**Node.js Workshop 4: Web Server, Routes, Forms**

**After completing this workshop, student knows how to:**

* Install new modules using Node Package Manager (npm)
* Create a web server using Express.js
* Implement routes when serving content
* Serve both static and dynamic content

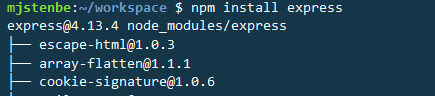
# Form Handling & Processing in Express.js

*Create a new folder called WS4 for these assignments. Place all your code there.*

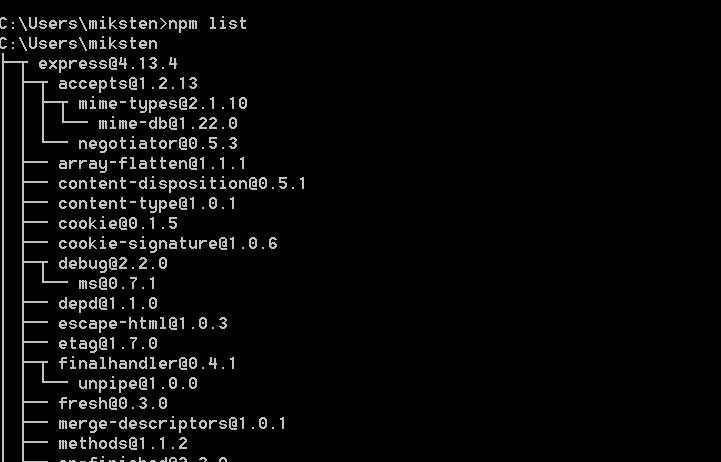
# Install new modules using Node Package Manager (npm)

Node.js comes shipped with a great tool called npm (node package manager). It is a command line tool which makes it easy to download and install additional libraries to use with Node.js. These libraries are written by developers and they contain ready-made functionality which makes coding faster and easier.

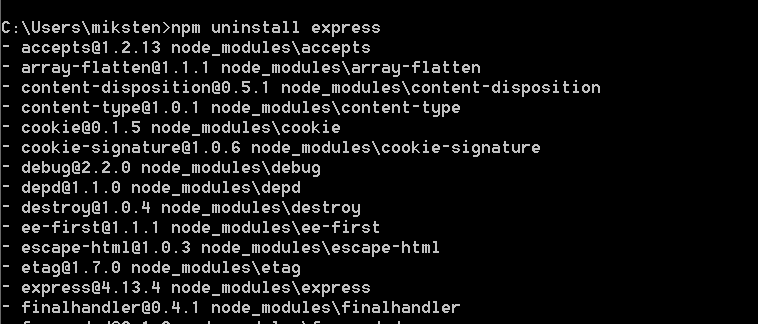
Npm is invoked using the command line terminal and typing “npm install” and then adding the name of the package at the end. For example, installing a well-known web development framework called express, would be done as described below. Try to install packages called “mysql”, “sails” and “mongodb”.



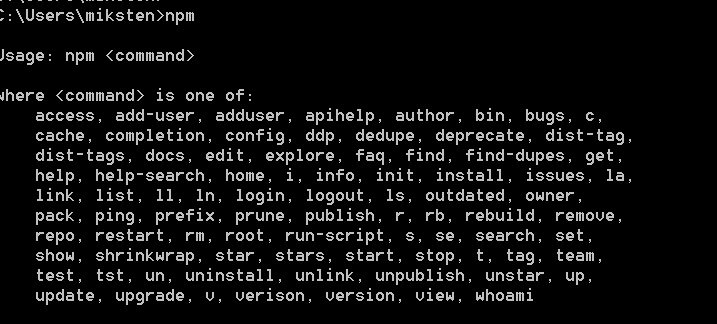
If you wish to see all the packages that are installed, type “npm list”.



Uninstalling a package can be done in a similar fashion, by typing “npm uninstall” and finish the line with the name of the package.



There are lots of switches to use with npm, you can see them all by typing plain “npm” on the console.

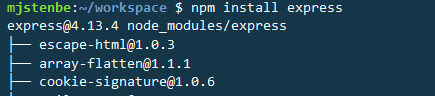


You can read more about npm and search for available packages on the website: <https://www.npmjs.com/>.

