# Create a React FrontEnd, a Node/Express BackEnd and connect them together

[João Henrique](https://medium.com/@jrshenrique)

Sep 7, 2018

Here I’ll walk you through the process of creating a simple React app and connect it to a simple Node/Express API that we will also be creating.

I won't go into much detail about how to work with any of the technologies I will mention in this tutorial but I will leave links, in case you want to learn more.

You can find all the code in [**this repository**](https://github.com/Joao-Henrique/React_Express_App_Medium_Tutorial) I made for the tutorial.

The objective here is to give you a practical guide on how to set up and connect the **front-end client** and the**back-end API**.

Before we get our hands dirty, make sure you have [Node.js](https://nodejs.org/en/) running on your machine.

**Create the Main Project directory**

In your terminal, navigate to a directory where you would like to save your project. Now create a new directory for your project and navigate into it:

mkdir my\_awesome\_project  
cd my\_awesome\_project

**Create a**[**React**](https://reactjs.org/)**App**

This process is really straightforward.

I will be using the facebooks [create-react-app](https://github.com/facebook/create-react-app) to… you guessed, easily create a react app named **client**:

npx create-react-app client  
cd client  
npm start

*Let’s see what I have done:*

1. *Used npm’s [npx](https://medium.com/@maybekatz/introducing-npx-an-npm-package-runner-55f7d4bd282b" \t "_blank) to create a react app and named it client.*
2. *cd(change directory) into the client directory.*
3. *Started the app.*

In your browser, navigate to <http://localhost:3000/>.

If all is ok, you will see the react welcome page. Congratulations! That means you now have a basic [**React**](https://reactjs.org/) application running on your local machine. Easy right?

To stop your react app, just press `**Ctrl + c**` in your terminal.

**Create an**[**Express**](https://expressjs.com/)**App**

Ok, this will be as straightforward as the previous example. Don’t forget to navigate to your project top folder.

I will be using the [Express Application Generator](https://expressjs.com/en/starter/generator.html) to quickly create an application skeleton and name it **api:**

npx express-generator api  
cd api  
npm install  
npm start

*Let’s see what I have done:*

1. *Used npm’s npx to install express-generator globally.*
2. *Used express-generator to create an express app and named it api.*
3. *cd into the API directory.*
4. Installed all dependencies.
5. Started the app.

In your browser, navigate to <http://localhost:3000/>.

If all is ok, you will see the express welcome page. Congratulations! That means you now have a basic [**Express**](https://expressjs.com/) application running on your local machine. Easy right?

To stop your react app, just press `**Ctrl + c**` in your terminal.

**Configuring a new**[**route**](https://expressjs.com/en/guide/routing.html)**in the Express API**

Ok, let’s get our hands dirty. Time to open your favorite code editor *(I’m using*[*VS Code*](https://code.visualstudio.com/)*)*and navigate to your project folder.

*If you named the****react app as client****and the****express app as api****, you will find two main folders:****client****and****api.***

1. Inside the **API** directory, go to **bin/www** and change the port number on line 15 from 3000 to 9000. We will be running both apps at the same time later on so, doing this will avoid issues. The result should be something like this:



my\_awesome\_project/api/bin/www

2. On **api/routes**, create a **testAPI.js** file and paste this code:

var express = require(“express”);

var router = express.Router();

router.get(“/”, function(*req*, *res*, *next*) {

res.send(“API is working properly”);

});

module.exports = router;

3. On the **api/app.js** file, insert a new route on line 24:

4. Ok, you are “telling” express to use this route but, you still have to require it. Let’s do that on line 9:

var testAPIRouter = require("./routes/testAPI");

The only changes are in line 9 and line 25. It should end up something like this:



my\_awesome\_project/api/app.js

5. Congratulations! You have created a new [route](https://expressjs.com/en/guide/routing.html).

If you start your API app (in your terminal, navigate to the API directory and “**npm start”**), and go to <http://localhost:9000/testAPI> in your browser, you will see the message: ***API is working properly.***

**Connecting the React Client to the Express API**

1. On your code editor, let’s work in the **client** directory. Open **app.js** file located in **my\_awesome\_project/client/app.js**.
2. Here I will use the [F**etch API**](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch)to retrieve data from the **API.**Just paste this code after the Class declaration and before the render method:

constructor(props) {

*super*(props);

*this*.state = { apiResponse: "" };

}

callAPI() {

fetch("http://localhost:9000/testAPI")

.then(*res* => res.text())

.then(*res* => *this*.setState({ apiResponse: res }));

}

componentWillMount() {

*this*.callAPI();

1. Inside the render method, you will find a **<p>** tag. Let’s change it so that it renders the **apiResponse:**

<p *className*="App-intro">{*this*.state.apiResponse}</p>

On the end, this file should look something like this:

I know!!! This was a bit too much. Copy paste is your friend, but you have to understand what you are doing. Let’s see what I did here:

1. *On lines 6 to 9, we inserted a constructor, that initializes the default state.*
2. *On lines 11 to 16, we inserted the method****callAPI()****that will fetch the data from the API and store the response on****this.state.apiResponse.***
3. *On lines 18 to 20, we inserted a react lifecycle method called****componentDidMount(),****that will execute the****callAPI()****method after the component mounts.*
4. Last, on line 29, I used the **<p>** tag to display a paragraph on our client page, with the text that we retrieved from the API.



**What the heck!!**[**CORS**](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS)**?**

Oh yeah, baby! We are almost done. But if we start both our apps (client and API) and navigate to <http://localhost:3000/>. You still won't find the expected result displayed on the page. If you open chrome developer tools, you will find why. On the console, you will see this error:

*Failed to load*[*http://localhost:9000/testAPI*](http://localhost:9000/testAPI)*: No ‘Access-Control-Allow-Origin’ header is present on the requested resource. Origin ‘*[*http://localhost:3000'*](http://localhost:3000%27/)*is therefore not allowed access. If an opaque response serves your needs, set the request’s mode to ‘no-cors’ to fetch the resource with CORS disabled.*

This is simple to solve. We just have to add CORS to our API to allow cross-origin requests. Let’s do just that. You should [check here](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS) to find out more about [CORS](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS).

