

Gerard Sans

[Follow](#)

Google Developer Expert | Coding is fun | Just be awesome | Blogger Speaker Trainer Community Leade...
Nov 8, 2015 · 6 min read

Angular 2—Introduction to new HTTP Module

New Data Architecture using Reactive Programming and Observables



Paper.js spiral rasterisation plus effects by gsans

Angular 2 introduces many innovations: performance improvements, the Component Router, sharpened Dependency Injection (DI), lazy loading, async templating, Server rendering (aka Angular Universal), orchestrated animations, mobile development with Native Script; all topped with a solid tooling (thanks to TypeScript and the new Angular CLI) and excellent testing support.

In this post we will focus on the new Angular 2 **HTTP Module**. We are going to look into:

- New Angular 2 Data Architecture using RxJS 5

- Setting up angular/http
- Creating a simple Http Service (using http.get)
- Basic Authentication scenario (using http.post)
- Differences between \$http and angular/http

Angular 2 uses **Reactive Programming** as its core building block. This is based on Asynchronous Data Streams aka **Observables**.


[Demo](#) | [Source](#)

Find the latest Angular 2 content following my feed at [@gerardsans](#).

New Angular 2 Data Architecture using RxJS 5

Reactive Extensions for JavaScript (RxJS) is a reactive streams library that allows you to work with Observables. RxJS combines **Observables**, **Operators** and **Schedulers** so we can subscribe to streams and react to changes using composable operations.

Angular 2 team decided to use **RxJS 5** instead of **RxJS 4**. This is a rewrite focused on improving performance and reducing the API complexity for easier adoption. It is understood version 5 will supersede version 4 by the time Angular 2 is released.

 *Angular 2 uses RxJS 5. Take this into account when looking into specific operators and documentation.*

TypeScript

We are going to use TypeScript in this post. This is a superset of ES6. Read an [introduction to ES6](#) to familiarise with common ES6 features.

We are going to use [basic types](#), [optional parameters](#), [interfaces](#) and [Generics](#). Follow the links to learn more about these features.

Setting up angular/http

In order to use the new Http module in our components we have to import **Http** (see [ES6 import](#)). After that, we can just inject it via Dependency Injection in the constructor. Find an example below.

```

1  import { Injectable } from '@angular/core';
2  import { Http } from '@angular/http';
3
4  @Injectable()
5  export class MyComponent {
6
7  }

```

Besides the changes to our components using **Http**, we also need to include **HttpModule** during bootstrap.

```

1  // main.ts
2  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
3  import { BrowserModule } from '@angular/platform-browser';
4  import { HttpModule } from '@angular/http';
5  import { AppComponent } from './app.component';
6
7  @NgModule({
8    imports: [ BrowserModule, HttpModule ],
9    declarations: [ AppComponent ],
10    bootstrap: [ AppComponent ]

```

In our App, we made **HttpModule** available during bootstrap, using an Array (line 8) for the property **imports** within **ngModule**. Note we imported it from: angular/http (line 4). For Angular 2, Http is no longer included in the core but as a separate file.

Creating a simple Http Service

Let's create a simple example using a Service that reads some data from a JSON file.

```

1  import { Injectable } from '@angular/core';
2  import { Http } from '@angular/http';
3  import 'rxjs/add/operator/map';
4
5  @Injectable()
6  export class PeopleService {
7    constructor(private http: Http) { }
8
9    getPeople() {
10      return http.get('api/people.json')
11        .map(response => response.json());

```

New Dependency Injection

In order to make our service available for DI we added the @Injectable annotation (line 5). This indicates the Dependency Injector that the class has dependencies that should be injected into the constructor when creating an instance of the class.

Annotations allow us to declaratively add metadata to classes and properties for Angular to use. Common annotations are: @Component, @Injectable, @Pipe or @NgModule.

We defined a simple service using a class (line 6). We also preceded it with export to make it accessible to other components. Find below the syntax you use to inject Http into the constructor's http argument.

```
import {Http} from '@angular/http';

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

Http GET

On the constructor we are setting a public property *this.http* and calling http.get. Let's take a look at its definition.

