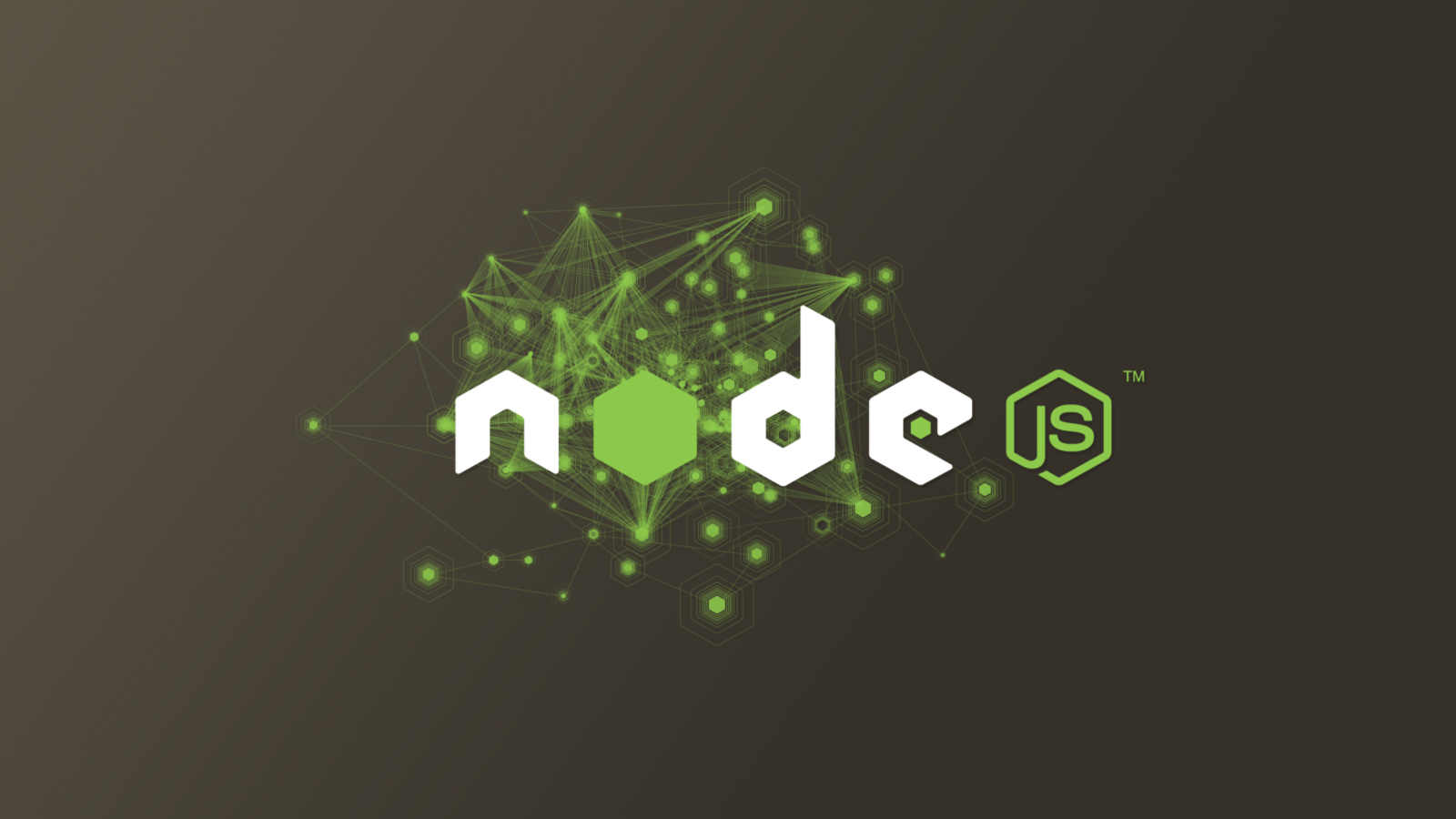
**How to build a real time chat application in Node.js using Express, Mongoose and Socket.io**



In this tutorial, we will use the Node.js platform to build a **real time chat application** that sends and shows messages to a recipient instantly without any page refresh. We will use the JavaScript framework Express.js and the libraries Mongoose and Socket.io to achieve this.