1. In your terminal navigate to the API directory and install the **CORS**package:

npm install --save cors

2. On your code editor go to the API directory and open the **my\_awesome\_project/api/app.js** file.

3. On line 6 require **CORS:**

var cors = require("cors");

4. Now on line 18 “tell” express to use **CORS**:

app.use(cors());

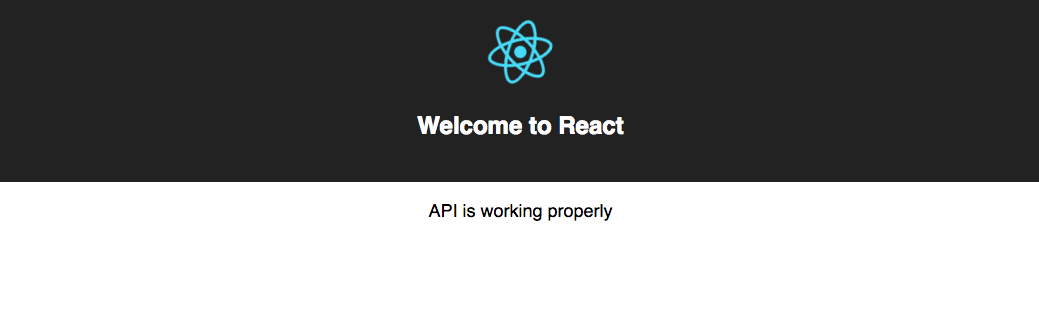
The API **app.js** file should end up something like this:



my\_awesome\_project/api/app.js

**Great Work. It’s all done!!!**

Ok!!! We are all set!

Now start both your apps (client and API), in two different terminals, using the **npm start** command.

If you navigate to <http://localhost:3000/> in your browser you should find something like this:

Of course, this project as it is won’t do much, but is the start of a Full Stack Application. You can find all the code in [**this repository**](https://github.com/Joao-Henrique/React_Express_App_Medium_Tutorial) that I’ve created for the tutorial.

Next, I will work on some complementary tutorials, like how to connect this to a MongoDB database and even, how to run it all inside Docker containers.

As a good friend of mine says:

*Be Strong and Code On!!!*

…and don't forget to be awesome today ;)

# Connecting ReactJS Frontend with NodeJs Backend (Uploading Files)

By Aman Mittal 20th Jun, 2018

Uploading files might seem like a task that needs to be conquered especially in web development. In this tutorial, we will see how to upload a simple AJAX based file using [Reactjs](https://www.zeolearn.com/react-js-training?utm_source=blog" \t "_blank) on front-end and [Node.js](https://www.zeolearn.com/node-js-training?utm_source=blog) on the back-end. This is easy to accomplish with the following technologies since the whole source code will be in one language i.e JavaScript. To show you how to combine a [Node.js](https://www.zeolearn.com/magazine/list-of-6-nodejs-modules-for-developing-networking-and-server-side-apps?utm_source=blog) backend with React Js front-end, we will be making the use of a simple file upload example. The topics we will be covering are:

* Setting up a Back-end of the app using express-generator
* Using create-react-app to scaffold a front-end Reactjs app
* Using axios for cross-origin API calls
* Handling POST requests on our server
* Using express-fileupload, a promise based library
* Lastly, connecting a Reactjs and Node.js

## Getting Started

We will be starting without back-end first. We will write a server application with necessary configurations required to accept cross-origin requests and [uploading files](https://www.zeolearn.com/magazine/uploading-files-to-aws-s3-using-nodejs?utm_source=blog). First, we need to install express-generator which is the official and quickest way to start with an Express back-end application.

npm install -g express-generator

We will install this module globally from our terminal. After installing this global npm module, we have an instance of it named express to generate our project structure.

mkdir fileupload-example

express server

cd server

When changing the current directory to the project express command just scaffolded, we can observe the following structure and files:

|  |  |
| --- | --- |
| Connecting ReactJS Frontend with NodeJs Backend | Bin  www  node\_modules  public  routes  views  app.js  package-lock.json  package.json  README.md |

To run this backend server on default configuration, we have to install the dependencies mentioned in package.json first.

npm install

npm start

Express-generator comes with the following dependencies. Some of them are essential to use such as morganand body-parser and some we can ignore for this project.

**"dependencies": {**

**"body-parser": "~1.18.2",**

**"cookie-parser": "~1.4.3",**

**"debug": "~2.6.9",**

**"express": "~4.15.5",**

**"jade": "~1.11.0",**

**"morgan": "~1.9.0",**

**"serve-favicon": "~2.4.5"**

**}**

I will be adding two more packages for our configurable back-end application to behave in the way we want to.

npm install --save cors express-fileupload

cors provide a middleware function for Express applications to enable various Cross-Origin Resource Sharing options. CORS is a mechanism that allows restricted resources (in our case, API or AJAX requests) on a web page from another domain. It helps a browser and a server to communicate and can be hosted on separate domains. You will understand it more when you will see it in action.

The other module, express-fileupload is a bare minimum express middleware function for uploading files. The advantage of it is that it supports for Promises and can handle multiple file uploads.

With these two important packages added as dependencies in our project, we can now start by modifying the default Express back-end in app.js file.

const express = require('express');

const path = require('path');

const favicon = require('serve-favicon');

const logger = require('morgan');

const cookieParser = require('cookie-parser');

const bodyParser = require('body-parser');

const cors = require('cors'); *// addition we make*

const fileUpload = require('express-fileupload'); *//addition we make*

const index = require('./routes/index');

const users = require('./routes/users');

const app = express();

*// view engine setup*

app.set('views', path.join(\_\_dirname, 'views'));

app.set('view engine', 'jade');

*// uncomment after placing your favicon in /public*

*//app.use(favicon(path.join(\_\_dirname, 'public', 'favicon.ico')));*

app.use(logger('dev'));

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({ extended: true }));

app.use(cookieParser());

*// Use CORS and File Upload modules here*

app.use(cors());

app.use(fileUpload());

app.use('/public', express.static(\_\_dirname + '/public'));

app.use('/', index);

*// catch 404 and forward to error handler*

app.use(function(*req*, *res*, *next*) {

const err = new Error('Not Found');

err.status = 404;

next(err);

});

*// error handler*

app.use(function(*err*, *req*, *res*, *next*) {

*// set locals, only providing error in development*

res.locals.message = err.message;

res.locals.error = req.app.get('env') === 'development' ? err : {};

*// render the error page*

res.status(err.status || 500);

res.render('error');

});

module.exports = app;

In the above code, you would notice that we made some additions. The first addition we did is to import packages cors and express-fileupload in app.js after other dependencies are loaded.

const cors = require('cors'); *// addition we make*

const fileUpload = require('express-fileupload'); *//addition we make*

Then just after other middleware functions, we will instantiate these two newly imported packages.