# Create a web server using Express.js

Install and take a look at Express.js framework. Express is *a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications*. Homesite and lots of information can be found at: <http://expressjs.com/>

Since Express.js is not part of Node.js core functionality, you need to install it using npm. Many other libraries are utilizing Express.js as well, so it might be that some other library already downloaded and installed Express for you as well.



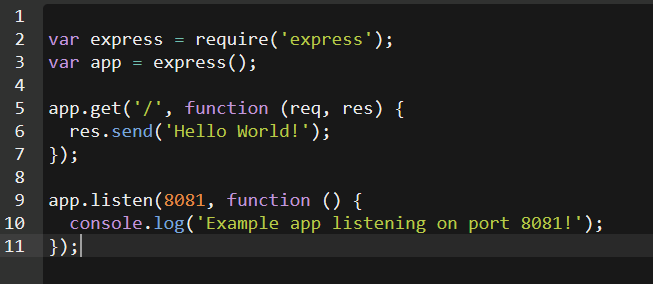
**First Express.js application**

After successful Express installation type in the following codes in new file called “Task1.js”. Note on code line #2, to be able to use the Express, require express library into the code.

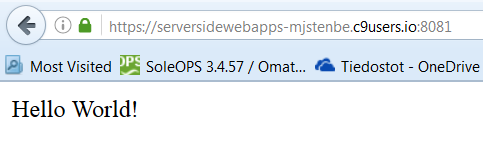
On lines 5-6 we are creating a route. Server knows how to react when a browser is sending a request to a specific url (“/”). Server is responding and sending text “Hello world” to the browser.

With “traditional programming languages and techniques”, they usually require to have a physical file from where the code is run. When using routes you can simply type in the code within the route clause in the web server code.

Finally on lines 9-10 we are engaging the web server, setting it to listen to port 8081 and logging a message to the console.



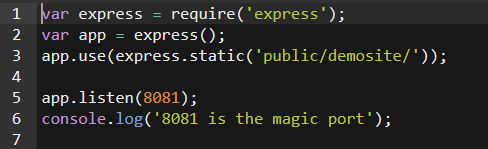
Now run the program, point your browser to the Node server and try to get the following response. Make sure you remember to include the port number after the URL, for example, Im using <https://serversidewebapps-mjstenbe.c9users.io:8081/>. If you’re developing on a local machine, you would use <http://127.0.0.1:8081/> or http://localhost:8081.



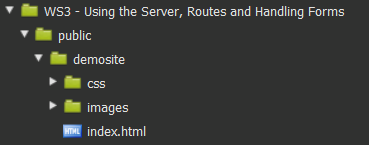
# Serving static content with routes

Previously we have learned how to serve single file to the browser. Let’s try to serve a complete directory tree to the user. This way the Node.js acts as a traditional web server serving everything from a specific directory to the browser. Note that you can still define routes within the code if needed. Now create a new file called “directoryserver.js”.

