(public youtube: YouTubeService,

105 private el: ElementRef) {

106 }

In our constructor we inject:

1. Our YouTubeService 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.

**SearchBox 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.

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

108 ngOnInit(): void {

109 // convert the `keyup` event into an observable stream

110 Observable.fromEvent(this.el.nativeElement, 'keyup')

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:

1 .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.

1 .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.

1 .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.

1 .**do**(() => **this**.loading.next(**true**)) *// enable loading*

Using do on a stream is 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.next(true) means, emit a true event on the loading EventEmitter.

1 .map((query: string) => **this**.youtube.search(query))

2 .**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” 1⁷¹. That is, if a new search comes

in, we want to use the most recent and discard the rest.

110 Observable.fromEvent(this.el.nativeElement, 'keyup')

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

112 .filter((text: string) => text.length > 1) // filter out if empty

113 .debounceTime(250) // only once every 250ms

114 .do(() => this.loading.next(true)) // enable loading

115 // search, discarding old events if new input comes in

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

117 .switch()

119 .subscribe(

120 (results: SearchResult[]) => { // on sucesss

121 this.loading.next(false);

122 this.results.next(results);

123 },

124 (err: any) => { // on error

125 console.log(err);

126 this.loading.next(false);

127 },

128 () => { // on completion

129 this.loading.next(false);

130 }

131 );

**SearchBox Component: Full Listing**

89 */\*\**

90 *\* SearchBox displays the search box and emits events based on the results*

91 *\*/*

92

93 **@Component**({

94 **outputs**: *[*'loading', 'results'*]*,

95 **selector**: 'search-box',

96 **template**: `

97 <**input type**="text" **class**="form-control" **placeholder**="Search" **autofocus**>

98 `

99 })

100 **class SearchBox implements OnInit** {

101 loading: EventEmitter<boolean> = new EventEmitter<boolean>();

102 results: EventEmitter<SearchResult*[]*> = new EventEmitter<SearchResult*[]*>();

103

104 constructor(public youtube: YouTubeService,

105 private el: ElementRef) {

106 }

107

108 **ngOnInit**(): **void** {

109 // convert the `keyup` event into an observable stream

110 Observable.fromEvent(this.el.nativeElement, 'keyup')

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

112 .filter((text: string) => text.length > 1) // filter out if empty

113 .debounceTime(250) // only once every 250ms

114 .do(() => this.loading.next(true)) // enable loading

115 // search, discarding old events if new input comes in

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

117 .switch()

118 // act on the return of the search

119 .subscribe(

120 (results: SearchResult*[]*) => { // on sucesss

121 this.loading.next(false);

122 this.results.next(results);

123 },

124 (**err**: **any**) => { // on error

125 console.log(err);

126 this.loading.next(false);

127 },

128 () => { // on completion

129 this.loading.next(false);

130 }

131 );

132

133 }

134 }

**Writing SearchResultComponent**

136 @Component({

137 inputs: ['result'],

138 selector: 'search-result',

139 template: `

140 **<div** class="col-sm-6 col-md-3"**>**

141 **<div** class="thumbnail"**>**

142 **<img** src="*{{*result.thumbnailUrl*}}*"**>**

143 **<div** class="caption"**>**

144 **<h3>***{{*result.title*}}***</h3>**

145 **<p>***{{*result.description*}}***</p>**

146 **<p><a** href="*{{*result.videoUrl*}}*"

147 class="btn btn-default" role="button"**>**Watch**</a></p>**

148 **</div>**

149 **</div>**

150 **</div>**

151 `

152 })

153 export class SearchResultComponent {

154 result: SearchResult;

155 }

**Writing YouTubeSearchComponent**

157 @Component({

158 selector: 'youtube-search',

159 directives: [SearchBox, SearchResultComponent],

189 **export class YouTubeSearchComponent** {

190 results: SearchResult*[]*;

191

192 updateResults(results: SearchResult*[]*): void {

193 this.results = results;

194 // console.log("results:", this.results); // uncomment to take a look

195 }

196 }

**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

160 template: `

161 **<div** class='container'**>**

162 **<div** class="page-header"**>**

163 **<h1>**YouTube Search

164 **<img**

165 style="float: right;"

166 \*ngIf="loading"

167 src='*${*loadingGif*}*' **/>**

168 **</h1>**

169 **</div>**

Notice that our img has a src of ${loadingGif} - that loadingGif variable came from a

require statement earlier in the program. Here we’re taking advantage of webpack’s image loading feature.

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

171 **<div** class="row"**>**

172 **<div** class="input-group input-group-lg col-md-12"**>**

173 **<search-box**

174 (loading)="loading = $event"

175 (results)="updateResults($event)"

176 **></search-box>**

177 **</div>**

178 **</div>**

180 **<div** class="row"**>**

181 **<search-result**

182 \*ngFor="let result of results"

183 [result]="result"**>**

184 **</search-result>**

185 **</div>**

186 **</div>**

187 `

188 })

157 @Component({

158 selector: 'youtube-search',

159 directives: [SearchBox, SearchResultComponent],

160 template: `

161 **<div** class='container'**>**

162 **<div** class="page-header"**>**

163 **<h1>**YouTube Search

164 **<img**

165 style="float: right;"

166 \*ngIf="loading"

167 src='*${*loadingGif*}*' **/>**

168 **</h1>**

169 **</div>**

170

171 **<div** class="row"**>**

172 **<div** class="input-group input-group-lg col-md-12"**>**

173 **<search**-box

174 (loading)="loading = $event"

175 (results)="updateResults($event)"

176 **></search-box>**

177 **</div>**

178 **</div>**

179

180 **<div** class="row"**>**

181 **<search**-result

182 \*ngFor="let result of results"

183 [result]="result"**>**

184 **</search-result>**

185 **</div>**

186 **</div>**

187 `

188 })

189 export class YouTubeSearchComponent {

190 results: SearchResult[];

191

192 updateResults(results: SearchResult[]): void {

193 this.results = results;

194 // console.log("results:", this.results); // uncomment to take a look

195 }

196 }

**@angular/http API**

**Making a POST request**

33 makePost(): void {

34 this.loading = true;

35 this.http.post(

36 'http://jsonplaceholder.typicode.com/posts',

37 JSON.stringify({

38 body: 'bar',

39 title: 'foo',

40 userId: 1

41 }))

42 .subscribe((res: Response) => {

43 this.data = res.json();

44 this.loading = false;

45 });

46 }

**PUT / PATCH / DELETE / HEAD**

57 **makeHeaders**(): **void** {

58 let headers: Headers = new Headers();

59 headers.append('X-API-TOKEN', 'ng-book');

60

61 let opts: RequestOptions = new RequestOptions();

62 opts.headers = headers;

63

64 this.http.get('http://jsonplaceholder.typicode.com/posts/1', opts)

65 .subscribe((res: Response) => {

66 this.data = res.json();

67 });

68 }