TUTORIAL

# Create a Real-Time App with Socket.IO, Angular, and Node.js

Angular

By Seth Gwartney

Posted November 14, 2018 141.8k

While this tutorial has content that we believe is of great benefit to our community, we have not yet tested or edited it to ensure you have an error-free learning experience. It's on our list, and we're working on it! You can help us out by using the "report an issue" button at the bottom of the tutorial.

WebSocket is the internet protocol that allows for full duplex communication between a server and clients. This protocol goes beyond the typical HTTP request/response paradigm; with WebSockets, the server may send data to a client without the client initiating a request, thus allowing for some very interesting applications. Most tutorials you'll find on WebSockets have you build a chat app, so I thought we'd tackle the topic a little differently: we'll be building a real-time document collaboration app (a la Google Docs). We'll be using the popular Socket.IO Node.js server framework to accomplish this.

## Project Setup

WebSockets are pretty widely supported, but for the purposes of this demo, I'll be using Angular 7 for the client, and Node.js for the server. You can use pretty much any front-end and server framework, and find the necessary plugins and libraries to make WebSockets work for you. My environment consists of Angular 7.0.4, Node.js 8.11.4, and npm 6.4.1.

## Socket Server

From some base directory, run the following commands to initialize your server project:

```
$ mkdir socket-server
$ cd socket-server
$ mkdir src
$ npm init
$ npm i express socket.io @types/socket.io --save
```

Now create a new file called `app.js` in the `src` directory, and open it using you favorite text editor.

At the top, we'll need our `require` statements for Express and Socket.IO:

```
const app = require('express')();
const http = require('http').Server(app);
const io = require('socket.io')(http);
```

As you can tell, we're using Express and Socket.IO to set up our server. Socket.IO provides a layer of abstraction over native WebSockets. It comes with some nice features, such as a fallback mechanism for older browsers that do not support WebSockets, and the ability to create "rooms". We'll see this in action in a minute.

Our next line will be our in-memory store of documents. Disclaimer: you probably shouldn't do this in production. Use a real database for this.

```
const documents = {};
```

Now let's define what we want our socket server to actually do.

```
io.on("connection", socket => {
  let previousId;
  const safeJoin = currentId => {
```

```
  socket.on("getDoc", docId => {
    safeJoin(docId);
    socket.emit("document", documents[docId]);
  });

  socket.on("addDoc", doc => {
    documents[doc.id] = doc;
    safeJoin(doc.id);
    io.emit("documents", Object.keys(documents));
    socket.emit("document", doc);
  });

  socket.on("editDoc", doc => {
    documents[doc.id] = doc;
    socket.to(doc.id).emit("document", doc);
  });

  io.emit("documents", Object.keys(documents));
});
```

Let's break this down. `.on('...')` is an event listener. The first parameter is the name of the event, and the second one is usually a callback executed when the event fires, with the event payload. The first example we see is when a client connects to the socket server (`connection` is a reserved event type in Socket.IO) . We get a `socket` variable to pass to our callback, to initiate communication to either that one socket, or to multiple sockets (i.e. broadcasting).

I've set up a local function (`safeJoin`) that takes care of joining and leaving "rooms". In this case, when a client has joined a room, they are editing a particular document. So if multiple clients are in the same room, they are all editing the same document. Technically, a socket can be in multiple rooms, but we don't want to let one client edit multiple documents at the same time, so if they switch documents, we need to leave the previous room and join the new room. This little function takes care of that.

There are three event types that our socket is listening for from the client:

1. `getDoc`
2. `addDoc`

And two event types that are emitted by our socket to the client:

1. `document`
2. `documents`

When the client emits the `getDoc` event, the socket is going to take the payload (in our case, it's just an id), join a room with that doc id, and emit the stored document back to the initiating client only. That's where `socket.emit('document', ...)` comes into play.

With the `addDoc` event, the payload is a document object, which, at the moment, consists only of an id generated by the client. We tell our socket to join the room of that id, so that any future edits can be broadcast to anyone in the same room. Next, we want everyone connected to our server to know that there is a new document to work with, so we broadcast to all clients with the `io.emit('documents', ...)` function. You'll notice this same emit also happens whenever a new connection is made. Note the difference between `socket.emit()` and `io.emit()` - the `socket` version is for emitting back to only initiating the client, the `io` version is for emitting to everyone connected to our server.