Before we start, lets have a quick look at the basics of Node.js

**Node.js**

[Node.js](https://en.wikipedia.org/wiki/Node.js) is an open-source, cross-platform JavaScript run-time environment that executes JavaScript code outside the browser. The most important advantage of using Node is that we can use JavaScript as both a front-end and back-end language.

As we know, JavaScript was used primarily for client-side scripting, in which scripts were embedded in a webpage’s HTML and run client-side by a JavaScript engine in the user’s web browser.

Node.js lets developers use JavaScript to write Command Line tools and for server-side scripting — running scripts server-side to produce dynamic web page content before the page is sent to the user’s web browser.

To install node:

[https://nodejs.org/en/download](https://nodejs.org/en/download/)/

Even though the node is single threaded it’s still faster to use asynchronous functions. For example, Node can process other things while a file is being read off disk, or while waiting for an HTTP request to complete. The asynchronous behaviour can be implemented using callbacks. Also the JavaScript works well with JSON and No-SQL databases.

**NPM Modules**

Nodejs allows the modules of libraries to be included in the application. These modules can be user-defined or third party modules.

The third party modules can be installed using the following command:

npm install module\_name

and the installed modules can be used using the **require()** function:

var module = require(‘module\_name’)

In Node apps we will be using a package.json file to maintain the module versions. This file can be created by this command:

npm init

and the packages must be installed as follows:

npm install -s module\_name

There are many frameworks that can be added as modules to our Node application. These will be explained further on as needed.

**Simple Chat Application**

The app must allow multiple users to chat together. The messages must be updated without refreshing the page. For simplicity we will be avoiding the authentication part.

We can start by creating a new project directory and moving into it. Then we can initiate our project by the following command:

npm init

This will prompt us to enter details about our project.

After this a **package.json** file will be created:

{  
 “name”: “test”,  
 “version”: “1.0.0”,  
 “description”: “”,  
 “main”: “index.js”,  
 “scripts”: {  
 “test”: “echo \”Error: no test specified\” && exit 1"  
 },  
 “author”: “”,  
 “license”: “ISC”  
}

Our app directory is now set.

The first thing we need to create is a server. In order to create that, we will be making use of a framework named **Express.**

**Express.js**

Express.js, or simply Express, is a web application framework for Node.js. Express [provides a robust set of features for web and mobile applications](https://expressjs.com/). Express provides a thin layer of fundamental web application features, without obscuring Node.js features.

We will install Express.js using the following command:

npm install -s express

Inside the package.json file a new line will be added:

dependencies”: {  
 “express”: “⁴.16.3”  
 }

Next we will create a **server.js** file.

In this file we need to require Express and create a reference to a variable from an instance of Express. Static contents like HTML, CSS or JavaScript can be served using express.js:

var express = require(‘express’);

var app = express();

and we can start listening to a port using the code:

var server = app.listen(3000, () => {  
 console.log(‘server is running on port’, server.address().port);  
});

Now we need to create an HTML file index.html that displays our UI. I have added bootstrap and JQuery cdn.

//index.html

<!DOCTYPE html>  
<html>  
<head>  
 <! — include bootstap and jquery cdn →  
</head>  
<body>  
<div class=”container”>  
 <br>  
 <div class=”jumbotron”>  
 <h1 class=”display-4">Send Message</h1>  
 <br>  
 <input id = “name” class=”form-control” placeholder=”Name”>  
 <br>  
 <textarea id = “message” class=”form-control” placeholder=”Your Message Here”>  
</textarea>  
 <br>  
 <button id=”send” class=”btn btn-success”>Send</button>  
 </div>  
 <div id=”messages”>

</div>  
</div>  
<script>

</script>  
</body>  
</html>

Please note that the empty *<script> </script>*tag will be the place where we will write the client side JavaScript code.