*// Use CORS and File Upload modules here*

app.use(cors());

app.use(fileUpload());

Also, we need to allow data coming from a form. For this, we have to enable urlencoded options of the body-parser module and specify a path in order to store the image file coming from the client.

app.use(bodyParser.urlencoded({ extended: true }));

*// below, also change this to*

app.use('/public', express.static(\_\_dirname + '/public'));

With this, we can see if our server is working correctly by running:

npm start

If you get the screen at left by navigation on port http://localhost:3000, it means that our server is running perfectly.

Before we move to generate our front-end application, we need to change to port for our backend. Since front-end application generated using create-react-app will also be running on port 3000. Open bin/www file and edit:

*/\*\**

*\* Get port from environment and store in Express.*

*\*/*

*// 3000 by default, we change it to 4000*

var port = normalizePort(process.env.PORT || '4000');

app.set('port', port);

## Setting up Front-end

create-react-app is another command line utility used to create a default Reactjs front-end application.

create-react-app node-react-fileupload-front-end

We will also install the required library that we are going to use for making [API](https://www.zeolearn.com/magazine/designing-a-rest-api-with-nodejs-and-mongodb-atlas?utm_source=blog) calls to our backend server.

yarn add axios

index.js is the starting point of our application in the src/ directory. It registers the render function using ReactDOM.render() by mounting App component. Components are the building blocks in any [Reactjs application](https://www.zeolearn.com/magazine/how-to-create-error-boundaries-in-reactjs-application?utm_source=blog" \t "_blank). This App component comes from src/App.js. We will be editing this file in our front-end source code.

## File Upload Form

We will be using the HTML form element that has an input. This provides access to the value, that is the file, using refs. Ref which is a special attribute that can be attached to any component in React. It takes a callback function and this callback will be executed immediately after the component is mounted. It can be also be used on an [HTML](https://www.zeolearn.com/magazine/web-components-in-html-5?utm_source=blog) element and the callback function associated will receive the DOM element as the argument. This way, the ref can be used to store a reference for that DOM element. That is exactly what we are going to do.

class App extends Component {

*// We will add this part later*

render() {

return (

<div *className*="App">

<h1>FileUpload</h1>

<form *onSubmit*={*this*.handleUploadImage}>

<div>

<input

*ref*={*ref* => {

*this*.uploadInput = ref;

}}

*type*="file"

/>

</div>

<br />

<div>

<input

*ref*={*ref* => {

*this*.fileName = ref;

}}

*type*="text"

*placeholder*="Enter the desired name of file"

/>

</div>

<br />

<div>

<button>Upload</button>

</div>

<hr />

<p>Uploaded Image:</p>

<img *src*={*this*.state.imageURL} *alt*="img" />

</form>

</div>

);

}

}

The input element must have the type="file" otherwise it would not be able to recognize what type we are using it for. It is similar to the values like email, password, etc.

The handleUploadImage method will take care of the API calls that we need to request to the server. If that call is successful, then the local state of our React application will be set to let the user know that the upload was successful. Inside this function, to make the API call, we will be using the axios library that we installed when setting up our front end app.

constructor(props) {

*super*(props);

*this*.state = {

imageURL: ''

};

*this*.handleUploadImage = *this*.handleUploadImage.bind(*this*);

}

handleUploadImage(ev) {

ev.preventDefault();

const data = new FormData();

data.append('file', *this*.uploadInput.files[0]);

data.append('filename', *this*.fileName.value);

fetch('http://localhost:4000/upload', {

method: 'POST',

body: data

}).then(*response* => {

response.json().then(*body* => {

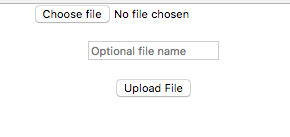
*this*.setState({ imageURL: `http://localhost:4000/${body.file}` });

});

});

}

The FormData object lets you compile a set of key/value pairs to send using XMLHttpRequest. It is primarily intended for use in sending form data but can be used independently from forms in order to transmit keyed data. To build a FormData object, instantiating it then appending fields to it by calling its append() method like we did above.



Since we are not using any styling, our form looks bare minimum. But, you can go ahead and make it look more professional. For brevity, I am going to keep things simple. I recommend you to always enter a file name, otherwise, it will store the file with a undefined.jpg name.

##### Updating the server to handle AJAX Request

Right now, we do not have in our server code to handle the POST request React app makes a request to. We will add the route in our app.js in our Express application where the default route is defined.

app.post('/upload', (*req*, *res*, *next*) => {

*// console.log(req);*

let imageFile = req.files.file;

imageFile.mv(`${\_\_dirname}/public/${req.body.filename}.jpg`, *err* => {

if (err) {

return res.status(500).send(err);

}

res.json({ file: `public/${req.body.filename}.jpg` });

console.log(res.json);

});

});

npm start

This route gets triggered when a request is made to /upload/. The callback associated using the route contain req, res objects and access to next. A standard way of defining a middleware function in an Express application. The req object has the file and the filename that was uploaded during form submission from the client application. If any error occurs, we return the 500 server error code. Otherwise, we return the path to the actual file and console the response object to check if everything is working as expected.

.mv file is promise-based and provided to us by the express-fileupload package we installed earlier. Try uploading an image file from the client now. Make sure both the client and server are running from different terminal tabs at this point. You should get a success message like this in your terminal:

POST /upload 200 98.487 ms - 25

GET /public/abc.jpg 200 6.231 ms - 60775

At the same time, the client is requesting to view the file on the front-end with a GET HTTP method. That means the route /upload from the browser is successfully called and everything is working fine. Once the file is uploaded to the server, it will be sent back to the client to reflect that the user has successfully uploaded the file.

You can find the complete code for this example at [FileUpload-Example](https://medium.com/r/?url=https%3A%2F%2Fgithub.com%2Famandeepmittal%2Ffileupload-example" \t "_blank) Github Repository.

