High Performance Angular Grid With Web Sockets

by Dhananjay Kumar MVB · Aug. 06, 18 · Web Dev Zone · Tutorial

You may have come across the requirement to push data in real-time to an Angular Grid. To push data to the browser, you need a technology called WebSocket. You can implement that using Node.js or ASP.NET SignalR. For the purpose of this article, we will use Web Sockets with Node.js.

In the first half of this article, we will create an API which will use Web Sockets to push data to the client, and, in the second half of the article, we will create an Angular application to consume that. In the Angular application, we will use Ignite UI for Angular Grid. However, you can also use a simple HTML table to consume data in real-time from the web socket. In this article, we will learn to consume data in real-time from a Node.js Web Socket in a HTML table as well as Ignite UI Angular Data Grid. We will also look at the difference in performance in these two approaches.

You can learn more about Ignite UI for Angular here.

Node.js API

Let's start by creating a Node.js API. Create a blank folder and add a file called **package.json**. In package.json, add dependencies of:

- Core-js
- Express
- socket.io

More or less your package.json file should look like below:

```
{
  "name": "demo1",
  "version": "1.0.0",
```

```
  "description": "nodejs web socket demo",
  "main": "server.js",
  "dependencies": {
    "core-js": "^2.4.1",
    "express": "^4.16.2",
    "socket.io": "^2.0.4"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Dhananjay Kumar",
  "license": "ISC"
}
```

You can pull data from any type of database such as a relational database, NoSQL database, etc. However, for the purpose of this post, I am going to keep it simple and have hardcoded data in the data.js file. This file will export a JSON array, which we will push using web socket and timer.

Add a file in the folder called data.js and add the following code in it.

data.js

```
module.exports = {
    data: TradeBlotterCDS()
};

function TradeBlotterCDS() {
    return [
        {
            "TradeId": "1",
            "TradeDate": "11/02/2016",
            "BuySell": "Sell",
            "Notional": "50000000",
            "Coupon": "500",
            "Currency": "EUR",
            "ReferenceEntity": "Linde Aktiengesellschaft",
            "Ticker": "LINDE",
            "ShortName": "Linde AG",
            "Counterparty": "MUFJ",
            "MaturityDate": "20/03/2023",
            "EffectiveDate": "12/02/2016",
            "Tenor": "7",
            "RedEntityCode": "DI537C",
            "EntityCusip": "D50348",
```

```
            "EntityType": "Corp",
            "Jurisdiction": "Germany",
            "Sector": "Basic Materials",
            "Trader": "Yael Rich",
            "Status": "Pending"
        }
        // ... other rows of data
    ]
}
```

You can find data with 1200 rows here.

From data.js file, we are returning TradeBlotter data. Now in your project folder, you should have two files: package.json and data.js

At this point in time, run the command, `npm install` , to install all dependencies mentioned in the package.json file. After running the command, you will have the **node_modules** folder in your project folder. Also, add **server.js** file to the project. After all these steps, your project structure should have following files and folders.

- data.js
- server.js
- Node_modules folder

In server.js, we will start by first importing the required modules:

```
const express = require('express'),
    app = express(),
    server = require('http').createServer(app);
io = require('socket.io')(server);
let timerId = null,
    sockets = new Set();
var tradedata = require('./data');
```

Once the required modules are imported, add a route using Express as below:

```
app.use(express.static(__dirname + '/dist'));
```

By connecting the socket, we are performing the following tasks:

1. Fetching data.
2. Starting timer (we will talk about this function later in the post).
3. On the disconnect event the socket is deleted.

```
io.on('connection', socket => {

    console.log(`Socket ${socket.id} added`);
    localdata = tradedata.data;
    sockets.add(socket);
    if (!timerId) {
        startTimer();
    }
    socket.on('clientdata', data => {
        console.log(data);
    });
    socket.on('disconnect', () => {
        console.log(`Deleting socket: ${socket.id}`);
        sockets.delete(socket);
        console.log(`Remaining sockets: ${sockets.size}`);
    });

});
```

Next we have to implement, the `startTimer()` function. In this function, we are using the JavaScript `setInterval()` function and emitting data in each 10 millisecond time frame.

```
function startTimer() {
    timerId = setInterval(() => {
        if (!sockets.size) {
            clearInterval(timerId);
            timerId = null;
            console.log(`Timer stopped`);
        }
        updateData();
        for (const s of sockets) {
            s.emit('data', { data: localdata });
        }

    }, 10);
}
```

