Display real-time data in Angular

26 June 2018 | By TAMAS PIROS | ANGULAR

In this article, we'll be taking a look at two ways to display real-time data in an Angular application - one using a timeout and the other using sockets.

In this article, we'll be taking a look at two ways to display real-time data in an Angular application. We'll discuss how to push real-time data via a service. One approach will be using sockets while the other will be using the Angular `AsyncPipe` and Observables.

Setting the scene

Often in an application, we work with a backend API service. We create a component, we call an Angular service which in turn calls an API. That API call returns some data and that data is then displayed in the template of the component. This is a very simple scenario. But what happens when data that arrives is updated frequently - think about stock symbols and their values, an online radio that needs to display a new artist & song title. We somehow need to update the component when the data changes at the API level.

Async Pipe & Observables

The first approach that we'll take a look doesn't require any modification at the API level. In light of this, we'll be using the `Async Pipe` . Pipes in Angular work just as pipes work in Linux. They accept an input and produce an output. What the output is going to be is determined by the pipe's functionality. This pipe accepts a promise or an observable as an input, and it can update the template whenever the promise is resolved or when the observable emits some new value. As with all pipes, we need to apply the pipe in the template.

Let's assume that we have a list of products returned by an API and that we have the following service available:

```
// api.service.ts
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable()
export class ApiService {

  constructor(private http: HttpClient) { }

  getProducts() {
    return this.http.get('http://localhost:3000/api/products');
  }

}
```

The code above is straightforward - we specify the `getProducts()` method that returns the HTTP GET call.

It's time to consume this service in the component. And what we'll do here is create an Observable and assign the result of the `getProducts()` method to it. Furthermore, we'll make that call every 1 second, so if there's an update at the API level, we can refresh the template:

```
// some.component.ts
import { Component, OnInit, OnDestroy, Input } from '@angular/core';
import { ApiService } from './../api.service';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/interval';
import 'rxjs/add/operator/startWith';
import 'rxjs/add/operator/switchMap';

@Component({
  selector: 'app-products',
  templateUrl: './products.component.html',
  styleUrls: ['./products.component.css']
})

export class ProductsComponent implements OnInit {
  @Input() products$: Observable<any>;
  constructor(private api: ApiService) { }

  ngOnInit() {
    this.products$ = Observable
      .interval(1000)
      .startWith(0).switchMap(() => this.api.getProducts());
  }

}

```

And last but not least, we need to apply the `async` pipe in our template:

```
<!-- some.component.html -->
<ul>
  <li *ngFor="let product of products$ | async">{{ product.prod_name }} for {{ product.price | currency:'£'}}
</li>
</ul>
```

This way, if we push a new item to the API (or remove one or multiple item(s)) the updates are going to be visible in the component in 1 second.

Sockets

Another approach to creating a component and a service that accepts push data from the server is by implementing sockets. To achieve such functionality, changes need to be performed both at the API and the Client side as well.

API level modifications

At the API level, we need to enable sockets, and one of the most used packages that developers use is `socket.io` which can be installed via `npm i socket.io` .

Here's an implementation of the server using Restify and Socket.io:

```
const restify = require('restify');
const server = restify.createServer();
const products = require('./products');

const io = require('socket.io')(server.server);
let sockets = new Set();

const corsMiddleware = require('restify-cors-middleware');
const port = 3000;

const cors = corsMiddleware({
  origins: ['*'],
});

server.use(restify.plugins.bodyParser());
server.pre(cors.preflight);
server.use(cors.actual);

io.on('connection', socket => {
  sockets.add(socket);
  socket.emit('data', { data: products });
  socket.on('clientData', data => console.log(data));
  socket.on('disconnect', () => sockets.delete(socket));
});

server.get('/', (request, response, next) => {
  response.end();
  next();
});

server.post('/api/products', (request, response) => {
  const product = request.body;
  products.push(product);
  for (const socket of sockets) {
    console.log(`Emitting value: ${products}`);
    socket.emit('data', { data: products });
  }
  response.json(products);
});
server.listen(port, () => console.info(`Server is up on ${port}.`));
```

Note how Restify requires us to use `server.server` *when requiring* `socket.io` *.*

The above code may look complex; however, it is a straightforward implementation. The required `products` file contains an array of objects which represent some data. On the first connection to the server we send data to the requester as well as making sure that we store the socket in a JavaScript `Set` :

```
io.on('connection', socket => {
  sockets.add(socket);
  socket.emit('data', { data: products });
  socket.on('clientData', data => console.log(data));
  socket.on('disconnect', () => sockets.delete(socket));
});
```

When a new product is added (in this case it's just a simple push to the `products` array), then we again, emit the updated array to all the clients who are connected:

```
server.post('/api/products', (request, response) => {
  const product = request.body;
  products.push(product);
  for (const socket of sockets) {
    console.log(`Emitting value: ${products}`);
    socket.emit('data', { data: products });
  }
  response.json(products);
});
```

> *Note, that in this article we're only going through the basics and henceforth the API is kept at an elementary level.*

Client side modifications

At the client side - from our Angular application - we also need to connect to the socket, and for this, we'll be using a package called `socket.io-client` along with its typing. Both of these can be installed via npm: `npm i socket.io-client @types/socket.io-client` .

Once installed we can update our Angular service:

```
// api.service.ts
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import * as socketIo from 'socket.io-client';
import { Observer } from 'rxjs/Observer';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class ApiService {

  observer: Observer<any>;

  getProducts() {
    const socket = socketIo('http://localhost:3000/');
    socket.on('data', response => {
      return this.observer.next(response.data);
    });
    return this.createObservable();
  }

  createObservable() {
    return new Observable(observer => this.observer = observer);
  }

}
```

Here we are creating an observer first, then connect to the socket server running on port 3000 (or whatever port we have specified for the API). If data is emitted from the socket server (which happens on the first load as well as when someone adds a new product), an observable is created. This is what gets passed on to the component and then to the template which still utilises the `async` pipe - the rest of the code does not change.

Adding a new product will also now mean that the list of products is updated.

Conclusion

In this article, we had a look at two ways to achieve real-time data updates in Angular components.

Tamas Piros

Tamas is a Developer Evangelist and experienced Technical Trainer. He has delivered successful training classes, workshops as well as presentations worldwide at prestigious conferences and meetups.

London, United Kingdom https://www.fullstacktraining.com @tpiros

Share this post

Related posts

07 March 2019
404 after refreshing the browser for Angular / Vue.js app

03 December 2018
Add Material Design to an Angular application

26 November 2018
Add Bootstrap to an Angular application

Latest articles

07 March 2019
404 after refreshing the browser for Angular / Vue.js app

26 February 2019
Retrieve only queried element in an object array in MongoDB collection

18 February 2019
An Overview of ES2015 (ES6) Modules

11 February 2019
Remove unused CSS / JavaScript code from your project

05 February 2019
Adaptive image caching based on network speed with Workbox.js

Subscribe

Your email address

Subscribe

Tags

Angular Authentication AWS Blockchain Bootstrap Cloudinary Custom Elements Design DevTools ECMA2015 Express Fetch API Ghost GraphQL HTML JavaScript Machine Learning MarkLogic Material Design MongoDB Network Information API Next.js Node.js Performance Progressive Web Apps Puppeteer PWA Quick Fix Quick Tip Rest Rest API 101 RxJS Security Serverless Service Workers Software Design Patterns SSR Tutorial TypeScript TypeScript-101 Vue.js Web Development Workbox.js