Finally, with the `editDoc` event, the payload will be the whole document at its state after any keystroke. We'll replace the existing document in the database, and then broadcast the new document to only the clients that are currently viewing that document. We do this by calling `socket.to(doc.id).emit(document, doc)`, which emits to all sockets in that particular room.

After the socket functions are all set up, pick a port and listen on it.

```
http.listen(4444);
```

We now have a fully-functioning socket server for document collaboration! Run `$ node src/app.js` to start it.

## Angular Client App

```
$ ng new socket-app --routing=false --style=SCSS
$ cd socket-app
$ npm i ngx-socket-io --save  ## This is an Angular wrapper over socket.io client libraries
$ ng g class document
$ ng g c document-list
$ ng g c document
$ ng g s document
```

As you can tell, I'm not that creative when naming things.

## App Module

Before your `@NgModule` declaration, add these lines:

```
// ...other imports
import { SocketIoModule, SocketIoConfig } from 'ngx-socket-io';

const config: SocketIoConfig = { url: 'http://localhost:4444', options: {} };
```

Now add to your `imports` array, so it looks like:

```
imports: [
    BrowserModule,
    FormsModule,
    SocketIoModule.forRoot(config)
],
```

This will fire off the connection to our socket server as soon as AppModule loads.

## Document Service

Add a `document.ts` file and write:

models/document.ts

```
export class Document {
    id: string;
    doc: string;
}
```

services/document.service.ts

```
import { Injectable } from '@angular/core';
import { Socket } from 'ngx-socket-io';
import { Document } from '../models/document';

@Injectable({
  providedIn: 'root'
})
export class DocumentService {
  currentDocument = this.socket.fromEvent<Document>('document');
  documents = this.socket.fromEvent<string[]>('documents');

  constructor(private socket: Socket) { }

  getDocument(id: string) {
    this.socket.emit('getDoc', id);
  }

  newDocument() {
    this.socket.emit('addDoc', { id: this.docId(), doc: '' });
  }

  editDocument(document: Document) {
    this.socket.emit('editDoc', document);
  }

  private docId() {
    let text = '';
    const possible = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';

    for (let i = 0; i < 5; i++) {
      text += possible.charAt(Math.floor(Math.random() * possible.length));
    }

    return text;
  }
}
```

The methods here represent each emit the three event types that the socket server is listening for, and the properties `currentDocument` and `documents` represent the events emitted by the socket server, which is consumed on the client as an `Observable`, so we can do a lot of cool things with them if we wanted.



this service

## Document List Component

Let's put the list of documents in a sidenav. Right now, it's only showing the doc id - that random string of characters. Not that pretty, but it gets the job done. In `document-list.component.html`, write the following:

components/document-list/document-list.component.html

```
<div class='sidenav'>
    <span (click)='newDoc()'>New Document</span>
    <span [class.selected]='docId === currentDocId' (click)='loadDoc(docId)' *ngFor='let docId of doc
</div>
```

And give it some style in `document-list.component.scss`:

```
.sidenav {
  position: fixed;
  height: 100%;
  width: 220px;
  top: 0;
  left: 0;
  background-color: #111111;
  overflow-x: hidden;
  padding-top: 20px;
    span {
      padding: 6px  8px  6px  16px;
      text-decoration: none;
      font-size: 25px;
      font-family: 'Roboto', Tahoma, Geneva, Verdana, sans-serif;
      color: #818181;
      display: block;
    }.selected {
      color: #e1e1e1;
    }:hover {
      color: #f1f1f1;
      cursor: pointer;
    }
}
```