We are calling a function, `updateData()` ,which will update data. In this function, we are looping through local data and updating two properties, `Coupon` and `Notional` , with a random number between ranges.

```
function updateData() {
    localdata.forEach(
```

```
        (a) => {
            a.Coupon = getRandomInt(10, 500);
            a.Notional = getRandomInt(1000000, 7000000);
        });
}
```

We have implemented the `getRandomInit` function as shown below:

```
function getRandomInt(min, max) {
    min = Math.ceil(min);
    max = Math.floor(max);
    return Math.floor(Math.random() * (max - min)) + min;
}
```

By putting everything together, sever.js should have following code

Server.js

```
const express = require('express'),
    app = express(),
    server = require('http').createServer(app);
io = require('socket.io')(server);
let timerId = null,
    sockets = new Set();
var tradedata = require('./data');

var localdata;

app.use(express.static(__dirname + '/dist'));

io.on('connection', socket => {

    console.log(`Socket ${socket.id} added`);
    localdata = tradedata.data;
    sockets.add(socket);
    if (!timerId) {
        startTimer();
    }
    socket.on('clientdata', data => {
        console.log(data);
    });
    socket.on('disconnect', () => {
        console.log(`Deleting socket: ${socket.id}`);
        sockets.delete(socket);
        console.log(`Remaining sockets: ${sockets.size}`);
    });
```

```
});

function startTimer() {
    timerId = setInterval(() => {
        if (!sockets.size) {
            clearInterval(timerId);
            timerId = null;
            console.log(`Timer stopped`);
        }
        updateData();
        for (const s of sockets) {
            s.emit('data', { data: localdata });
        }

    }, 10);
}

function getRandomInt(min, max) {
    min = Math.ceil(min);
    max = Math.floor(max);
    return Math.floor(Math.random() * (max - min)) + min;
}

function updateData() {
    localdata.forEach(
        (a) => {
            a.Coupon = getRandomInt(10, 500);
            a.Notional = getRandomInt(1000000, 7000000);
        });
}

server.listen(8080);
console.log('Visit http://localhost:8080 in your browser');
```

We have created Web Sockets in Node.js which is returning data chunks every 10 milliseconds.

Creating an Angular Application

In this step, let's create an Angular application. We are going to use Angular CLI to create an application and then add Ignite UI for Angular Grid. Follow this article to create an Angular application and add Ignite UI for Angular Grid in the application.

If you are following the above article, you need to make some changes in step three in which we are

creating an Angular service to consume an API.

Let's start by installing socket.io-client in our Angular project. To do that run npm install:

```
npm i socket.io-client
```

We will write an Angular service to create the connection with Node.js Web Socket. In app.service.ts, let us start with importing.

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import { Observer } from 'rxjs/Observer';
import { map, catchError } from 'rxjs/operators';
import * as socketIo from 'socket.io-client';
import { Socket } from './interfaces';
```

We have imported the required modules. Later, we will see how socket types are defined inside the interface.ts file. Next, let's create a connection to Web Socket and fetch our next set of data from the response. Before returning the next data chunk from the web socket, we are converting that to an Observable.

```
getQuotes(): Observable < any > {
    this.socket = socketIo('http://localhost:8080');
    this.socket.on('data', (res) => {
        this.observer.next(res.data);
    });
    return this.createObservable();
}

createObservable(): Observable < any > {
    return new Observable<any>(observer => {
        this.observer = observer;
    });
}
```

The above two functions will make connections to web socket, fetch data chunks, and convert that to an observable. Putting everything together, app.service.ts will look like:

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import { Observer } from 'rxjs/Observer';
import { map, catchError } from 'rxjs/operators';
import * as socketIo from 'socket.io-client';
import { Socket } from './interfaces';
```

```
@Injectable()
export class AppService {

    socket: Socket;
    observer: Observer<any>;

    getQuotes(): Observable<any> {
        this.socket = socketIo('http://localhost:8080');
        this.socket.on('data', (res) => {
            this.observer.next(res.data);
        });
        return this.createObservable();
    }

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

    private handleError(error) {
        console.error('server error:', error);
        if (error.error instanceof Error) {
            let errMessage = error.error.message;
            return Observable.throw(errMessage);
        }
        return Observable.throw(error || 'Socket.io server error')
;
    }

}
```

In the service, we are using a type called Socket. We created this type in the file **interfaces.ts** as shown below:

```
export interface Socket {
    on(event: string, callback: (data: any) => void);
    emit(event: string, data: any);
}
```

Now the Angular service which will make a connection to the Node.js Web socket and fetch data from the API as an observable is ready.