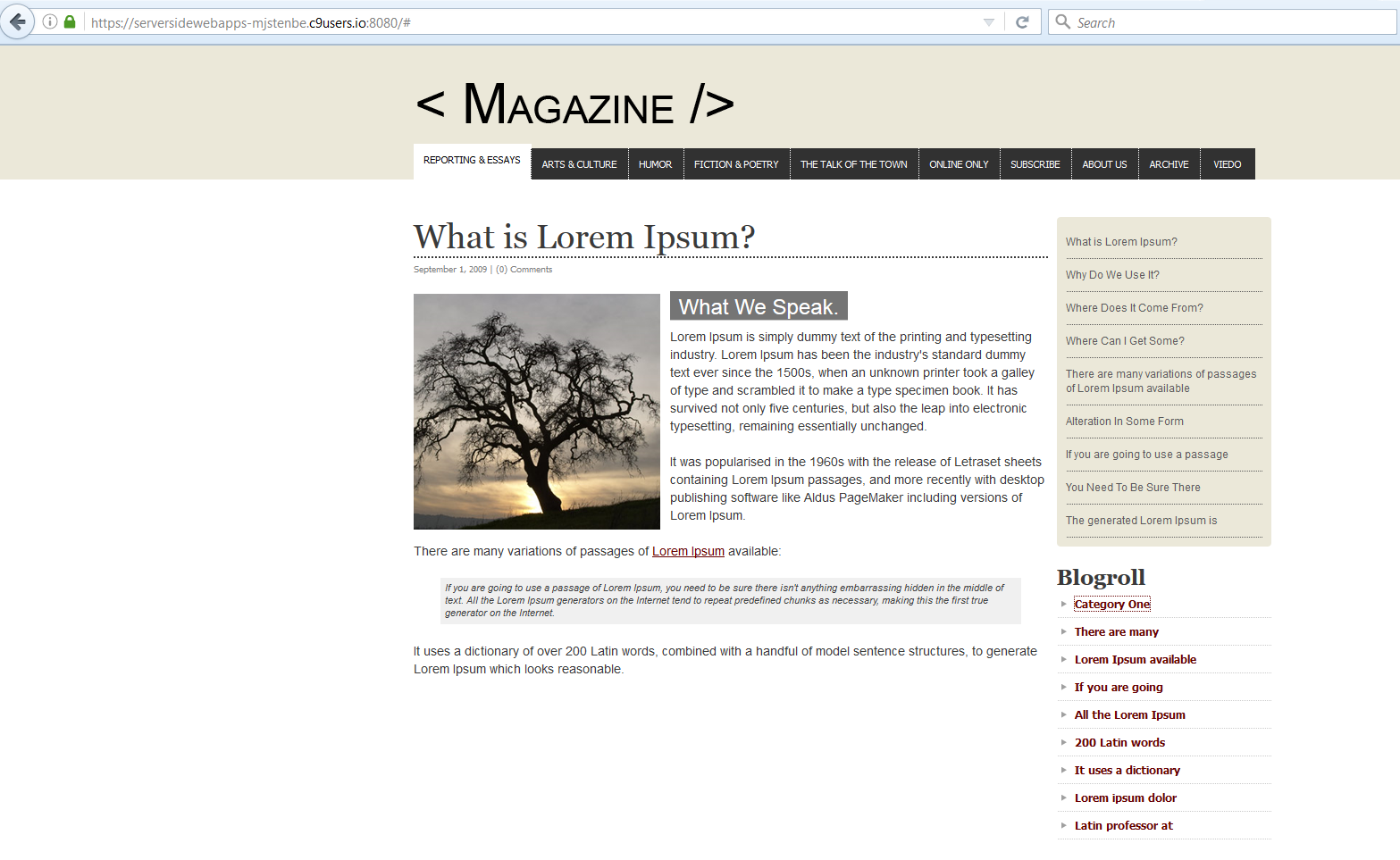
The code below would tell Node.js to serve all the static content from a subdirectory called public/demosite under the root (“/”) url of your site.



Note that the directory structure should be as below. Your application code should be place in WS3 folder.



As for the demosite, you can upload the demosite we have used in JavaScript –course as your demosite contents. Below I’m browsing to the root of my server and Node.js is displaying my demosite from public/demosite –folder.



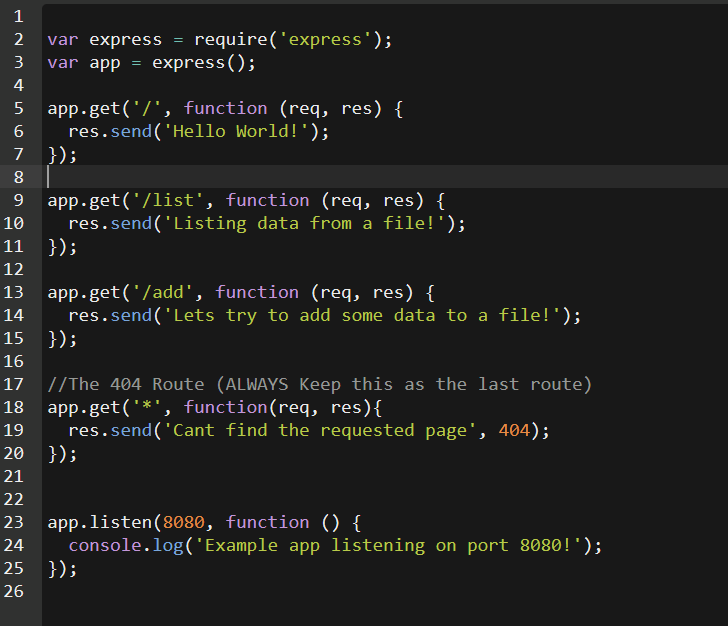
# Implement more routes with Express.js

Routes are a powerful way of creating program logic and control flow to your web app. Earlier we wrote only one route for our application, now let’s write another app which has more routes and functionality.