In-order to tell Express that, we will be using a static file. We will add a new line inside **server.js:**

app.use(express.static(\_\_dirname));

We can run the server.js using the command

node ./server.js

or a package called **nodemon**, so that the changes made in the code will be automatically detected. We will download nodemon using the command

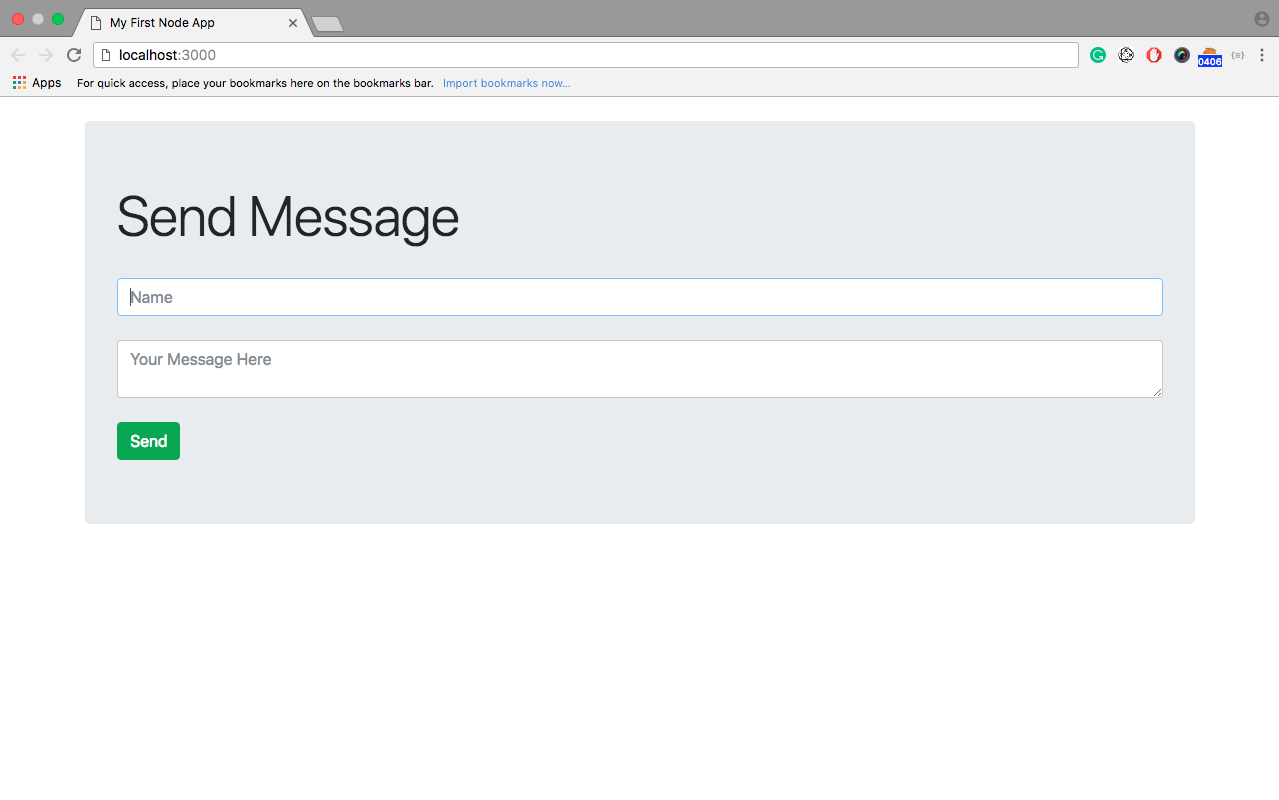
npm install -g nodemon

-g — global, so that it is accessible in all projects.