### AmanAman Mittal, Blog Author

Aman Mittal is a Full Stack JavaScript developer specialising in Node.js. Currently he is fascinated by the aspect of React and React Native. When he is not writing JavaScript he is a bibliophile. Website:<http://www.amanhimself.me/>

# How To Combine a NodeJS Back End with a ReactJS Front End App.

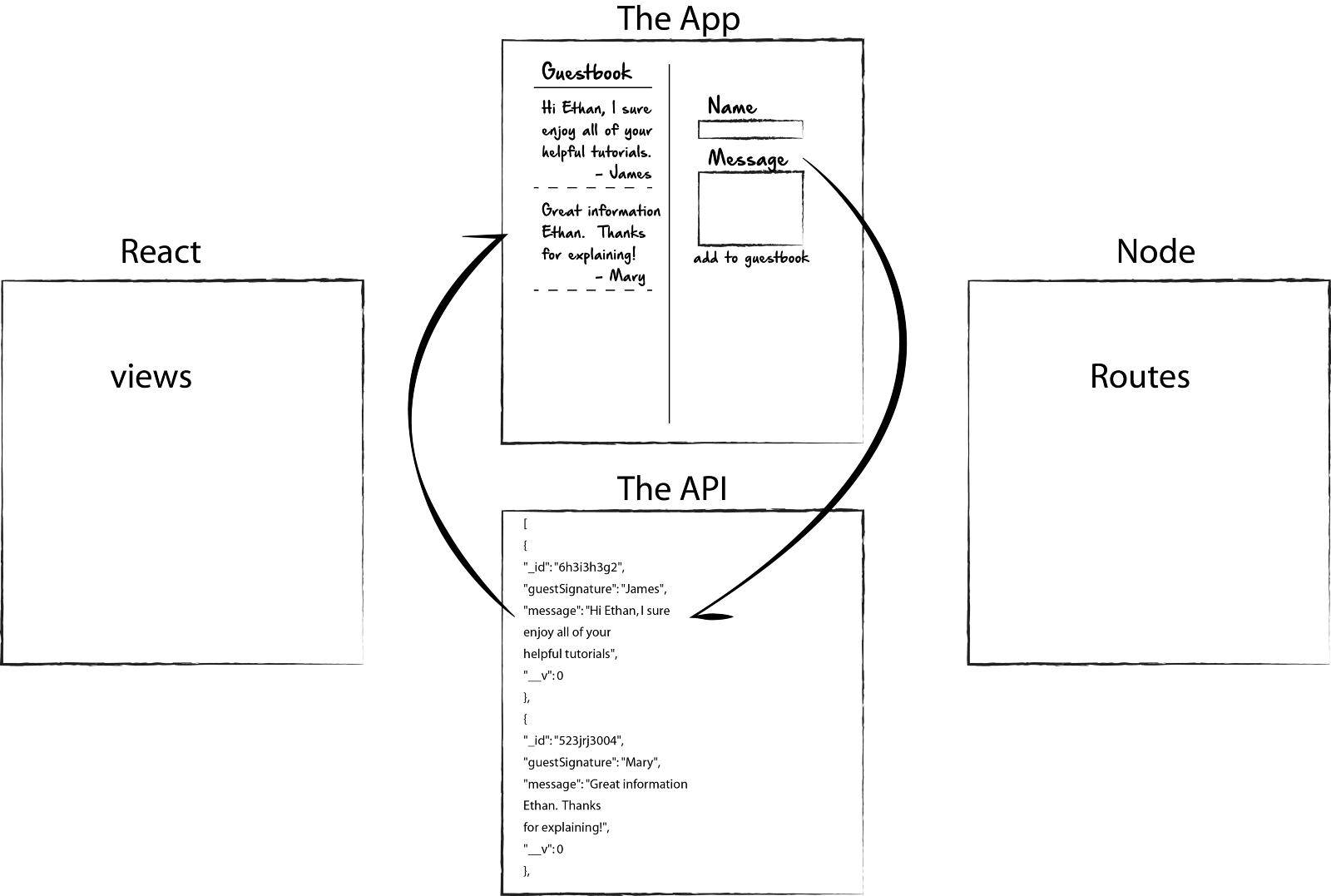
[](https://hackernoon.com/@ethan.jarrell?source=post_header_lockup)

[Ethan Jarrell](https://hackernoon.com/@ethan.jarrell)Follow

Sep 13, 2017

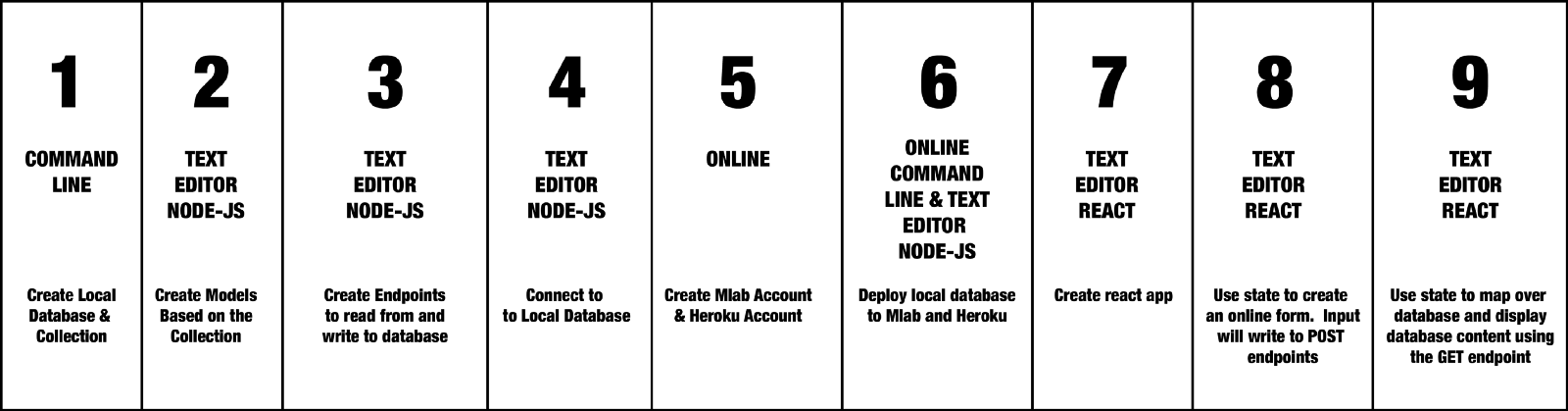
In this post, I want to show how you can build the front and back end of a website using NodeJS for the back end. We’ll use node to create endpoints, and set up a database in JSON format. Then, we’ll create a front end application using React that will post to the database, and also fetch data from it.