```
http.get(url: string, options?: RequestOptionsArgs)
: Observable<Response>

interface RequestOptionsArgs {
  url?: string;
  method?: string | RequestMethods;
  search?: string | URLSearchParams;
  headers?: Headers;
  body?: string;
}
```

The *http.get* signature above is written in TypeScript and it reads as follows. We need to provide a url and optionally an options object, after the colons we find the corresponding types.

Type information can be used by IDEs to provide features similar to IntelliSense in Visual Studio and type checking.

When providing an *options* object it will follow the RequestOptionsArgs interface. In our example we did not use the options argument. Finally the last bit indicates that http.get returns an *Observable* emitting Response objects.

The first thing you will notice is that instead of returning a promise http.get returns an *Observable*. Finally we used *map* to parse the result into a JSON object. This code is using an arrow function (in ES6) or fat arrow (aka lambda syntax in TypeScript). Let's see how it would translate to ES5 (line 8):

```
// TypeScript/ES6
.map(response => response.json())

// ES5
.map(function(response){ return response.json(); })
```

The result of *map* is also an *Observable* that emits a JSON object containing an Array of people.

Using a Service returning an Observable

Let's take a look at how we would use the new *PeopleService* we just created to display a list of people. See a simplified version of the code below:

```
1  <html>
2    <body>
3      <my-app>Loading...</my-app>
4    </body>
```

```

1  import { PeopleService } from './peopleService';
2
3  @Component({
4    selector: 'my-app',
5    template: `<div *ng-for="let person of people">{{person.n
6  }}
7  export class App {
8    constructor(peopleService: PeopleService) {
9      peopleService.getPeople()
10     .subscribe(
11       people => this.people = people,

```

We imported *PeopleService* (line 1). Following, in the constructor, we injected the service using the same syntax we saw before (line 8).

During *App* instantiation, *peopleService.getPeople()* will return an Observable. We want to display a list of people so we can use *subscribe* to set our local *people* variable when the data is emitted. In this case, we just set the resulting Array into *this.people*.

Explore the examples above using this [Plunker](#).

Basic Authentication scenario

Let's see how a common authentication scenario would be implemented using Angular 2. We will use a simplified example from [Auth0](#) and go over the code below.

```

1  authenticate(username, password) {
2    var body = `username=${username}&password=${password}`;
3    var headers = new Headers();
4    headers.append('Content-Type', 'application/x-www-form-ur
5
6    this.http
7      .post('http://localhost:3001/sessions/create', body, {
8        .map(response => response.json())
9        .subscribe(
10         response => this.storeToken(response.id_token),

```

Http POST

Let's imagine we received the *username* and *password* from a form the user submitted. We would call *authenticate* to log in the user. Once the

user is logged in we will proceed to store the token so we can include it in following requests.

```
http.post(url: string, body: string, options?:  
RequestOptionsArgs)  
: Observable<Response>
```

The [http.post](#) signature above reads as follows. We need to provide a url and a body, both strings and then optionally an options object. In our example we are passing the modified *headers* property. [http.post](#) returns an *Observable*, we use [map](#) to extract the JSON object from the response and [subscribe](#). This will setup our stream as soon as it emits the result. Finally We call *this.storeToken* with the corresponding user token.

Note we are passing the password as clear text. In a real scenario, We would have applied more strict options and would have used https.

Differences between \$http and angular/http

Angular 2 Http by default returns an *Observable* opposed to a *Promise* ([\\$q module](#)) in \$http. This allows us to use more flexible and powerful RxJS operators like [switchMap](#) (flatMapLatest in version 4), [retry](#), [buffer](#), [debounce](#), [merge](#) or [zip](#).


By using *Observables* we improve readability and maintenance of our application as they can respond gracefully to more complex scenarios involving multiple emitted values opposed to only a one-off single value.

Observables vs Promises

When used with **Http**, both implementations provide an easy API for handling requests, but there are some key differences that make **Observables** a superior alternative:

- Promises only accept one value unless we compose multiple promises (Eg: [\\$q.all](#)).
- Promises can't be cancelled.

That's all folks! Hope this post provided you with some insights. Thanks for reading! Have any questions? Ping me on Twitter at [@gerardsans](https://twitter.com/gerardsans)

<p>AngularZone Community</p> <p>Welcome to our community. Our passion is Angular. Join us! 🚀</p>	
--	---

More resources

- [Injecting services into services in Angular 2](#), by Pascal Pretch [@PascalPrecht](#)
- AngularConnect talk “[Angular 2 Data Flow](#)” by Jeff Cross [@jeffbcross](#), Rob Wormald [@robwormald](#) and Alex Rickabaugh [@synalx](#)
- [Angular 2 Observables, Http and separating Services and Components](#), by Ken Rimple [@krimple](#)
- [Introduction to Reactive Extensions](#), previous post