We will run the code using the command

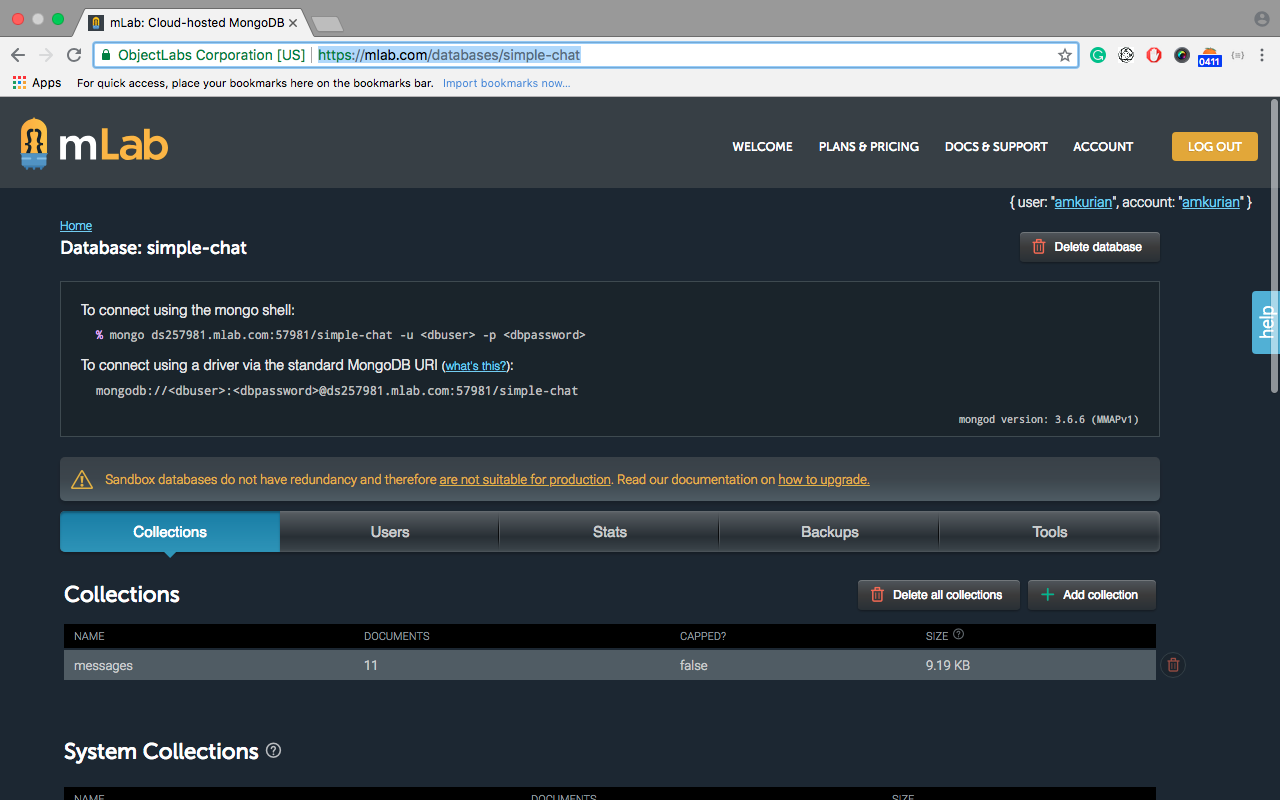
nodemon ./server.js

If you go to localhost:3000 we can see the index file:



index.html

Now that our server is up and running, we need to create our database. For this app we will having a No-SQL database and will be using **Mongodb**. I am setting up my mongodb in [**mlab.com**](https://mlab.com/). Our database will contain a single collection called **messages** with fields **name** and **message.**



In-order to connect this database to the app, we will use another package called **Mongoose**.

**Mongoose**

Mongoose is a MongoDB object modeling tool designed to work in an asynchronous environment. Mongoose can be installed using the command

npm install -s mongoose

Inside **server.js** we will require mongoose:

var mongoose = require(‘mongoose’);

And we will assign a variable, the URL of our mlab database:

var dbUrl = ‘mongodb://username:pass@ds257981.mlab.com:57981/simple-chat’

Mongoose will connect to the mlab database with the connect method:

mongoose.connect(dbUrl , (err) => {   
 console.log(‘mongodb connected’,err);  
})

And we will be defining our message model as

var Message = mongoose.model(‘Message’,{ name : String, message : String})

We can implement the chat logic now. But before that there is one more package that needs to be added.