In `document-list.component.ts`, add the following in the class definition:

```
import { DocumentService } from 'src/app/services/document.service';

@Component({
  selector: 'app-document-list',
  templateUrl: './document-list.component.html',
  styleUrls: ['./document-list.component.scss']
})
export class DocumentListComponent implements OnInit, OnDestroy {
  documents: Observable<string[]>;
  currentDoc: string;
  private _docSub: Subscription;

  constructor(private documentService: DocumentService) { }

  ngOnInit() {
    this.documents = this.documentService.documents;
    this._docSub = this.documentService.currentDocument.subscribe(doc => this.currentDoc = doc.id);
  }

  ngOnDestroy() {
    this._docSub.unsubscribe();
  }

  loadDoc(id: string) {
    this.documentService.getDocument(id);
  }

  newDoc() {
    this.documentService.newDocument();
  }

}
```

Let's start with the properties. `documents` will be a stream of all available documents. `currentDocId` is the id of the currently selected document. The document list needs to know what document we're on, so we can highlight that doc id in the sidenav. `_docSub` is a reference to the `Subscription` that gives us the current/selected doc. We need this so we can unsubscribe in the `ngOnDestroy` lifecycle method.

You'll notice the methods `loadDoc()` and `newDoc()` don't return or assign anything - remember, these just fire off events to the socket server, which turns around and fires an event back to our Observables. The returned values for getting an existing document or

This will be the document editing surface. Open `document.component.html` and replace the contents with:

```
<textarea [(ngModel)]='document.doc' (keyup)='editDoc()' placeholder='Start typing...'></textarea>
```

To prevent eye injuries, let's change some styles on the default html textarea in `document.component.scss`.

```
textarea {
  position: fixed;
  width: calc(100% - 235px);
  height: 100%;
  right: 0;
  top: 0;
  font-size: 18pt;
  padding-top: 20px;
  resize: none;
  border: none;
  padding: 20px  0px  20px  15px;
}
```

Finally, add the following code in `document.component.ts`.

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { DocumentService } from 'src/app/services/document.service';
import { Subscription } from 'rxjs';
import { Document } from 'src/app/models/document';
import { startWith } from 'rxjs/operators';

@Component({
  selector: 'app-document',
  templateUrl: './document.component.html',
  styleUrls: ['./document.component.scss']
})
export class DocumentComponent implements OnInit, OnDestroy {
  document: Document;
  private _docSub: Subscription;
  constructor(private documentService: DocumentService) { }