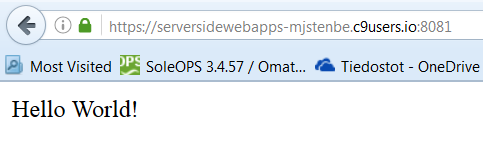
Study the code below and write it to your editor. As you see the first lines 2-3 are similar as we had before; we are just importing the express library and creating a new express.js instance as a variable called app.

Then we’re creating 4 different routes in the code. One when the browser is requesting the root page (“/”), one for “/list” and another one for “/add”.

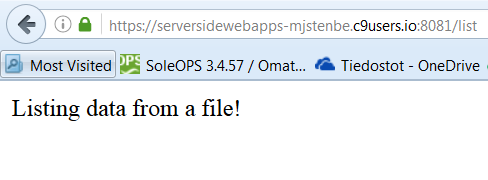
The last route on lines 18-20 is a default error page. On route “\*” we’re telling Node that if none of the earlier routes matched the incoming request, then we will use this one and sending out a simple error message with 404 error code. This route should always be the last one, because it blocks all the routes defined after is from executing.



Now run the code and try out the URL with different page requests. As you can see, by calling different routes you can control which program segments are called. Thus the routes act almost as functions in a conventional program.



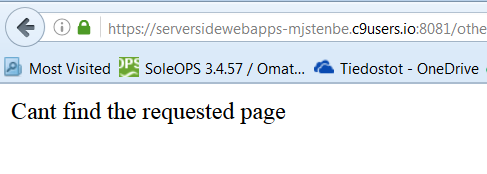
Route for: /



Route for: /add

Route for: /list





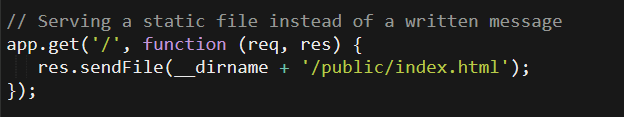
Route for: \* 

# Serving content with routes

Previously we only responded with a line of text when different routes were executed. However, now you can replace the content of the routes with whatever JavaScript functionality you can think of: filereads or writes, database requests etc. Note, that although Node.js might be serving static content on some routes, we can serve dynamic content at the other routes, such as “/list”.

Let’s work on the program by adding some functionality to **all the 4 routes** we just created.

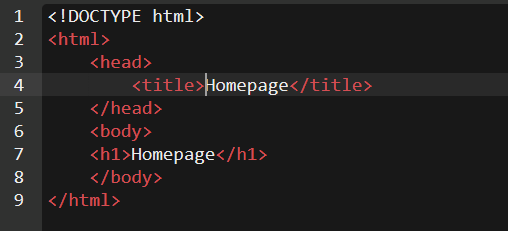
1. **The root:** Suppose we would want our program to serve a readymade HTML page when the browser requests for site root url (/). We could do this by adding the following lines to our route clause:



So instead of using res.send() –function to send plain text, we could use res.sendFile() –to send an entire file to the browser. The variable \_\_dirname refers to the current directory my project lies in. Of course you need to create the public –folder as well as the index.html file to your project.

*NOTE: If you get error saying res.sendFile() is not defined, you are using an older version of Express framework. Either try to update it or use res.sendfile(‘index.html’) instead.*

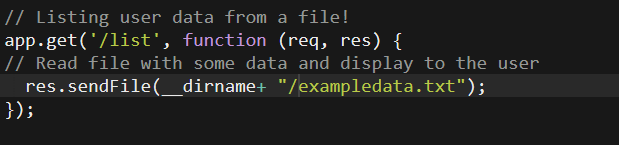
For this workshop you can use any HTML page you wish. My demo index page looks like this:



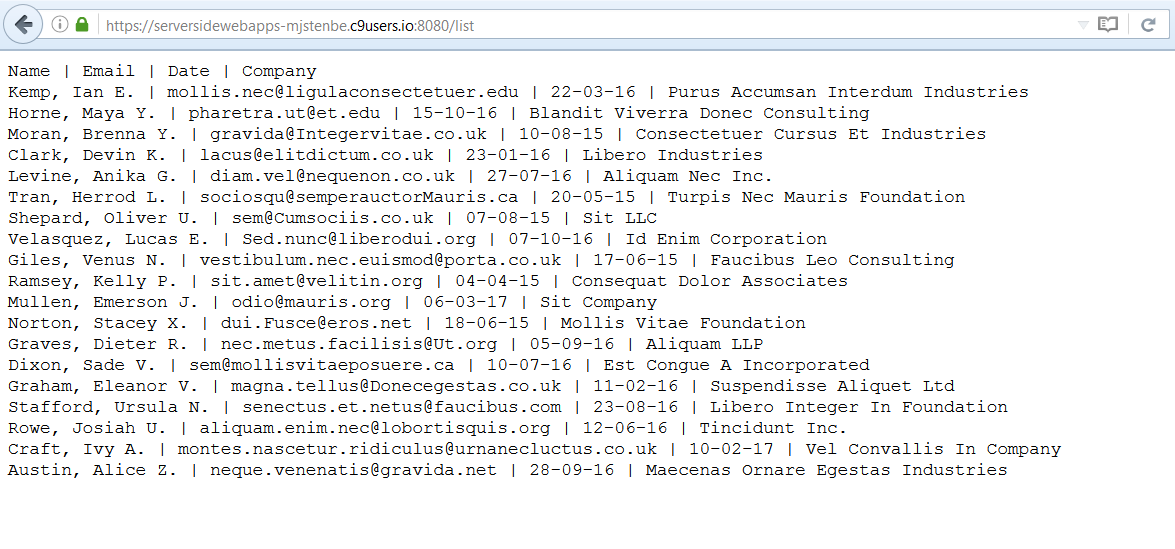
And when browsing to my URL: <https://serversidewebapps-mjstenbe.c9users.io:8080/>, I would get the following output to the browser.



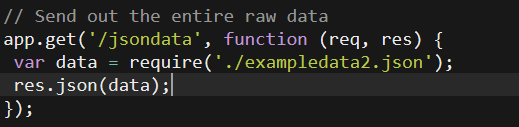
Should there be any links, images or CSS styles, they all work accordingly.

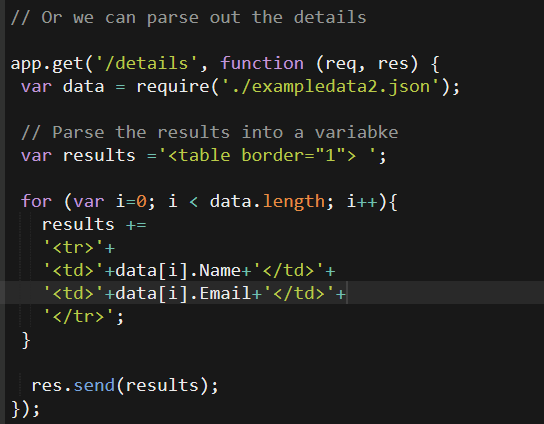
1. **The /list:** The second route we added was for /list. This time I would just send contents of a textfile to the user, again using the res.sendFile() –function. I have used a set of text data, you can find it here: <http://pastebin.com/ppJ81Y5N>.   
     
     
   

And the output to the browser would be as below:

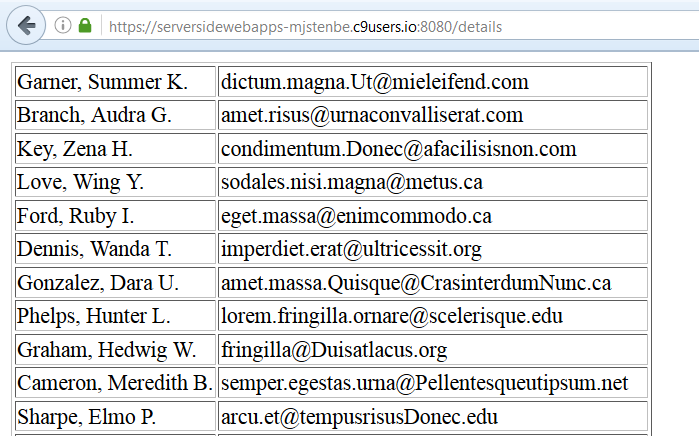


Ofcourse the data could be in JSON format as well (and it usually is when working with Node). Try another dataset in JSON from here: <http://pastebin.com/HxbRFvjv>. Below I have created a new route called /jsondata which loads the JSON file and sends it to the browser.