**Body-Parser**

Body-Parser extracts the entire body portion of an incoming request stream and exposes it on req.body. The middleware was a part of Express.js earlier, but now you have to install it separately.

Install it using the following command:

npm install -s body-parser

Add the following codes to **server.js:**

var bodyParser = require(‘body-parser’)  
app.use(bodyParser.json());  
app.use(bodyParser.urlencoded({extended: false}))

**Routing**

Routing refers to how an application’s endpoints (URIs) respond to client requests. You define routing using methods of the Express app object that correspond to HTTP methods: app.get() to handle GET requests and app.post() to handle POST requests.

These routing methods [specify a callback function](https://expressjs.com/en/guide/routing.html) (sometimes called “handler functions”) called when the application receives a request to the specified route (endpoint) and HTTP method. In other words, the application “listens” for requests that match the specified routes and methods, and when it detects a match, it calls the specified callback function.

Now we need to create two routes to the messages for our chat to work.

Inside **server.js:**

**get :**will get all the message from database

app.get('/messages', (req, res) => {  
 Message.find({},(err, messages)=> {  
 res.send(messages);  
 })  
})

**post :**will post new messages created by the user to the database

app.post('/messages', (req, res) => {  
 var message = new Message(req.body);  
 message.save((err) =>{  
 if(err)  
 sendStatus(500);  
 res.sendStatus(200);  
 })  
})

In order to connect these routes to the front end we need to add the following code in the client side script tag in the **index.html:**

$(() => {  
 $("#send").click(()=>{  
 sendMessage({  
 name: $("#name").val(),   
 message:$("#message").val()});  
 })  
 getMessages()  
 })

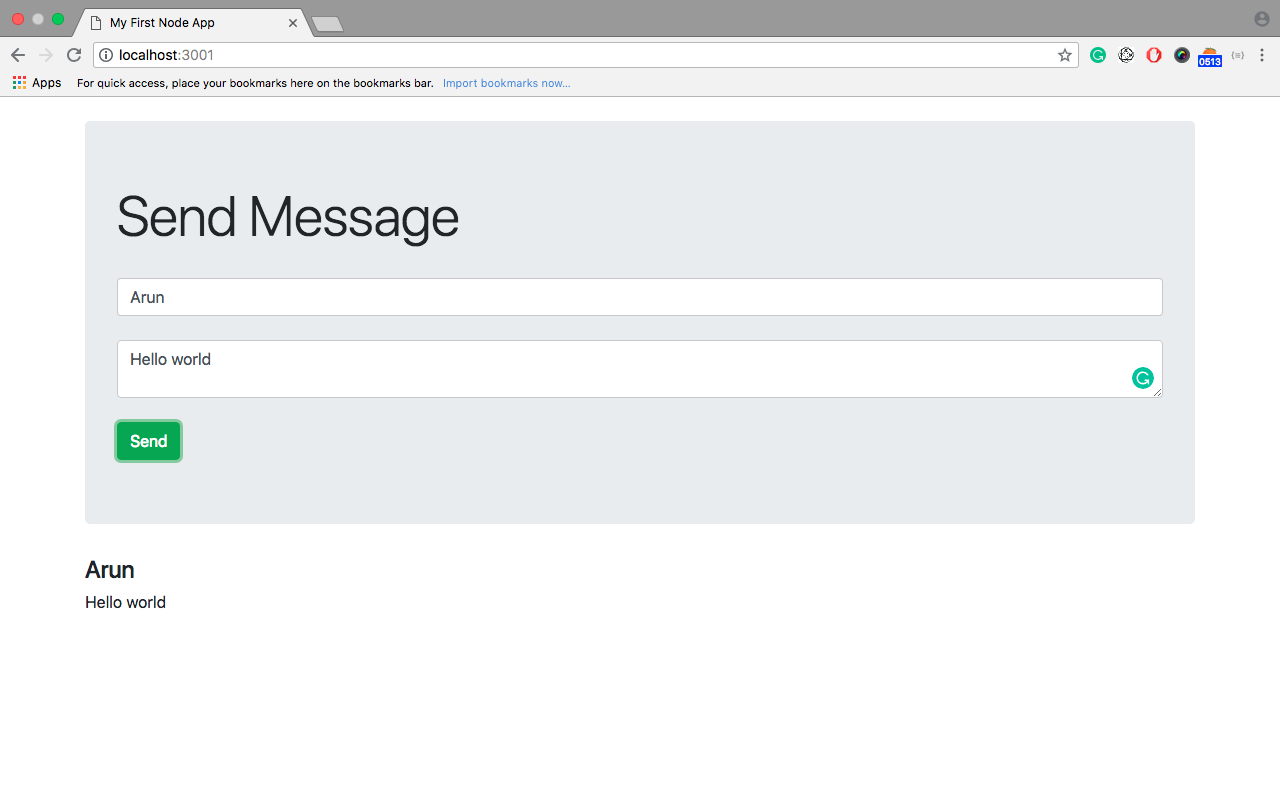
function addMessages(message){  
 $(“#messages”).append(`  
 <h4> ${message.name} </h4>  
 <p> ${message.message} </p>`)  
 }

