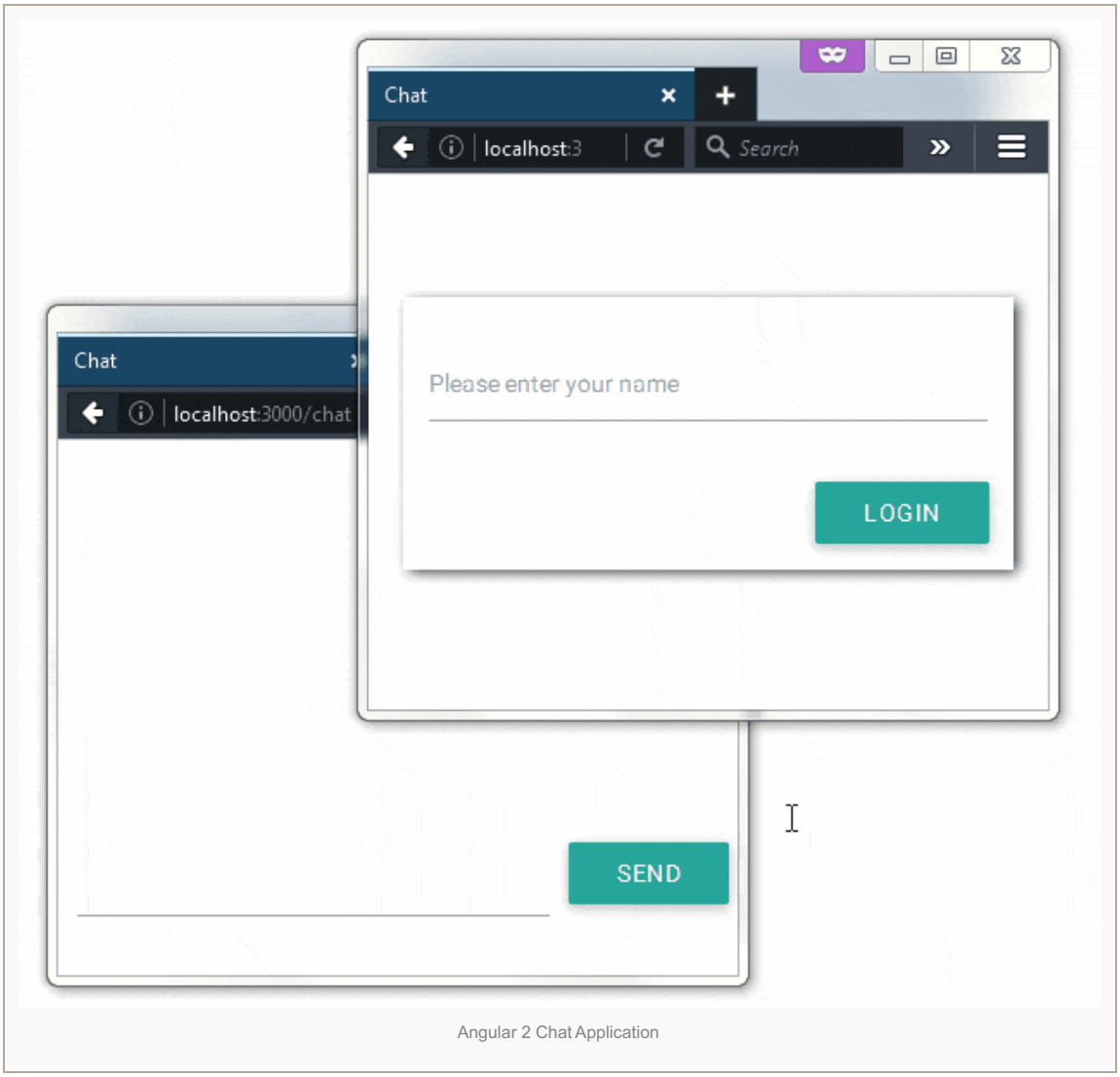
Node.js | 4Dev Web Development Tutorials

4dev.tech /category/node-js/

Posts on this Category

Angular2 Tutorial: Developing a Realtime Chat App



Angular 2 Chat Application

Hi guys, today we're going to make an awesome chat application with Angular2, NodeJS and Socket.io, this tutorial will involve quite a lot of things, and there are also lots of small details, so it will be a little bit longer than usual. To easily understand and make the most of it you should already have some knowledge about all the frameworks involved, if something is unclear for you please let me know in the comments, I'll try to explain it better. With that in mind let's stop wasting time and get started!

Setting up the Environment

I'm going to use Express to create my project, if you're new to express and don't know how to do it click here to see the instructions. After creating the project there will be a file called `package.json` on the project folder, that's where all the dependencies are listed, open this file and copy/paste the following code:

```
{
  "name": "angular2-chat",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "postinstall": "npm run typings install",
    "tsc": "tsc",
    "tsc:w": "tsc -w",
    "start": "concurrent \"node ./bin/www\" \"npm run
tsc:w\"",
    "typings" : "typings"
  },
   "license": "ISC",
  "dependencies": {
    "body-parser": "~1.13.2",
    "cookie-parser": "~1.3.5",
    "debug": "~2.2.0",
    "express": "~4.13.1",
    "jade": "~1.11.0",
    "morgan": "~1.6.1",
    "serve-favicon": "~2.3.0",
    "angular2": "2.0.0-beta.16",
    "systemjs": "0.19.26",
    "es6-promise": "^3.0.2",
    "es6-shim": "^0.35.0",
    "reflect-metadata": "0.1.2",
    "rxjs": "5.0.0-beta.2",
    "zone.js": "0.6.12",
    "socket.io":"1.4.5"
  },
  "devDependencies": {
    "concurrently": "^2.0.0",
    "lite-server": "^2.2.0",
    "typescript": "^1.8.10",
    "typings":"^0.8.1"
  }
}
```

Besides the NodeJS and Express dependencies (which were already there), I've also added Socket.io and everything related to Angular2. If you run the command `npm install` on your terminal it'll download and put all the dependencies on a folder called `/node_modules`. Lots of scripts from this folder will be used on the front-end, so we have to reference them on the `index.html` file, which we're going to create right now.