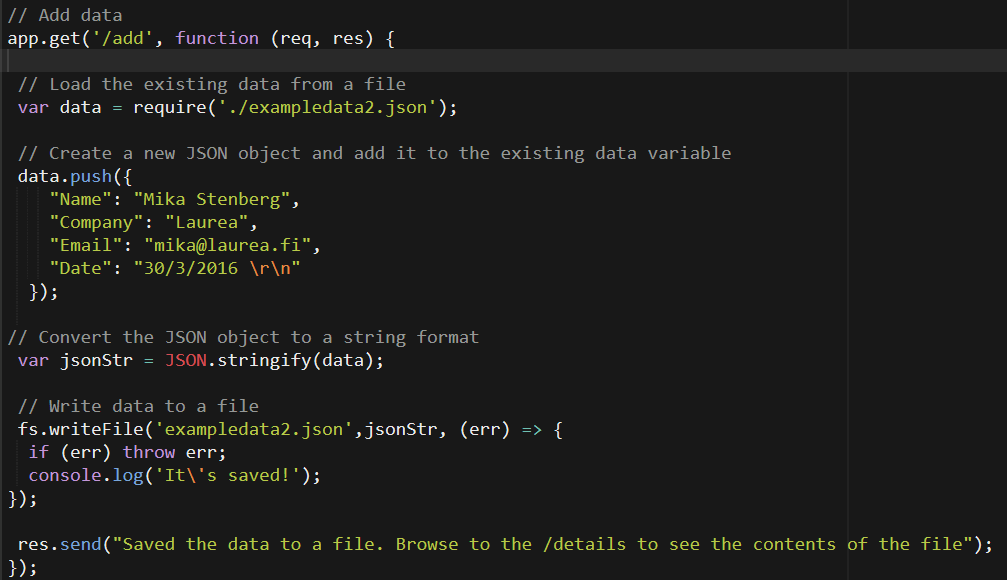


And yet, someone might appreciate if we would parse the data to the screen. Remember we already tried this on last weeks Workshops? There are many ways to approach this, but you could do something like this to parse the data from JSON object:  


The output would be something like below. **Try modifying the code above and and the remaining data fields to the output. See the sample dataset to find out what data is not yet printed.**

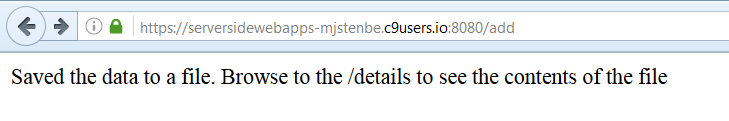


1. **The /add:** Next we will implement the /add route, which will let us add data to a JSON ojbect. Remember we did this already during last weeks session, this time we will only add the output to the browser. See the comments in the code to understand what happens.

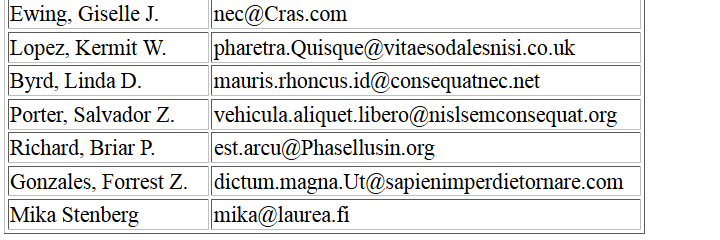


Hopefully most of the code looks familiar to you. Basically we just load the existing data, add a new JSON data on the bottom of it using push() and then saving it to the server again. This would be the easiest way to implement a simple database functionality using textfiles.

By browsing to the /add you should get the notification below.



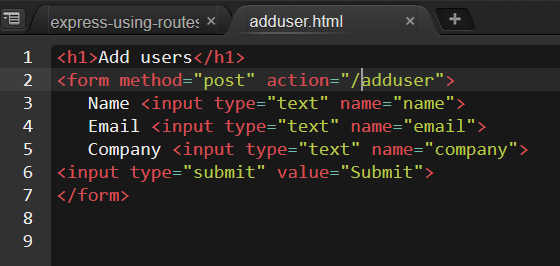
And looking the /details from the browser you can see the added data at the bottom:



Now you must be thinking; how can I specify the data to be saved. And this is what we will implement next using forms.