For this example, I’ll be making a guestbook, where users can submit their names and leave messages. Because users won’t need to log in, or retrieve any data, I won’t need to store anything in a session. Here’s the way this will likely work:



Basically, we will have this all one one page, with a form for submitting a message, alongside the actual guestbook. To make this work we’’ll start by creating the endpoints for our POST and GET requests. We’ll set up our database, deploy it, so that we can create a React app, and pull data from the online API.

Here’s the order in which I like to do things. You may prefer to work differently. This process makes sense to me, because each step builds on the step before it. During this tutorial, I’ll go through each step in more detail.



To see how the finished project could work, here’s a link to my project:

<http://ethan.jarrell.webdeveloper.surge.sh/GuestBook>

### STEP 1 -

Our database collection will contain 2 items:

1. The name of the user
2. The message the user writes

If you haven’t already done so, go ahead and install MongoDB, and get it running on port 27017. Then, on the command line, we’ll create the database.

We’ll call our database signatures. And the collection, guest\_signatures.

> Show dbs  
> Use signatures  
switched to db signatures  
> show collections  
> db.createCollection(guest\_signatures)

Now that we have that set up, we can switch to the text editor, and create the models for the database.

### STEP 2 -

Use the command line to start a new express app. Make the new directory, and then use npm init to create the app. This will automatically create your pkg.JSON file.

Since this will be a very basic app, we’ll only need 2 other files in our express app. One for our routes. I like to include all my dependencies in this file as well. Then our second fill will be for our models/schema. We’ll tackle the model file first. It should look something like this:

const mongoose = require('mongoose');

let Schema = mongoose.Schema;

const signatureSchema = new Schema({  
 guestSignature: {  
 type: mongoose.Schema.Types.Mixed,  
 required: true,  
 },  
 message: {  
 type: mongoose.Schema.Types.Mixed,  
 required: true,  
 },  
})

const Signature = mongoose.model('Signature', signatureSchema);

module.exports = Signature;

### STEP 3 -

Before we create our endpoints here, let’s go ahead an list our dependencies at the top of our file. We don’t need too many.

//====LIST DEPENDENCIES===//

const express = require('express');  
const parseurl = require('parseurl');  
const bodyParser = require('body-parser');  
const path = require('path');  
const expressValidator = require('express-validator');  
const mongoose = require('mongoose');  
const Signature = require('./models/signature.js')  
const app = express();  
const url = 'mongodb://localhost:27017/signatures';

//=========================//

You don’t necessarily need to connect with mongoose, so feel free to use MongoClient, if you prefer that. We’re also using the Signature schema that we just created in the previous step.

Now, let’s create our endpoints. We only have one model, and we’re only going to read and write to it for now, so we’re only going to have 2 endpoints.

1. Our root directory “/”, where we’ll redirect to our API
2. Our api, we’ll call “/api/signatures, where we’ll read and write to.

At our API endpoint, we’ll have a GET and a POST. We’ll reference our models, using the find() and create() methods for GET and POST respectively. And our response will need to be in JSON format, so that we can easily access the data there from React later on. Here’s how those endpoints might look:

//====ROOT DIRECTORY===//

app.get('/', function(req, res) {  
 res.json('you did it');  
});

//==========================//

//====GET ALL SIGNATURES===//

app.get('/api/signatures', function(req, res) {  
 Signature.find({}).then(eachOne => {  
 res.json(eachOne);  
 })  
 })

//==========================//

//====POST NEW SIGNATURE===//

app.post('/api/signatures', function(req, res) {  
 Signature.create({  
 guestSignature: req.body.SignatureOfGuest,  
 message: req.body.MessageofGuest,  
 }).then(signature => {  
 res.json(signature)  
 });  
});

//==========================//

The only thing I didn’t really mention was the req.body.SignatureOfGuest and req.body.MessageofGuest. Everything else in this section refers to something we’ve already created, like our database, collections and models. This refers to the name of the field that we’ll use in our React App, Our input will have to use the names SignatureOfGuest and MessageofGuest, so we’ll need to remember that.

### STEP 4 -

Now, we’ll want to connect to our local database from out text editor. If you remember, we had this constant in our list:

const url = 'mongodb://localhost:27017/signatures';

We’ll write a function using this constant to connect to our local database.

//====MONGOOSE CONNECT===//

mongoose.connect(url, function (err, db) {  
 if (err) {  
 console.log('Unable to connect to the mongoDB server. Error:', err);  
 } else {  
 console.log('Connection established to', url);  
 }  
});

//==========================//

In order to make sure everything is working properly, we’ll also need to add an app.listen at the end of our file.

At this point, it’s a good idea to pause, and make sure the endpoints and local connection works. You can do that by using Postman to make GET and POST requests. If the endpoints or connections don’t work, you should be able to diagnose the problem here based on the error messages you receive. If everything is working, it should allow you to appropriately read and write to the database.

Once everything is working, it’s time to set up our account on Heroku and Mlab.

### STEP 5-

Sign up for a Heroku account [here](https://signup.heroku.com/). And create an mLab account [here](https://mlab.com/signup/).

### STEP 6 -

1. After creating your mLab account, click on the Create New button and select a Single node sandbox. There are paid options as well, but the sandbox has plenty of space for getting started. Give your database a name. Since my local database is signatures, I’m using the same name for my mLab database.
2. Now that you have your database created, you can start a new collection from here.
3. You will also need to add a User or Users who can access your database. Without doing this step, and adding yourself as a User with access, authentication will always fail when you try to deploy it.

Now your database is running on mLab. When you click on the database, you should see some information at the top, telling you how to integrate the connection. It should look something like this:

mongodb://<dbuser>:<dbpassword>[@ds7](http://twitter.com/ds129024)9234.mlab.com:9234/signatures

The actual url of the database will just have the username and password replaced with your username and password.