  ngOnInit() {
    this._docSub = this.documentService.currentDocument.pipe(
```

```
    ngOnDestroy() {
      this._docSub.unsubscribe();
    }

    editDoc() {
      this.documentService.editDocument(this.document);
    }
  }
```

Similar to the pattern we used in the `DocumentListComponent` above, we're going to subscribe to the changes for our current document, and fire off an event to the socket server whenever we change the current document. This means that we will see all the changes if any other client is editing the same document we are, and vice versa. We use the RxJS `startWith` operator to give a little message to our user when they first open the app.

## AppComponent

Now compose the two custom components by replacing the contents of the `app.component.html` file:

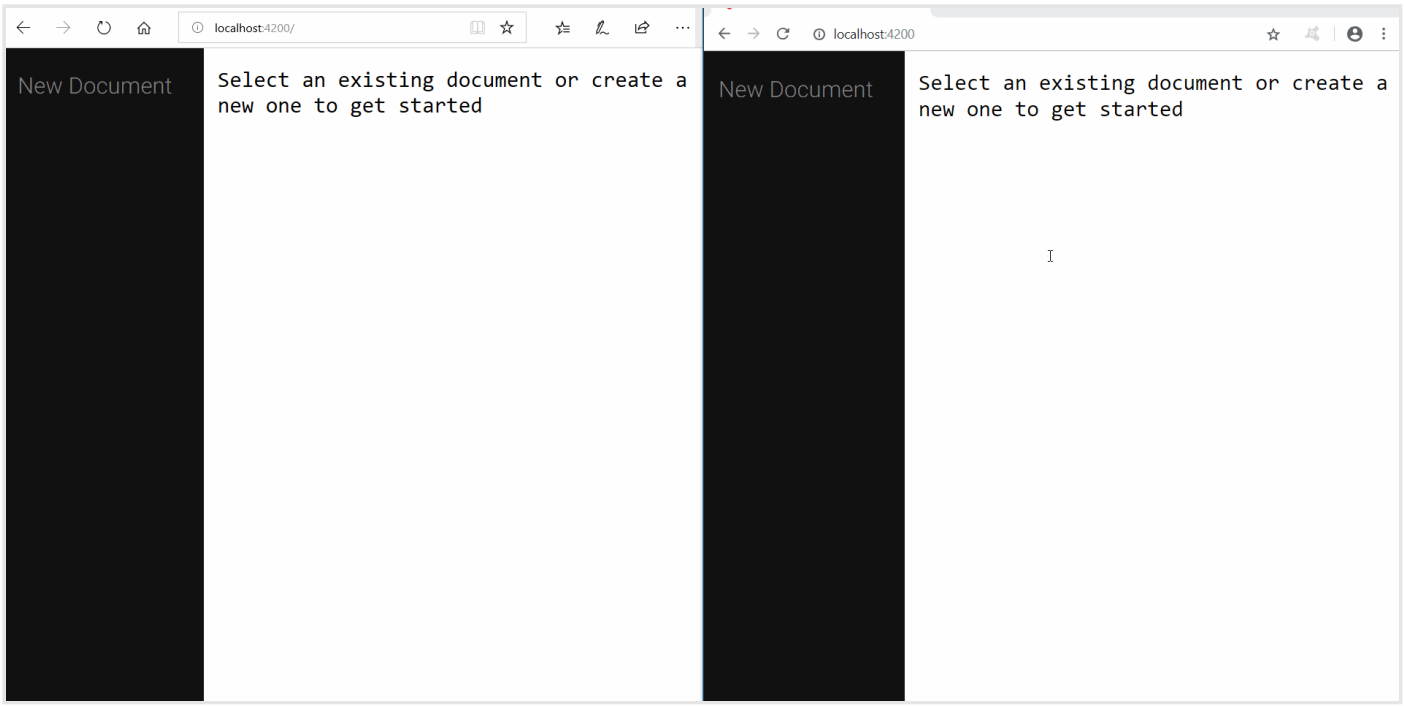
```
<app-document-list></app-document-list>
<app-document></app-document>
```

## Putting it all together

With our socket server running in a separate terminal process, let's start our Angular app:

```
$ ng serve
```

Open more than one instance of http://localhost:4200 (I've done it here in separate browsers for added wow factor) and watch it in action.




Document Collaboration app in action

## Resources

- Socket.IO documentation
- WebSocket on Wikipedia
- Github repo for example project source code
- ngx-socket-io - Angular Module for Socket.IO

**Was this helpful?** Yes No 0

Report an issue

**Sign up for our newsletter.** Get the latest tutorials on SysAdmin and open source topics.

Enter your email address Sign Up

ROLL TO TOP

**Seth Gwartney**

is a Community author on DigitalOcean.

## Related

TUTORIAL

### Component Communication in Angular

Overview of the 3 mains ways to pass some data in between components in Angular: via the Angular ...

TUTORIAL

### Custom Form Validation in Angular

In this post we'll go over how to create a custom validator in Angular for both template-driven and reactive forms.

TUTORIAL

### How To Upgrade Angular Sorting Filters

In the early days of AngularJS, one of the most celebrated features was the ability to filter and sort data on the page using ...

TUTORIAL

### Building Maps in Angular using Leaflet, Part 4: The Shape Service

In this post you'll learn how to use Leaflet with Angular to generate shapes on a dynamic and ...

## 0 Comments

Leave a comment...

Sign In to Comment

**BECOME A CONTRIBUTOR**

You get paid; we donate to tech nonprofits.


**GET OUR BIWEEKLY NEWSLETTER**

Sign up for Infrastructure as a Newsletter.

**HUB FOR GOOD**

Working on improving health and education, reducing inequality, and spurring economic growth? We'd like to

API
Documentation
Release Notes

Community

Tutorials
Q&A
Tools and Integrations
Tags
Product Ideas
Meetups
Write for DOnations
Droplets for Demos
Hatch Startup Program
Shop Swag
Research Program
Open Source
Code of Conduct

Contact

Get Support
Trouble Signing In?
Sales
Report Abuse
System Status