Open the `/views` folder and create the `index.html` with the following code:

```
<html>
  <head>
    <base href="/">
    <title>Chat</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <script
      src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/1.4.5/socket.io.min.js"></script>
    <script src="scripts/es6-shim/es6-shim.min.js"></script>
    <script src="scripts/systemjs/dist/system-polyfills.js"></script>
    <script src="scripts/angular2/bundles/angular2-polyfills.js"></script>
    <script src="scripts/systemjs/dist/system.src.js"></script>
    <script src="scripts/rxjs/bundles/Rx.js"></script>
    <script src="scripts/angular2/bundles/angular2.dev.js"></script>
    <script src="scripts/angular2/bundles/router.dev.js"></script>
    <script src="javascripts/jquery.js"></script>
    <link rel="stylesheet"

href="https://cdnjs.cloudflare.com/ajax/libs/materialize/0.97.5/css/materialize.min.css">
    <script

src="https://cdnjs.cloudflare.com/ajax/libs/materialize/0.97.5/js/materialize.min.js"></script>
    <script>
      System.config({
        packages: {
          app: {
            format: 'register',
            defaultExtension: 'js'
          }
        }
      });
      System.import('javascripts/app/main.js')
            .then(null, console.error.bind(console));
    </script>
  </head>
  <body>
    <chat-app>Loading...</chat-app>
  </body>
</html>
```

This file is very similar to the one from Angular2 Quickstart, but as you can see I've added three more scripts: Socket.io and the Materialize css and js files (materialize is a CSS framework we're going to use for styling). If you pay attention to this code you should notice that I'm not referencing the scripts from `/node_modules`, instead they're coming from a folder called `/scripts`. But where does this folder come from? I'm going to explain in a moment.

One more thing, in order to make Angular2 work you'll need three more configuration files : `tsconfig.json`, `typings.json` and `main.ts`. All of them can be found on the Angular2 Quickstart.

Now let's open the `app.js` file that Exapress created for us, you'll see that it already has some code, in order to make things easier I'm going to get rid of some lines of code we don't need, after the modifications the file should look like this:

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
var routes = require('./routes/index');
var app = express();

app.use('/scripts', express.static(__dirname + '/node_modules/'));
app.use('/templates', express.static(__dirname +
'/views/templates/'));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', routes);

/* Handle 404. */
app.use(function(req, res, next) {
  res.sendFile(path.join(__dirname, 'views', 'index.html'));
});

module.exports = app;
```

Just after the `var app = express();` line I've configured the paths `/scripts` (which we saw in the `index.html`) and `/templates` to serve some static files from `/node_modules` and `/views/templates` respectively. I had to do this because by default Angular 2 won't have access to these folders. But now, if I request a file from `/scripts` this configuration will make NodeJS understand that it's located on `/node_modules`.

At the end of the code (just above `module.exports`) I've changed how 404 errors are handled, instead of forwarding an error (as it was before) I'm redirecting to `index.html`. This will avoid some problems when we start using Angular2 routes.

Another thing we have to do is to make NodeJS and Angular2 work together, open the file `/routes/index.js` and replace the existing code with the following:

```
var express = require('express');
var router = express.Router();
var path = require('path');

/* GET home page. */
router.get('/', function(req, res, next) {
  res.sendFile(path.join(__dirname, '../', 'views',
'index.html'));
});

module.exports = router;
```

By default it was rendering a `.jade` file (express uses jade as the default view engine), and as I'm not interested in jade for this project I just changed this file a little bit in order to make it redirect to my `index.html` where Angular2 can control the views. For more details about this modification click here.

At this point we've already configured everything we need in our project to start developing our chat app, new let's start talking about how the application is going to work.

Defining the Structure

Our chat app will be composed of three Angular2 components, the first one will be a form where the user will be allowed to enter his name and join the chat, the second will be the chat itself, and the last will be where we're going to configure to routes for controlling which of the other two components should be displayed.

On the back-end we'll have Socket.io being responsible for receiving the messages and delivering them to all the other users, we won't store the messages there, each client will have its own messages list where every received message will be added.

Creating the Socket.io Server

Before we proceed to the front-end we first have to create the Socket.io server and prepare it to receive messages from the clients, to do that you have to create a new file on your project root folder called `socket.js` with the following code:

```
var express = require('express');
var app = express();
var http = require('http');
var server = http.createServer(app);
var io = require('socket.io').listen(server);

var chat = {
 start: function(){
  server.listen(8000);
  io.set("origins", "*:*");

  io.on('connection', function (socket) {          // line
12
   socket.on('newMessage', function (data) {
    socket.emit('chatUpdate',data);
    socket.broadcast.emit('chatUpdate',data);
   });
   socket.on('newUser', function (data) {
    socket.emit('chatUpdate',
     {'userName':'','text':data+' has entered the room'});
    socket.broadcast.emit('chatUpdate',
     {'userName':'','text':data+' has entered the room'});
   });
  });
 }
}

module.exports = chat;
```

Now probably is the best time for me to explain how Socket.io works, according to the official site Socket.IO is a framework that enables real-time bidirectional event-based communication, in other words you can create your own events that will allow the server to communicate with the clients and vice-versa. You can add listeners to these events and also emit events from both sides (client and server), once an event is emitted it will trigger its listener on the other side to execute an specific task. To emit events you can use the function `emit()` passing the event name and your data, and to add a listener you have to use the function `on()` with the event name as the first parameter and the callback function as the second.

Easy, right? Now that you understand how Socket.io works let's analyze our code. The first thing I did here was to create a socket.io server that listens on port 8000, but let's focus on the events, on the server side there is one event that we always have to use in order to start the communication with the clients, this event is called `connection`, it will be automatically emitted by the each client when it connects to the server (obviously), clearly we have to add a listener to proceed with the communication, as you can see in the code I'm doing it on line 12. Now take a look at the callback function associated with this listener, you'll see that it's registering listeners for two more events: `newMessage` and `newUser`. The first one is emitted when a user sends a message, and the second one when a new user joins the chat, the action performed by both is very similar, they will just emit another event called `chatUpdate` to notify all the connected clients about the new message or user. Note that when I emit this event I'm using both `socket.emit()` and `socket.broadcast.emit()`, the first one will notify the client who triggered the event, and the second will do the same to all the other clients.

Let's add the following two lines bellow the `var app = express();` on our `app.js`:

```
var socketServer = require('./socket');
socketServer.start();
```

Perfect! This will make the socket.io server initialize right after the application starts. We're done with the back-end now, let's start

creating the angular2 components.

Creating the User Registration Component

We can finally start writing some Angular2 code, open the `/public/javascripts` folder and create a new one called `/app`, that's where we're going to put our angular2 components. Open this new folder and add a new file called `userRegistration.component.js`, then copy/paste the following code:

```
import {Component} from 'angular2/core';
import {Router} from 'angular2/router';

@Component({
    selector: 'user-registration',
    templateUrl: 'templates/registration.html'
})
export class UserRegistrationComponent {
    userName = '';
    socket = null;

    constructor(
        private _router: Router){}

    ngOnInit() {
        this.socket = io('http://localhost:8000');
    }

    login() {
        if (this.userName !== null){
            sessionStorage.setItem("userName",
this.userName);
            this._router.navigate(['Chat']);
            this.socket.emit('newUser', this.userName);
        }
    }

    keypressHandler(event) {
        if (event.keyCode  === 13){
            this.login();
        }
    }
}
```

This will be the component where the user is going to enter his name and join the chat, I've used the hook `ngOnInit` to connect to the Websocket server, this will automatically emit the `connection` event I just talked about. I also have the `login()` method, which is responsible for putting the provided `userName` on the `sessionStorage` and emit the `newUser` event. Although I'm using the `sessionStorage` to store the user, I don't recommend you to do it, it's not very safe, any user with some javascript knowledge will be able to modify what's in the `sessionStorge`.

Note that I've created a new file to put the template, it's called `registration.html` and it's located on the folder `/views/templates`, here's the code:

```
<style type="text/css">
 .registrationContainer{margin-top:20%;padding:15px;box-shadow: 3px 2px 7px -
1px}
 .registrationContainer button{float: right;}
</style>
<div class="container">
 <div class="col s11 registrationContainer">

        <div class="row">
            <div class="input-field col s12">
                <input [(ngModel)]="userName" id="userName"
                 type="text" (keypress)="keypressHandler($event)">
                <label for="userName">Please enter your name</label>
            </div>
        </div>
     <button (click)="login()" type="submit"
      class="btn waves-effect waves-light col s3"
      name="action" >Login</button>
     <div style="clear:both;"></div>
 </div>
</div>
```

Nothing special here, just an input where the user can type his name and a button to call the `login()` function. The CSS classes you can see here are from materialize.

Creating the Chat Component

Now comes the most important part, the `chat.component.ts`:

```
import {Component} from 'angular2/core';
import {Router} from 'angular2/router';

@Component({
    selector: 'chat',
    templateUrl: 'templates/chat.html'
})
export class ChatComponent {
    message = '';
    conversation = [];
    socket = null;

    constructor(
        private _router: Router){}

    ngOnInit() {
        if (sessionStorage.getItem("userName") === null){
            this._router.navigate(['Registration']);
        }
        this.socket = io('http://localhost:8000');
        this.socket.on('chatUpdate', function(data) {
            this.conversation.push(data);
        }.bind(this));
    }

    send() {
        this.socket.emit('newMessage', {
            'userName':
sessionStorage.getItem("userName"),
            'text': this.message
        });
        this.message = '';
    }

    keypressHandler(event) {
        if (event.keyCode === 13){
            this.send();
        }
    }

    isNewUserAlert(data){
        return data.userName === '';
    }
}
```

Again I'm using the `ngOnInit` hook to connect to server, but this time there are a couple more things I'm doing here, I'm checking the `sessionStorage` to see if it contains the `userName`, if it doesn't the user will be redirected to the registration component, and I'm also adding a listener to the `chatUpdate` event, so when a new message arrives it'll be added to the `conversation` array.

The `send()` method is responsible for triggering the `newMessage` event, it just get the message typed by the user and send it to the server, it'll be executed when the user click on the send button or press enter.

Here's the template:

```
<style type="text/css">
 .userLabel{font-weight:bold;color:#26A69A;margin-right:10px}
 .chatContainer{height:85%;padding:15px;overflow-y:auto }
</style>
<div class="col s12">
 <div class="col s12 chatContainer">
        <ul *ngFor="#msg of conversation" style="margin:0">
            <li >
             <div *ngIf="!isNewUserAlert(msg)">
                 <span  class="userLabel" >{{msg.userName}}:</span>
                 <span>{{msg.text}}</span>
                </div>
             <div *ngIf="isNewUserAlert(msg)">
              <span style="font-weight:bold">{{msg.text}}</span>
             </div>
            </li>
        </ul>
 </div>
 <div class="row">
  <div class="col s9">
          <input id="country" type="text"
           [(ngModel)]="message"
           (keypress)="keypressHandler($event)">
        </div>
  <button (click)="send()"  type="submit"
   class="btn waves-effect waves-light s2"  name="action"
>Send</button>
 </div>
</div>
```

Lastly we have the component `app.component.ts` to control the routing, with `RouteConfig` I'm defining the two possible routes of our app, this allows the other components to redirect from one to another. Note that in the template I have only the `router-outlet` which will basically render the component related to the current route.

```
import {Component} from 'angular2/core';
import {UserRegistrationComponent} from 'javascripts/app/userRegistration.component.js';
import {ChatComponent} from 'javascripts/app/chat.component.js';
import {RouteConfig, ROUTER_DIRECTIVES} from 'angular2/router';

@Component({
    selector: 'chat-app',
    directives: [ROUTER_DIRECTIVES],
    template: `
            <router-outlet></router-outlet>
        `
})
@RouteConfig([
    { path: '/chat', name: 'Chat', component: ChatComponent, useAsDefault:true },
    { path: '/registration', name: 'Registration', component: UserRegistrationComponent
}
])
export class AppComponent {}
```

To make the routes work we have to make a little modification in our `main.ts`, just add the `ROUTER_PROVIDERS` as follows:

```
import {bootstrap}    from 'angular2/platform/browser'
import {AppComponent} from 'javascripts/app/app.component.js'
import {ROUTER_PROVIDERS} from 'angular2/router'

bootstrap(AppComponent, [ROUTER_PROVIDERS]);
```

That's it guys, we've completed our chat app, to run it just execute `npm start` and access it on your browser by typing `localhost:3000`. If you have any doubts please let me know in the comments, till next time!

Creating a NodeJS project with Express

Leonardo JinesNode.js

Express is one of the most popular npm packages out there, it's basically a NodeJS web application framework that provides lots of functionalities and helps you to manage everything and organize your app into MVC on the back-end, and it's very easy to get started with.

To use Express you first need to have NodeJS installed (obviously), if you don't have it yet click here and download the appropriate version for your OS. With nodejs installed you can now create your your project and start using express, in this tutorial I'm just going to cover how to create the project, but in the future I'll probably write more about it.

Express has a tool called `express-generator` that is used to create the project's basic structure, to install it just use the command:

```
npm install express-generator
-g
```

Now that we've installed it globally we're ready to create the project, open your terminal and navigate to the folder where you want to put your project, then simply run the command:

```
express myFirstExpressApp
```

Done! Now you have a folder called myFirstExpressApp (or any other name you've chosen), if you go inside it you'll see the project files and the following folder structure:

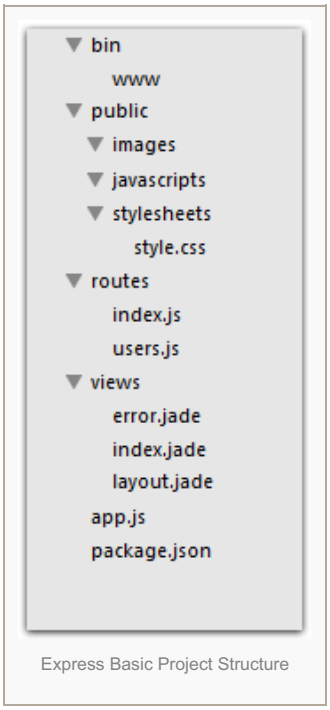
Creating a Live Auction App with Angular 2, Node.js and Socket.IO

In this tutorial we're going to build a very cool Live Auction Application, my goal is to demonstrate how Angular 2, Node.js and Socket.io work together, and as everyone knows Angular 2 will soon come out so I thought it would be great to write a tutorial involving it. Of course our app won't have all the features a real auction app has, this is just an example to demonstrate how to integrate the frameworks I mentioned.

Setting up the Environment

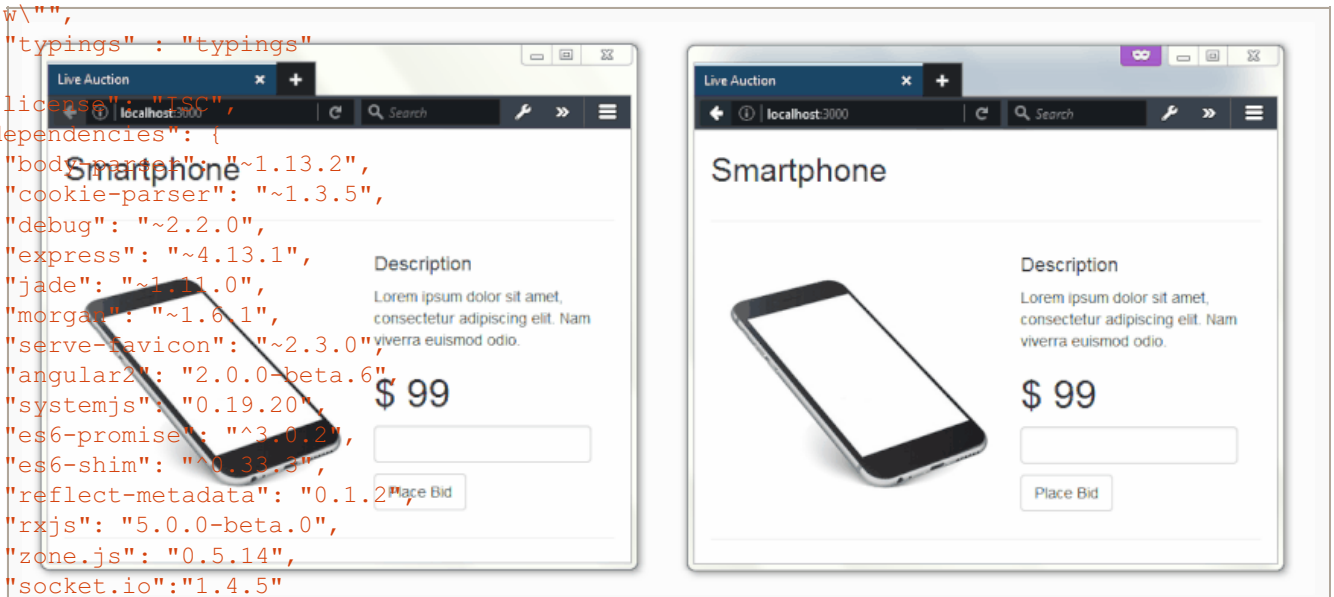
The first step is to configure the environment, let's get started by creating our Node.js project, I'm going to use express to do it, if you don't have it open your terminal and run the command `npm install express-generator -g` to install, then you can create the project with the command `expess {project name}` (replace `{project name}` with your actual project name).

Now you have the basic project structure created for you, but we need to make some changes in order add Angular 2 and Socket.io to the project, first open your `package.json` and paste the following code:



Express Basic Project Structure

```
{
  "name": "auction",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "postinstall": "npm run typings install",
    "tsc": "tsc",
    "tsc:w": "tsc -w",
    "start": "concurrent \"node ./bin/www\" \"npm run
tsc:w\"",
    "typings" : "typings"
  },
   "license": "ISC",
  "dependencies": {
    "body-parser": "~1.13.2",
    "cookie-parser": "~1.3.5",
    "debug": "~2.2.0",
    "express": "~4.13.1",
    "jade": "~1.11.0",
    "morgan": "~1.6.1",
    "serve-favicon": "~2.3.0",
    "angular2": "2.0.0-beta.6",
    "systemjs": "0.19.20",
    "es6-promise": "^3.0.2",
    "es6-shim": "^0.33.3",
    "reflect-metadata": "0.1.2",
    "rxjs": "5.0.0-beta.0",
    "zone.js": "0.5.14",
    "socket.io":"1.4.5"
  },
  "devDependencies": {
    "concurrently": "^1.0.0",
    "lite-server": "^2.0.1",
    "typescript": "^1.7.5",
    "typings":"^0.6.8"
  }
}
```



Angular 2 Auction App

Now create another file called `tsconfig.json` in the project root folder and put the following into the file:

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "system",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators":
true,
    "removeComments": false,
    "noImplicitAny": false
  },
  "exclude": [
    "node_modules",
    "typings/main",
    "typings/main.d.ts"
  ]
}
```

Also in the root folder create the `typings.json` and copy/paste the following code into it:

```
{
  "ambientDependencies": {
    "es6-shim": "github:DefinitelyTyped/DefinitelyTyped/es6-
shim
/es6-shim.d.ts#6697d6f7dadbf5773cb40ecda35a76027e0783b2"
  }
}
```

Ok, we have now created all the configuration files, but before we continue you have to install the dependencies by running the `npm install` command on the terminal.

If you open your `app.js` you'll see that express has already put some code in there, in order to make it as simple as possible let's delete the code we don't need, after the modification your `app.js` should look like this:

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
var routes = require('./routes/index');
var app = express();

app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', routes);

// catch 404 and forward to error handler
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});

module.exports = app;
```

By default express sets jade as the view engine, we've already deleted the lines responsible for that in the `app.js`, but if you check the folder `/views` you'll see some `.jade` files, you can delete them if you want, later we're going to place our Angular 2 HTML pages in this folder, but for now you can just create and leave an empty `index.html` in there.

After getting rid of the `.jade` files we have to tell Node.js that we want to use our `index.html` instead, we can do that by changing a little bit the code in the `routes/index.js` file, just open the file and replace its content with the following code:

```
var express = require('express');
var router = express.Router();
var path = require('path');