Now let’s head back to our text editor. We currently have the database running locally at this location:

const url = 'mongodb://localhost:27017/signatures';

To change it, so that we are connected to mLab, simply update the url variable with the information from mLab. It should look something like this:

const url = 'mongodb://username:password@ds79234.mlab.com:9234/signatures';

However, you’ll likely be putting this on github, or another public place. You don’t want your mlab username and password in a public sphere where anyone can see it. To fix that, we’ll set an environment variable on the command line, and then update our url variable one more time in the text editor. On the command line, use this command:

export MONGOLAB\_URI="mongodb://username:password@ds79234.mlab.com:9234/signatures';

Of course, replacing it with your own username, password, numbers and database name. Now, back in our text editor, we will change our url variable to the following:

const url = process.env.MONGOLAB\_URI;

We’ll also want to change our app.listen to reflect the new port. It could look like this:

app.listen(process.env.PORT || 3000);  
console.log('starting applicaiton. Good job!');

This way, it will try to run from mongolab, but if it can’t make the connection, it will still listen on port 3000 by default. This will also allow you to run the app locally or from mlab, in case you want to test changes on the local version.

The final step in making the connection is deploying your code to your Heroku App. to do this, you can use the following code from the command line:

heroku config:set MONGOLAB\_URI=mongodb://username:password@ds79234.mlab.com:9234/signatures

Your app should be successfully deployed on heroku, and you can open it from there now. If you are getting errors, double check to make sure it’s running locally. If so, then it’s probably an error along the way in the connection.

In these last two steps, there’s quite a bit of repetitive code. I’m going to outline the high level code here.

### STEP 7 -

Use ‘create-react-app’ from the command line to create a new react app.

### STEP 8 -

Now, we’ll want to create a form, to allow for user input. Here’s how we do that.

1. In a new component, we’ll create a new class.

class GuestBook extends Component

2. We’ll use a constructor and super method to pass props down.

constructor(props) {  
 super(props);

3. Use the this keyword to handle the name and message of guest, and bind it to (this).

this.handleSignatureOfGuest = this.handleSignatureOfGuest.bind(this);  
 this.handleMessageofGuest = this.handleMessageofGuest.bind(this);

4. Set the state of the name and message of guest to an empty string.

this.state = {  
 SignatureOfGuest: "",  
 MessageofGuest: "",  
 };

5. Listen for an event on the state of both the name and message input.

handleSignatureOfGuest(event) {  
 this.setState({ SignatureOfGuest: event.target.value });  
 }  
 handleMessageofGuest(event) {  
 this.setState({ MessageofGuest: event.target.value });  
 }

6. Create a function that changes the name and message to the value of the target input.

addToGuestBook = event => {  
 event.preventDefault();  
 this.setState({  
 SignatureOfGuest: event.target.value,  
 MessageofGuest: event.target.value,  
});

7. I’m now using axios to post the input data to our database, which is on heroku.

axios.post('<[your-heroku-url here>'](https://ancient-sea-87841.herokuapp.com/api/signatures%27), {  
 SignatureOfGuest: this.state.SignatureOfGuest,  
 MessageofGuest: this.state.MessageofGuest,  
 })  
 .then(response => {  
 console.log(response, 'Signature added!');  
 })  
 .catch(err => {  
 console.log(err, 'Signature not added, try again');  
 });

8. Then I’m resetting the state of the input to an empty string.

this.setState({  
 SignatureOfGuest: "",  
 MessageofGuest: "",  
 });  
};

9. Finally, we’ll make a render method, and return our page with the input fields. Inside the input of each field, we’ll give it an onChange, name, and value. the onChange will be set to the.handlemessage or this.handlename, for each field. The name of each field will be set to what we called it in our node app. The value will use state, and we’ll set that to this.state.message, and this.state.name.

<input  
 onChange={this.handleSignatureOfGuest}  
 name="SignatureOfGuest"  
 className="NameinputForm"  
 value={this.state.SignatureOfGuest}  
 placeholder="Enter your name"  
 />  
 <textarea  
 onChange={this.handleMessageofGuest}  
 name="MessageofGuest"  
 className="MessageinputForm"  
 value={this.state.MessageofGuest}  
 placeholder="Type a message"  
 />

10. Then we’ll add a submit button, where we call the function from earlier.

<button  
 className="submitbuttonguestbook"  
 type="submit"  
 onClick={this.addToGuestBook}  
 >  
 Submit to Guestbook<i className="GuestBookButton2" aria-hidden="true" />  
 </button>

### STEP 9-

Now we’ll make another component where we’ll render the data that’s being stored in our database. Then we can export that component, and put it on our guestbook page.

Inside this component we’ll do the following:

1. In a new component, we’ll create a new class.

class GuestNames extends Component {

2. We’ll use a constructor and super method to pass props down.

constructor(props) {  
 super(props);

3. Use this.state to set the state of our guestbook messages to an empty string.

this.state = {  
 messages: "",  
 };

4. Use a componentDidMount lifecycle method

componentDidMount() {

5. Inside that method we’ll use fetch and the url of our heroku api to fetch the information from the database.

fetch('<your-heroku-url-goes-here>['](https://ancient-sea-87841.herokuapp.com/api/signatures%27))  
 .then(results => {  
 return results.json();

6. We’ll map over the data, and return the data we want.

data.map((msg) => {  
 return(  
 <div key={msg.results}>  
 <h3 className="h3msg">{msg.message}</h3>

<h2 className="h2sig">-{msg.guestSignature}</h2>  
 </div>

7. Now we’ll use this.setState to set the state of the messages to the new state using the data we just fetched.

this.setState({messages: messages});

8. Now we’ll create render mehod.

render() {

9. Inside the render method, we’ll create JSX elements to render our data inside the component. I’m using this.state.messages inside an <h6> tag.

return (

<div className="guestdataContainer">  
 <h6>Guestbook Messages</h6>  
 {this.state.messages}  
 </div>

10. Finally, we’ll export the component, so we can use it on other pages.

export default GuestNames;

Again, if you have any questions feel free to reach out. Thanks!