# Reading form data using Node.js

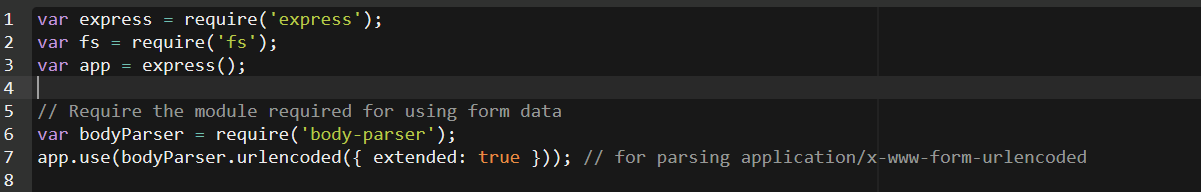
Lets first create an HTML form in a new file. Feel free to improvise. Note that the form “method” is set to POST. Also note that “action” is set to /adduser. These mean that the browser is sending the data using the POST method to the URL defined in action parameter.   
  
Here’s a simple suggestion:



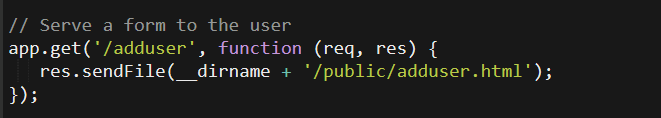
To keep things simple, lets create a new JavaScript file (formdemo.js) for this form demo. You can copy and paste code from the previous file if you want to later on.

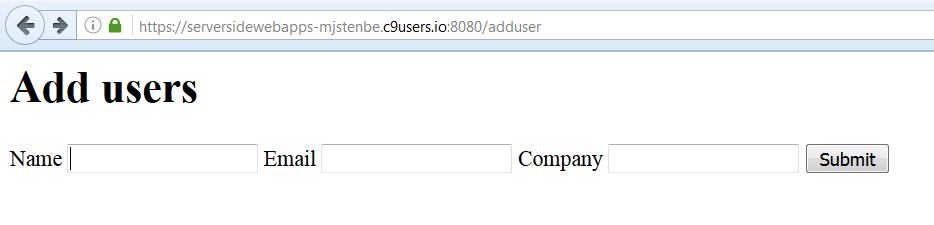
The beginning of the file contains two new lines: 6 and 7. Basically we’re just importing a module called bodyParser and applying it in our Express application. Bodyparser is needed in order to get the sent data from the forms.

*NOTE: If you get error about body-parser, you need to istall it using npm. So open a new terminal window or local Node.js terminal window and type in: “npm install body-parser”.*

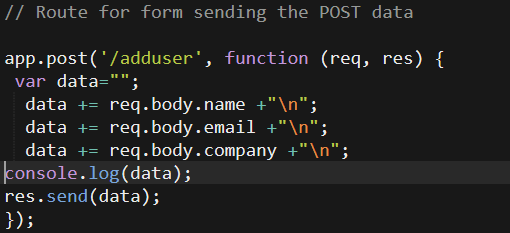


Next we’ll create a route which will show us the form:



Go ahead and try to see it in your web browser.   
  


Then we need to set up a route, which will handle the POST request sent by the form. We defined this as /adduser in the HTML of the form. NOTE that the route is defined as app.post – not app.get as before. This is because we are expecting a POST request.

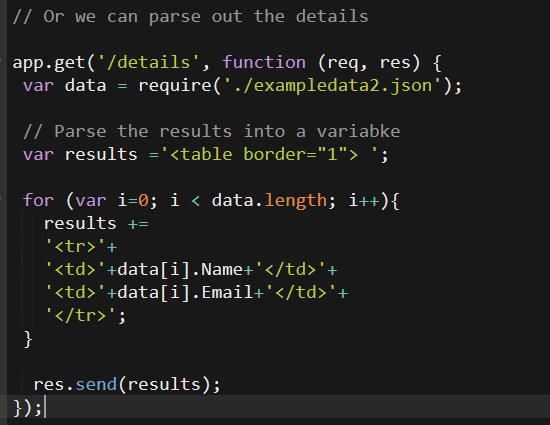
When we receive the POST request we can access the form data using the req –variable. The names of the variables are defined in the HTML of the form.  
  
So here goes:  
  


Running the file would give us:

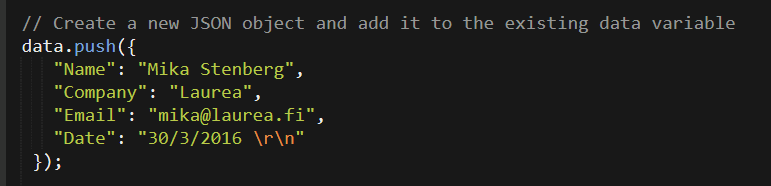


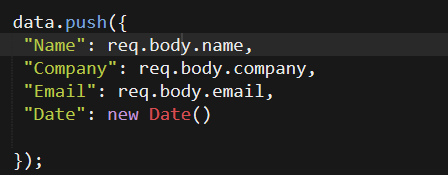
So now we know how to create a form, how to serve it and how to receive data. Next you can copy the details-route and the add-route from the previous demos. We will modify the add-route a bit and merge it with the adduser-route created.