/* GET home page. */
router.get('/', function(req, res, next) {
  res.sendFile(path.join(__dirname, '../', 'views',
'index.html'));
});

module.exports = router;
```

To make sure it's working run your project with the command `npm start` then enter the url `http://localhost:3000/` in your

browser, it should display an empty page, write something in the `index.html` you created and refresh the page, if it appears on the browser you've done everything right! Now we're ready to actually start writing some code.

Introducing Socket.IO

As we know, an auction app must be real-time, which means that when a user places a bid all the other connected users must instantly see it, that's where Socket.IO comes in, it allows Node.js to receive and broadcast events to all connected clients, so when a user places a bid Socket.IO will emit an event to the server which will update the product price and broadcast it to the other users.

Now let's see how that looks like in code, copy the following code and paste bellow the `var app = express();` in your `app.js`:

```
var http = require('http');
var server = http.createServer(app);
var io = require('socket.io').listen(server);
server.listen(8000);
io.set("origins", "*:*");

var currentPrice = 99;

io.on('connection', function (socket) {
 socket.emit('priceUpdate',currentPrice);
 socket.on('bid', function (data) {
  currentPrice = parseInt(data);
  socket.emit('priceUpdate',currentPrice);
  socket.broadcast.emit('priceUpdate',currentPrice);
 });
});
```

In the first 5 lines I'm just creating the server that Socket.IO will use to send/receive events, then I'm creating a listener to the `connection` event which will be emitted every time a user connects (we'll get to that in a moment), you can use the function `io.on(event, callback)` to create a listener to any event you want, you just need to pass the event as the fist argument and the callback function as the second, when it receives the event the callback will be executed.

In this case when a user connects I'm immediately emitting the event `priceUpdate` passing the current price, that's because at this moment the client-side doesn't have the current price yet, so that's the first thing we want to send. After that we have to create a listener to the `bid` event that will update the price and broadcast to the other users. Note that I'm calling `socket.emit()` and `socket.broadcast.emit()`, the first one will return the updated price to the user that triggered the event, and the second will do the same to all the other users.

Writing our First Angular 2 Component

Now that we have our server-side ready we can get started with Angular 2, let's create a folder called `app` inside `public/javascripts/` on our project, that's where we're going to put all our Angular 2 code. Now create a file and name it `app.component.ts`, then paste the following code into it:

```
import {Component} from 'angular2/core';

@Component({
    selector: 'auction-app',
    templateUrl: 'templates/product.html'
})

export class AppComponent {
    price: number = 0.0;
    socket = null;
    bidValue = '';

    constructor(){
        this.socket = io('http://localhost:8000');
        this.socket.on('priceUpdate',
function(data){
            this.price = data;
        }.bind(this));
    }

    bid(){
        this.socket.emit('bid', this.bidValue);
        this.bidValue = '';
    }
}
```

When declaring a `@Component`, you have to pass an object containing two attributes `selector` and `templateUrl` (or just `template`), the `selector` is the name we're going to use to call our component, in this case it's `auction-app`, and the `templateUrl` is the HTML implementation, you can put your HTML code directly on the object or you can put it in a separated file like I did.

We also have in this code the `AppComponent` class, that's kind of the 'controller' of our component, in the constructor I'm connecting to the Socket.IO from the server using the method `io('http://localhost:8000')`, this will emit the `connection` event we saw before, then I'm creating a listener to the `priceUpdate` event, the server will emit it right after the connection, remember? When Angular receives the updated price it will assign it to the `price` variable. This class also contains the `bid()` function, it will just emit the `bid` event passing the `bidValue`.

I forgot to mention that I created the `/template` folder inside `/views`, the problem is that our component doesn't have access to it, to solve this problem we have to make this folder a static route on our server, open your `app.js` and add this line:

```
app.use('/templates', express.static(__dirname +
'/views/templates/'));
```

Now let's take a look on our template, I'll not show the full HTML code here, just the lines that interact with our `AppComponent` class.

```
<h1>$ {{price}}</h1>
<input  [(ngModel)]="bidValue" >
<button (click)="bid()">Place
Bid</button>
```

There is not much to explain here, the syntax resembles a lot the first Angular, our `input` uses the attribute `[(ngModel)]` to reference the `bidValue` variable from our class, and the button uses the `(click)` to call the function `bid()`.

To be able to launch our application we have to create a file called `main.ts` on the `public/javascripts/` folder, just copy/paste the following into the file:

```
import {bootstrap}    from
'angular2/platform/browser'
import {AppComponent} from './app.component.js'

bootstrap(AppComponent);
```

Finally we can write our `index.html`, you have already created it in the beginning, remember? It's located on the `/views` folder, open it and add the following code:

```
<html>
  <head>
    <title>Live Auction</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/1.4.5/socket.io.min.js"></script>
    <script src="scripts/es6-shim/es6-shim.min.js"></script>
    <script src="scripts/systemjs/dist/system-polyfills.js"></script>
    <script src="scripts/angular2/bundles/angular2-polyfills.js"></script>
    <script src="scripts/systemjs/dist/system.src.js"></script>
    <script src="scripts/rxjs/bundles/Rx.js"></script>
    <script src="scripts/angular2/bundles/angular2.dev.js"></script>
    <script src="javascripts/jquery.js"></script>
    <script src="javascripts/bootstrap.min.js"></script>
    <link href="stylesheets/bootstrap.min.css" rel="stylesheet">
    <link href="stylesheets/portfolio-item.css" rel="stylesheet">
    <script>
      System.config({
        packages: {
          app: {
            format: 'register',
            defaultExtension: 'js'
          }
        }
      });
      System.import('javascripts/app/main.js')
            .then(null, console.error.bind(console));
    </script>
  </head>
  <body>
    <auction-app>Loading...</auction-app>
  </body>
</html>
```

Once again we have to edit the `app.js` file to be able to load the scripts directly from the `node_modules` folder, add the following line like we did before:

```
app.use('/scripts', express.static(__dirname +
'/node_modules/'));
```

This file is pretty simple, we have a script that configures System.js to load our application, and in the body we can call our component by the selector we specified before: `auction-app`.

Now we're done!! You can run your application by opening your terminal and running the command `npm start`, the result should similar to the gif in the beginning of this tutorial.

Just leave a comment if you have any problems, I'll be glad to help!!

How to Execute a .jar File with Node.js Child Processes

Leonardo JinesNode.js

You probably have already come across situations where your application had to execute other applications or native commands from the OS. Node.js can easily do that by spawning **child processes**. To show you how to do that I'm going to give you an example of how to execute a .jar file with node.

First let's create a very simple java application for us to use in this example, this is the one I'm going to use:

```
package javaapp;

public class JavaApp{

    public static void main(String[] args) {
        System.out.println("This is the Java application
output!!");
    }

}
```

This class is very simple but it will be enough to demonstrate how child processes work, just copy the code, compile it, and place the jar file into folder of your choice.

Now let's create a js file containing the following code:

```
var exec = require('child_process').exec;
var child = exec('java -jar
C:/javaApp.jar',
  function (error, stdout, stderr){
    console.log('Output -> ' + stdout);
    if(error !== null){
      console.log("Error -> "+error);
    }
});

module.exports = child;
```

Note how easy it is, first I required the module **child_process** and then I called the function `exec()` passing as an argument a native OS command to execute a java file, which is : `java -jar C:/javaApp.jar` . In the callback function I have 3 variables: `error`, `stdout`, `stderr`, but I'm going to use only the first two. The `javaApp.jar` output will be stored on the `stdout` variable, and the errors will be in the `error` variable, obviously.

To run this code just open the terminal, navigate to your js file location and execute the command:

```
node app.js
```

If everything is right you should see the output:

```
Output -> This is the Java application
output!!
```

Tutorial: Integrating AngularJS with NodeJS

I know this tutorial may seem a little bit too basic for a lot of you, but these are two of the most popular javascript frameworks at the moment, as a result of that there are lots of people just getting started with NodeJS and AngularJS, and it's a very common to see people asking how to integrate both, so this will be a very simple beginner level tutorial in which I'll give an example to demonstrate how AngularJS and NodeJS can work together.

I'm assuming you have already set your enviroment, so we are going to start by creating our NodeJS project, to do that I'm going to use express, if you don't have it yet just install via npm with the following command line:

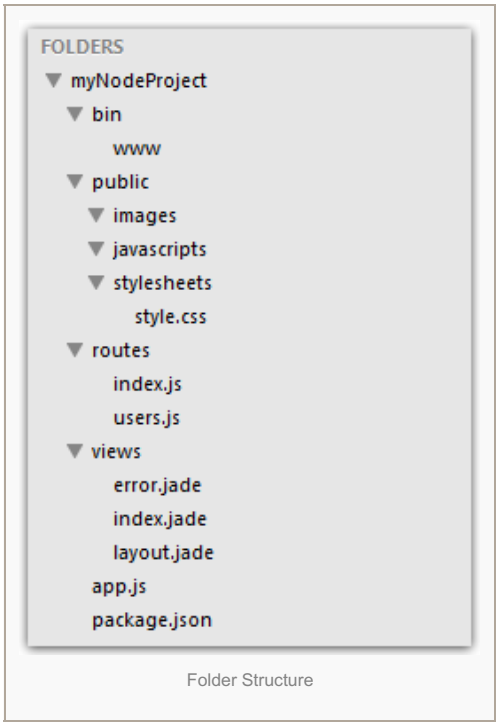
```
npm install express-generator
-g
```

With express installed simply navigate via terminal to the folder you want your project to be located and execute the command **express** passing your project name as a parameter, like this:

```
express myNodeProject
```

Integrating AngularJS with NodeJS

Now that your project is created, you can open it with the text editor of your choice, I'm going to use Sublime Text for this tutorial. When you do it you'll see that express has already created the basic project structure for you:



Folder Structure

By default, express uses Jade as the view engine, as you can see in the **views** folder we already have some jade files, which we are not going use, you can get rid of them later if you want. We'll need to make some changes here to be able to use angularjs. Let's begin by opening the file **routes/index.js**, it contains the following code:

```
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express'
});
});

module.exports = router;
```

That's the code responsible for rendering and redirecting to the **index.jade** file, what we need to do here is to change it a little bit and make it redirect to our angularjs **index.html** file (which we are going to create in a moment). After the changes the code should look like this:

```
var express = require('express');
var router = express.Router();
var path = require('path');

/* GET home page. */
router.get('/', function(req, res, next) {
  res.sendFile(path.join(__dirname, '../', 'views',
'index.html'));
});

module.exports = router;
```

Now we need to create our **index.html** file, I'm going to put it in the **views** folder with the jade files. This is how my HTML code looks like:

```
<!DOCTYPE html>
<html ng-app="angularjsNodejsTutorial">
    <head>
        <title>Integrating AngularJS with NodeJS</title>
        <script
src="http://cdnjs.cloudflare.com/ajax/libs/angular.js/1.4.2/angular.js"></script>
        <script src="javascripts/app.js" type="text/javascript"></script> <!-- not created
yet -->
    </head>
    <body >
        <div ng-controller="myController">
        </div>
    </body>
</html>
```

In this file you can use not only AngularJS, but any javascript library you want, if you run your project you'll see that nodejs is already redirecting to this file, you can now create you angularjs module and start writing some angularjs code as usual.

At this point you are already using angular and node in your application, but they are kind of independent from each other, there are no communication between them. To finish up this tutorial I'm going to show how AngularJS can make a request to NodeJS. But before we do that, we need to come back to the **index.js** file (the one we just modified) and create a function to return some data, which will be called when we make the request from our angularjs controller, just copy the following code and paste it right above the **module.exports** on your **index.js**.