# How to get create-react-app to work with a Node.js back-end API

[Go to the profile of Esau Silva](https://medium.freecodecamp.org/@_esausilva?source=post_header_lockup)

[Esau Silva](https://medium.freecodecamp.org/@_esausilva)Follow

Dec 20, 2017

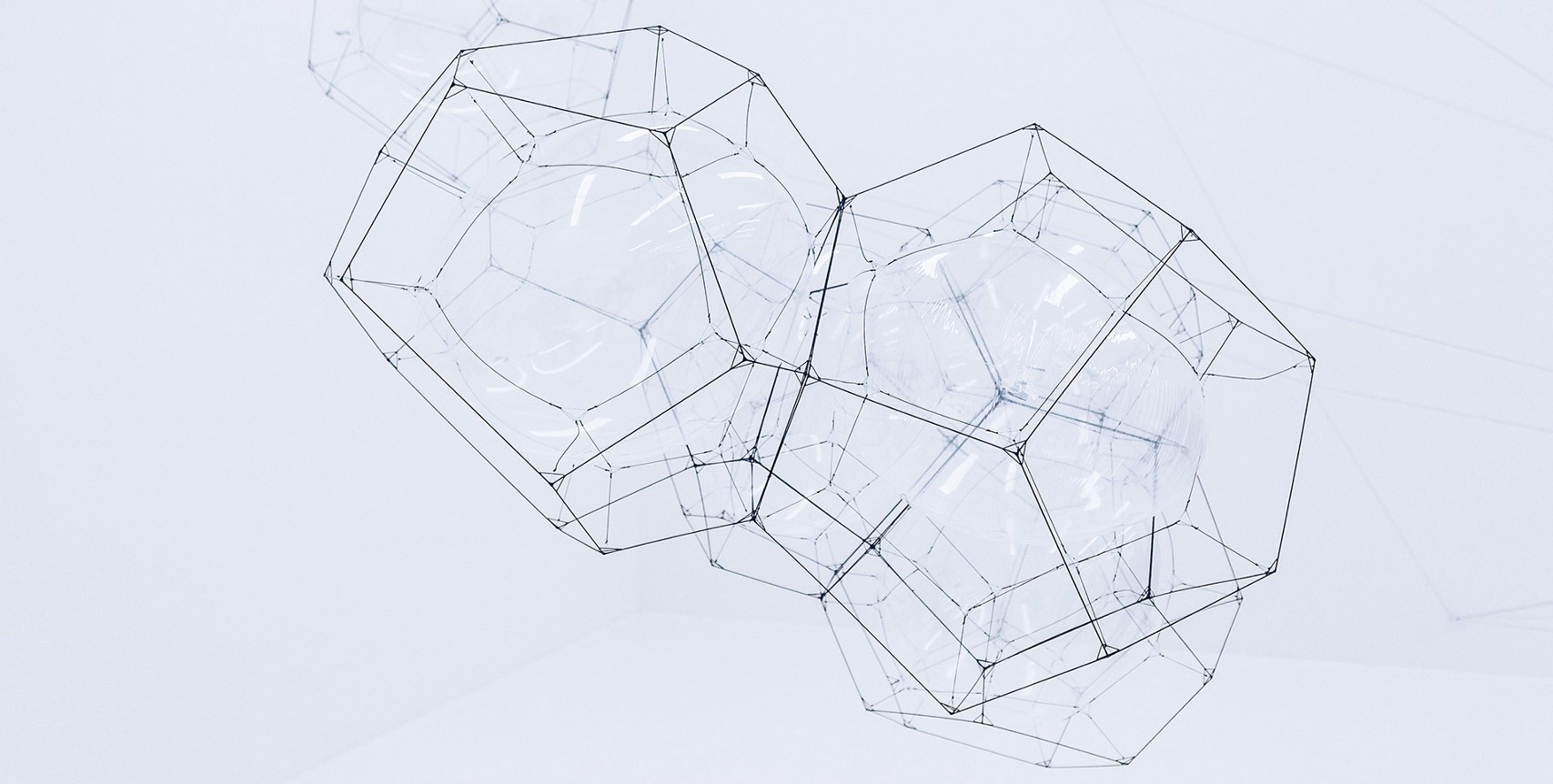


Photo by [Sebastien Gabriel](https://unsplash.com/photos/5rAcUaCtMzk?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText) on [Unsplash](https://unsplash.com/?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText" \t "_blank)

This is a very common question among newer React developers, and one question I had when I was starting out with React and Node.js. In this short example I will show you how to make create-react-app work with Node.js and Express Back-end.

#### create-react-app

Create a project using create-react-app.

npx create-react-app example-create-react-app-express

Create a /client directory under example-create-react-app-expressdirectory and move all the React boilerplate code created by create-react-app to this new client directory.

cd example-create-react-app-express  
mkdir client

#### The Node Express Server

Create a package.json file inside the root directory (example-create-react-app-express) and copy the following contents:

{  
 "name": "example-create-react-app-express",  
 "version": "1.0.0",  
 "scripts": {  
 "client": "cd client && yarn start",  
 "server": "nodemon server.js",  
 "dev": "concurrently --kill-others-on-fail \"yarn server\" \"yarn client\""  
 },  
 "dependencies": {  
 "body-parser": "^1.18.3",  
 "express": "^4.16.4"  
 },  
 "devDependencies": {  
 "concurrently": "^4.0.1"  
 }  
}

Notice I am using concurrently to run the React app and Server at the same time. The –kill-others-on-fail flag will kill other processes if one exits with a non zero status code.

Install nodemon globally and the server dependencies:

npm i nodemon -g  
yarn

Create a server.js file and copy the following contents:

const express = require('express');  
const bodyParser = require('body-parser');

const app = express();  
const port = process.env.PORT || 5000;

app.use(bodyParser.json());  
app.use(bodyParser.urlencoded({ extended: true }));  
  
app.get('/api/hello', (req, res) => {  
 res.send({ express: 'Hello From Express' });  
});

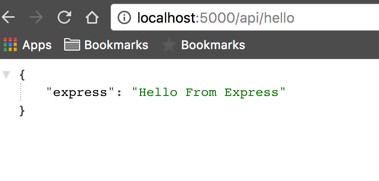
app.post('/api/world', (req, res) => {  
 console.log(req.body);  
 res.send(  
 `I received your POST request. This is what you sent me: ${req.body.post}`,  
 );  
});  
  
app.listen(port, () => console.log(`Listening on port ${port}`));

This is a simple Express server that will run on port 5000 and have two API routes: GET - /api/hello, and POST -/api/world.

At this point you can run the Express server with the following command (still inside the root directory):

node server.js

Now navigate to <http://localhost:5000/api/hello>, and you will get the following:



We will test the POST route once we build the React app.

#### The React App

Now switch over to the client directory where our React app lives.

Add the following line to the package.json file created by create-react-app.

"proxy": "http://localhost:5000/"

The key to using an Express back-end server with a project created withcreate-react-app is to use a proxy. This tells the Web-pack development server to proxy our API requests to our API server, given that our Express server is running on localhost:5000.