function getMessages(){  
 $.get(‘http://localhost:3000/messages', (data) => {  
 data.forEach(addMessages);  
 })  
 }

function sendMessage(message){  
 $.post(‘http://localhost:3000/messages', message)  
 }

Here the **sendMessage** is used to invoke the post route of the messages, and save a message sent by the user. The message is created when a user clicks the send button.

Similarly the **getMessage** is used to invoke the get route of messages. This will get all the messages saved in the database and will be appended to the messages div.



The only issue now is that there is no way for the client to know if the server is updated. So each time we post a message we need to refresh the page to see the new messages.

To solve this we can add a push notification system that will send messages from server to client. In Node.js we use **socket.io.**

**Socket.io**

Socket.IO is a JavaScript library for realtime web applications. [It enables realtime, bi-directional communication between web clients and server](https://www.tutorialspoint.com/socket.io/socket.io_overview.htm). It has two parts: a client-side library that runs in the browser, and a server-side library for Node.js. Socket.io enables real-time bidirectional event-based communication.

To install socket.io:

npm install -s socket.io

we also need an HTTP package for Socket.io to work:

npm install -s http

Add the following code to **server.js:**

var http = require(‘http’).Server(app);  
var io = require(‘socket.io’)(http);

And we can create a connection:

io.on(‘connection’, () =>{  
 console.log(‘a user is connected’)  
})

In the **index.html**add the following tag:

<script src=”/socket.io/socket.io.js”></script>

Now we need to create an emit action when a message is created in **server.js**. So the post route becomes this:

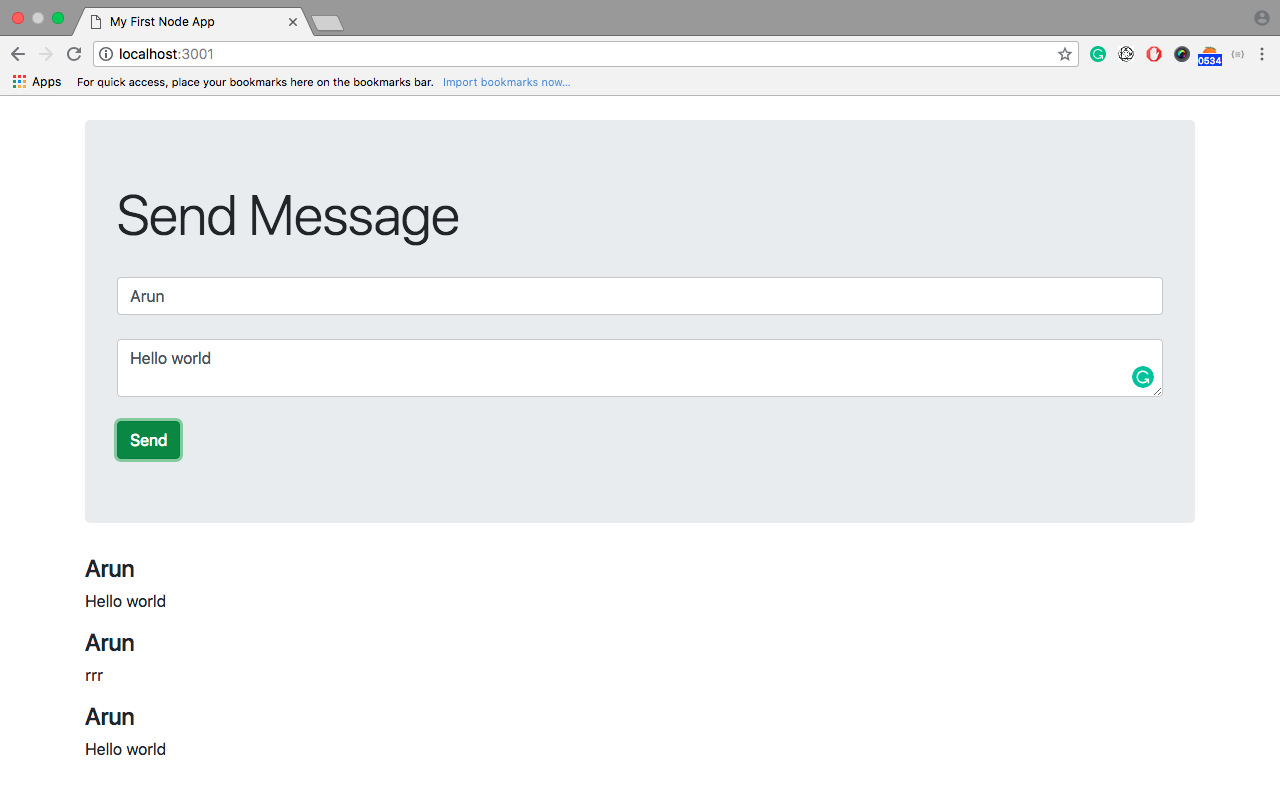
app.post('/messages', (req, res) => {  
 var message = new Message(req.body);  
 message.save((err) =>{  
 if(err)  
 sendStatus(500);  
 **io.emit('message', req.body);**  
 res.sendStatus(200);  
 })  
})

And in the client side script tag in **index.html,**add the following code:

var socket = io();

socket.on(‘message’, addMessages)

So each time a message is posted, the server will update the messages in the message div.



Great!!

This is very basic application that we can create in Node.js. There is lot of scope for improvement. The finished code can be found in <https://github.com/amkurian/simple-chat>

**server.js**

var express = require('express');  
var bodyParser = require('body-parser')  
var app = express();  
var http = require('http').Server(app);  
var io = require('socket.io')(http);  
var mongoose = require('mongoose');

app.use(express.static(\_\_dirname));  
app.use(bodyParser.json());  
app.use(bodyParser.urlencoded({extended: false}))

var Message = mongoose.model('Message',{  
 name : String,  
 message : String  
})

var dbUrl = 'mongodb://username:password[@ds257981.mlab.com](mailto:amkurian1@ds257981.mlab.com):57981/simple-chat'

app.get('/messages', (req, res) => {  
 Message.find({},(err, messages)=> {  
 res.send(messages);  
 })  
})

app.get('/messages', (req, res) => {  
 Message.find({},(err, messages)=> {  
 res.send(messages);  
 })  
})

app.post('/messages', (req, res) => {  
 var message = new Message(req.body);  
 message.save((err) =>{  
 if(err)  
 sendStatus(500);  
 io.emit('message', req.body);  
 res.sendStatus(200);  
 })  
})

io.on('connection', () =>{  
 console.log('a user is connected')  
})

mongoose.connect(dbUrl ,{useMongoClient : true} ,(err) => {  
 console.log('mongodb connected',err);  
})

var server = http.listen(3001, () => {  
 console.log('server is running on port', server.address().port);  
});

Hope this was helpful in understanding some basic concepts.

Some useful links