This is a normal Angular service and can be consumed in a component in the usual way. Start with

importing it in the module and then injecting that into the component constructor as shown below:

```
constructor(private dataService: AppService) { }
```

We can call the service method to fetch data in the `OnInit` life cycle:

```
ngOnInit() {
    this.sub = this.dataService.getQuotes()
        .subscribe(quote => {
            this.stockQuote = quote;
            console.log(this.stockQuote);
        });
}
```

Putting everything together, the component class will look like the below code.

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { AppService } from './app.service';
import { Subscription } from 'rxjs/Subscription';

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html'
})
export class AppComponent implements OnInit, OnDestroy {

    stockQuote: number;
    sub: Subscription;
    columns: number;
    rows: number;
    selectedTicker: string;

    constructor(private dataService: AppService) { }

    ngOnInit() {
        this.sub = this.dataService.getQuotes()
            .subscribe(quote => {
                this.stockQuote = quote;
                console.log(this.stockQuote);
            });
    }
    ngOnDestroy() {
        this.sub.unsubscribe();
    }
}
```

```
}
```

One important thing you may want to notice is that we are unsubscribing the observable returned in the `OnDestroy` lifecycle hook of the component. In the template, simply render data in a table as below:

```
<table>
    <tr *ngFor="let f of stockQuote">
        <td>{{f.TradeId}}</td>
        <td>{{f.TradeDate}}</td>
        <td>{{f.BuySell}}</td>
        <td>{{f.Notional}}</td>
        <td>{{f.Coupon}}</td>
        <td>{{f.Currency}}</td>
        <td>{{f.ReferenceEntity}}</td>
        <td>{{f.Ticker}}</td>
        <td>{{f.ShortName}}</td>
    </tr>
</table>
```

Since we are rendering data in real-time in a normal HTML table, you may experience flickering and some performance issues. Let's replace the HTML table with Ignite UI for Angular Grid.

Learn more about Ignite UI for Angular Grid here. You can learn to work with Ignite UI for Grid and REST Services in four simple steps here.

You can add Ignite UI Grid in an Angular application as shown below. We have set a data source for the igxGrid using data property binding and then manually added columns to the grid.

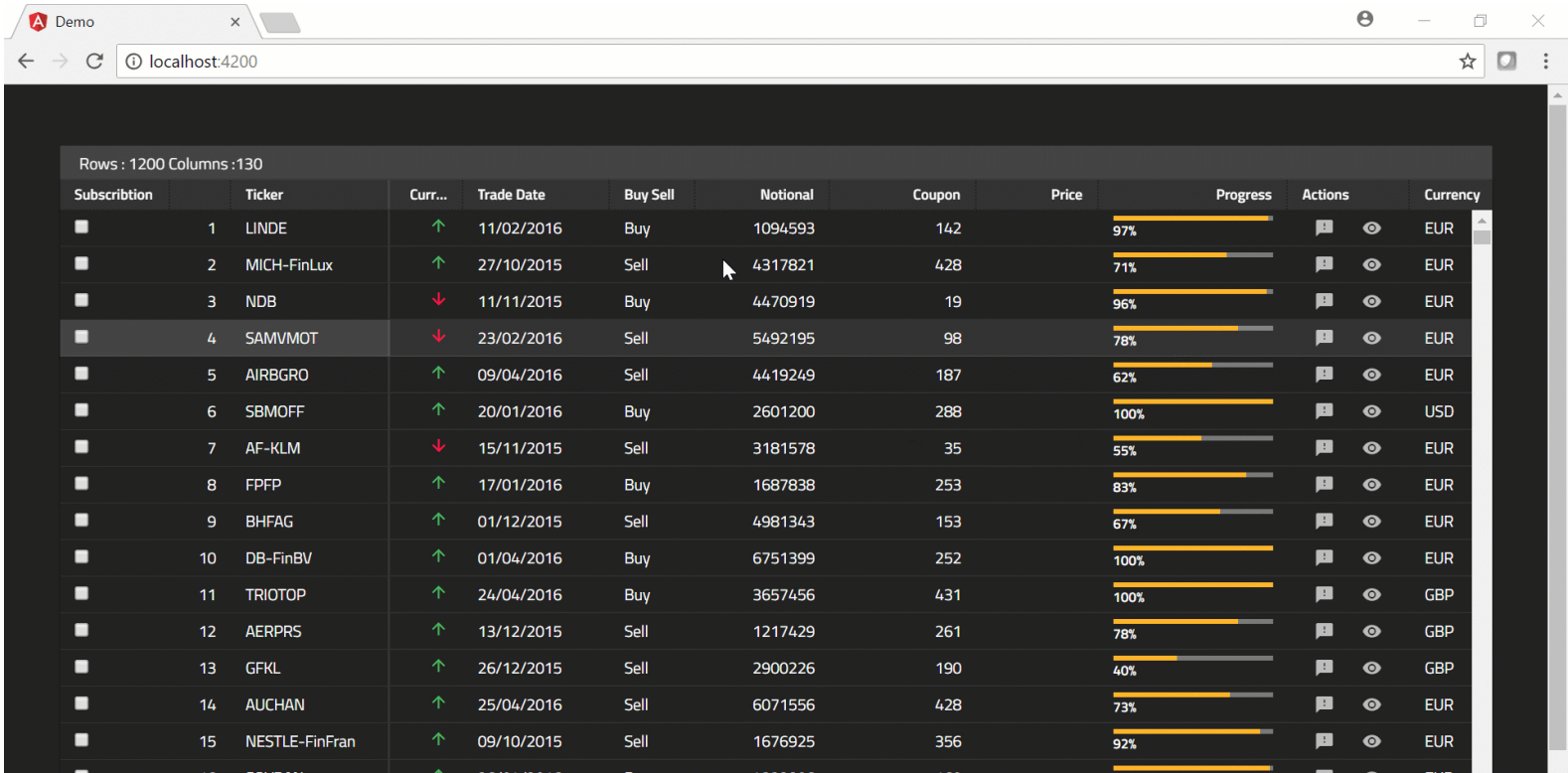
```
<igx-grid [width]="'1172px'" #grid1 id="grid1" [rowHeight]="30" [data]="stockQuote"
          [height]="'600px'" [autoGenerate]="false">