The details-route reads the JSON data from a file and displays it on the browser window  
 (see page 7 and 8 if you forgot). This is the same as before.

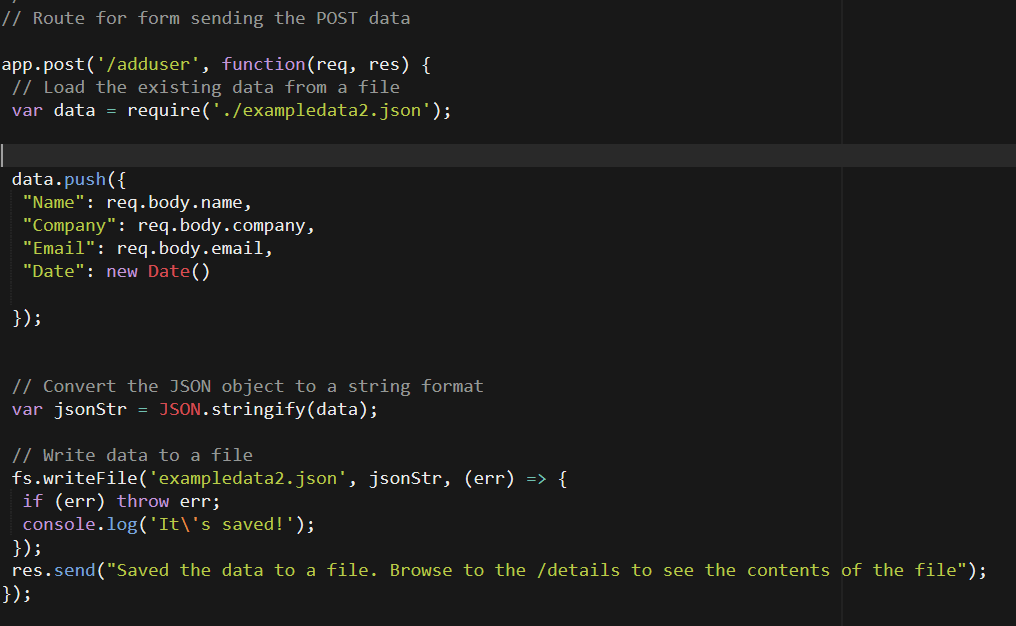


The add-route on the other hand is used to add data to that file. Earlier we only added data that was set in the code. Now we will modify the code so that we can add data from a form.

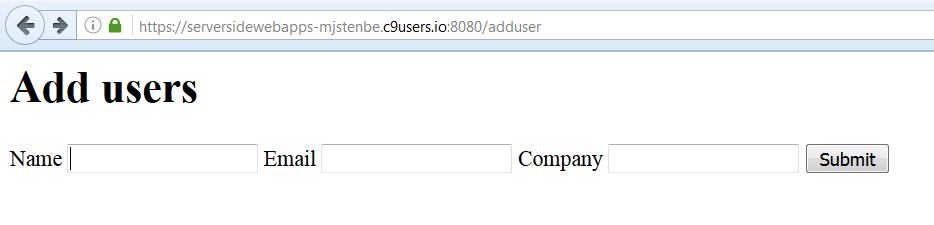


Now we will get the data from the FORM using the req variable, like below. NOTE: The structure of the data is the same, only the contents is handled different. See also, that the date is fetched using the JavaScripts new Date() object, to populate the field using current date.  
  


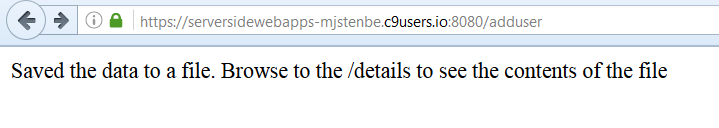
So the complete route for adding users would be (NOTE, we’re using app.post):



Now lets try it:



Clicking submit will give us:



And browsing ahead to /details gives us the extra data I just entered (at the bottom of the page):