```
router.get('/data', function(req,res){
 res.json([{"id": 1, "name": "Mymm", "city": "Pantano do
Sul"},
        {"id": 2, "name": "Skyble", "city": "Guilmaro"},
        {"id": 3, "name": "Tagfeed", "city": "Gnosjö"},
        {"id": 4, "name": "Realcube", "city": "Jrashen"},
        {"id": 5, "name": "Bluejam", "city": "Zhangjiawo"},
        {"id": 6, "name": "Jayo", "city": "Obonoma"},
        {"id": 7, "name": "Cogidoo", "city": "Sungsang"},
        {"id": 8, "name": "Avavee", "city": "Diawara"},
        {"id": 9, "name": "Tagtune", "city": "Monywa"},
        {"id": 10, "name": "Centimia", "city": "Retkovci"}]);
});
```

What is happening here is that the **router.get** function is assigning a function to the url **'/data'**, so when the user types **'/data'** in the browser, node will call this function, which is doing nothing more than returning a json, it could be getting and handling data from the database, but as I want to keep it simple, a static json will do the job.

Now let's create our **app.js** file, as you can see in the HTML code, it's already referenced there. I'm going to put it on the **public/javascript** folder.

```
var app = angular.module('angularjsNodejsTutorial',[]);
app.controller('myController', function($scope, $http) {
    $scope.data = [];
    var request = $http.get('/data');
    request.success(function(data) {
        $scope.data = data;
    });
    request.error(function(data){
        console.log('Error: ' + data);
    });
});
```

This is also a very straightforward code, I'm using the function **$http.get** with the argumentet **'/data'** to make the request, then I'm assigning the result to **$scope.data**.

Now we just need to modify a little bit our HTML to make it iterate over our data and show it on the screen, just add this to the div with the **ng-controller**:

```
<ul ng-repeat="item in data">
 <li>Name: {{item.name}}, City:
{{item.city}}</li>
</ul>
```

That's it!! Now just run your project by executing **npm start** on the terminal and you'll be able to access the application on your browser by typing **localhost:3000**, if you did everything right you should see a page with the data from our json.

Hope this tutorial was helpful, till the next one!!

Node.js Tutorial: Reading and Writing files

Reading and writing files is a very trivial task, no matter what programming language you use you probably have already came across a situation where you had to do it, and it's not different for **node.js** developers, luckily node.js makes this task very easy for us, with the file system(fs) module we can easily read files with very few lines of code as the following example:

```
1  var fs = require('fs');
2
3  fs.readFile('C:\\file.txt', 'utf8', displayData);
4
5  function displayData(err,data){
6    console.log(data);
7  }
```

In the example above I've just used the **fs.readFile** function to read a file located on my 'C:\', besides the file path I also provided the encoding and a callback which will be called once the file content is ready.

Now that we already know how to read files, let's learn how to write files, it's as easy as reading them, take a look at the following code:

```
1   var fs = require('fs');
2
3   fs.writeFile('C:\\file.txt', 'My Content', showMessage);
4
5   function showMessage(err){
6     if(err) {
7       console.log(err);
8     }else{
9       console.log('Your data has been successfully written!');
10    }
11  }
```

It's basically the same thing we saw on the first example, except that instead of using the **readFile** function I used the **writeFile**, also the second argument is the content I want to write on the file, and once again we have a callback that will execute after the content is written.

If you run this code more than once you should note that before writing your content it erases everything that is already on the file, to prevent this from happening you should use the function **appendFile** instead of **writeFile**.

Tutorial: Creating Node.js modules

In this short tutorial I'm going to explain how you can create your own modules in Node.js, as everyone knows we cannot keep our entire code in only one js file, as your application grows it'll be almost impossible for you to maintain your code, for this reason it's essential that we separate our code in modules, this way it'll ~~hopefully~~ always be organized and easy to maintain.

Node.js makes it very easy for us to create custom modules, let's suppose we need to put 2 functions in a separated module, we just need to create a new js file like this:

newModule.js

```
1  exports.function1 = function(){
2   console.log('Function 1');
3  }
4
5  exports.function2 = function(){
6   console.log('Function 2');
7  }
```

As you can see it's a very straightforward code, all I did was to create 2 functions and add them to module.exports, unless you don't want your functions to be visible to other modules it's very important that you add them to exports.

Now you can require this new module in another module as follows:

app.js

```
1  var newModule = require('./newModule');
2
3  newModule.function1();
4  newModule.function2();
```

When requiring a module you have to pass its full path to the require function, in this example I'm assuming that the 2 js files are in the same folder. After requiring your module you're done! Now this app.js file has access to all the functions you've exported from your new module.