    <igx-column [pinned]="true" [sortable]="true" width="50px" field="TradeId" header="Trade Id" [dataType]="'number'"> </igx-column>
    <igx-column [sortable]="true" width="120px" field="TradeDate" header="Trade Date" dataType="string"></igx-column>
    <igx-column width="70px" field="BuySell" header="Buy Sell" dataType="string"></igx-column>
    <igx-column [sortable]="true" [dataType]="'number'" width="110px" field="Notional" header="Notional">
    </igx-column>
    <igx-column width="120px" [sortable]="true" field="Coupon" header="Coupon" dataType="number"></igx-column>
    <igx-column [sortable]="true" width="100px" field="Price" head
```

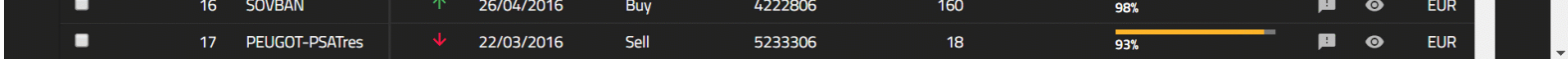
```
9   er="Price" dataType="number">
10      </igx-column>
        <igx-column width="100px" field="Currency" header="Currency" d
11  ataType="string"></igx-column>
        <igx-column width="350px" field="ReferenceEntity" header="Refe
12  rence Entity" dataType="string"></igx-column>
        <igx-column [sortable]="true" [pinned]="true" width="130px" fi
13  eld="Ticker" header="Ticker" dataType="string"></igx-column>
        <igx-column width="350px" field="ShortName" header="Short Name
14  " dataType="string"></igx-column>
15  </igx-grid>
```

A few points you should focus on in the grid we created:

1. By default, virtualization is enabled on the Ignite UI for Angular grid.
2. By setting a sortable property, you can enable sorting on the particular column.
3. By setting a pinned property, you can pin a column to the left of the grid.
4. By setting a data property, you can set the data source of the grid.
5. You can add columns manually by using <igx-column/>.
6. Field and header of <igx-column/> is used to set field property and header of the column.

Now when you run the application, you will find the grid is updating in real-time with data and also not flickering. You will find the grid is updating every 10 milliseconds. You should have the grid running with data getting updated in real-time as shown below:





In this way, you can push data in real-time using the Node.js Web Socket API in an Angular application. I hope you found this article useful. If you like this post, please share it. Also, if you have not checked out Infragistics Ignite UI for Angular Components, be sure to do so! They have 50+ Material-based Angular components to help you code web apps faster.

Like This Article? Read More From DZone

Angular 7 Uploads Backed by Node.js

Deploying a Node.js/Angular 5 Application to Kubernetes With Docker

Login Page Using Angular Material Design

Free DZone Refcard
Drupal 8

Topics: WEB DEV , ANGULAR , NODE.JS , FULLSTACK DEVELOPMENT , TUTORIAL