Now modify ./client/src/App.js to call our Express API Back-end, changes are in bold.

import React, { Component } from 'react';

import logo from './logo.svg';

import './App.css';

class App extends Component {  
 **state = {  
 response: '',  
 post: '',  
 responseToPost: '',  
 };**

**componentDidMount() {  
 this.callApi()  
 .then(res => this.setState({ response: res.express }))  
 .catch(err => console.log(err));  
 }**

**callApi = async () => {  
 const response = await fetch('/api/hello');  
 const body = await response.json();**

**if (response.status !== 200) throw Error(body.message);**

**return body;  
 };**

**handleSubmit = async e => {  
 e.preventDefault();  
 const response = await fetch('/api/world', {  
 method: 'POST',  
 headers: {  
 'Content-Type': 'application/json',  
 },  
 body: JSON.stringify({ post: this.state.post }),  
 });  
 const body = await response.text();**

**this.setState({ responseToPost: body });  
 };**

render() {  
 return (  
 <div className="App">  
 <header className="App-header">  
 <img src={logo} className="App-logo" alt="logo" />  
 <p>  
 Edit <code>src/App.js</code> and save to reload.  
 </p>  
 <a  
 className="App-link"  
 href="[https://reactjs.org](https://reactjs.org/)"  
 target="\_blank"  
 rel="noopener noreferrer"  
 >  
 Learn React  
 </a>  
 </header>  
 **<p>{this.state.response}</p>  
 <form onSubmit={this.handleSubmit}>  
 <p>  
 <strong>Post to Server:</strong>  
 </p>  
 <input  
 type="text"  
 value={this.state.post}  
 onChange={e => this.setState({ post: e.target.value })}  
 />  
 <button type="submit">Submit</button>  
 </form>  
 <p>{this.state.responseToPost}</p>**  
 </div>  
 );  
 }  
}

export default App;

We create callApi method to interact with our GET Express API route, then we call this method in componentDidMount and finally set the state to the API response, which will be Hello From Express.

Notice we didn’t use a fully qualified URL <http://localhost:5000/api/hello>to call our API, even though our React app runs on a different port (3000). This is because of the **proxy**line we added to the package.json file earlier.

We have a form with a single input. When submitted calls handleSubmit, which in turn calls our POST Express API route then saves the response to state and displays a message to the user: I received your POST request. This is what you sent me: [message from input].

Now open ./client/src/App.css and modify .App-header class as follows (changes in bold)

.App-header {  
**...**  
 **min-height: 50%;**  
**...**  
 **padding-bottom: 10px;**  
}

### Running the App

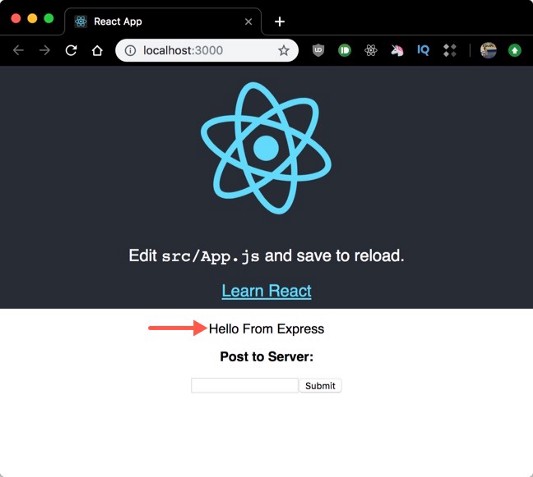
If you still have the server running, go ahead and stop it by pressing Ctrl+C in your terminal.

From the project root directory run the following:

yarn dev

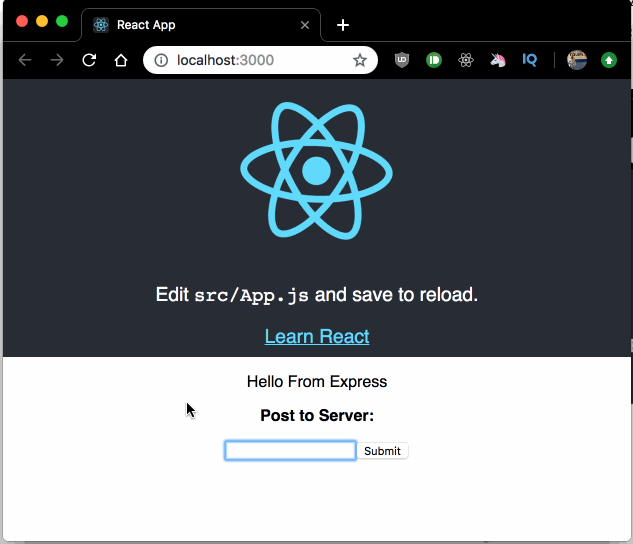
This will launch the React app and run the server at the same time.

Now navigate to [http://localhost:3000](http://localhost:3000/) and you will hit the React app displaying the message coming from our GET Express route. Nice 🎉!



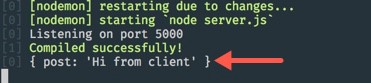
Displaying GET route

Now, type something in the input field and submit the form, you will see the response from the POST Express route displayed right below the input field.



Calling POST route

Finally take a look at at your terminal, you will see the message we sent from the client, that is because we call console.log on the request body in the POST Express route.



Node

### Production Deployment to Heroku

Open server.js and replace with the following contents:

const express = require('express');  
const bodyParser = require('body-parser');  
const path = require('path');

const app = express();  
const port = process.env.PORT || 5000;

app.use(bodyParser.json());  
app.use(bodyParser.urlencoded({ extended: true }));

// API calls  
app.get('/api/hello', (req, res) => {  
 res.send({ express: 'Hello From Express' });  
});

app.post('/api/world', (req, res) => {  
 console.log(req.body);  
 res.send(  
 `I received your POST request. This is what you sent me: ${req.body.post}`,  
 );  
});

if (process.env.NODE\_ENV === 'production') {  
 // Serve any static files  
 app.use(express.static(path.join(\_\_dirname, 'client/build')));

// Handle React routing, return all requests to React app  
 app.get('\*', function(req, res) {  
 res.sendFile(path.join(\_\_dirname, 'client/build', 'index.html'));  
 });  
}

app.listen(port, () => console.log(`Listening on port ${port}`));

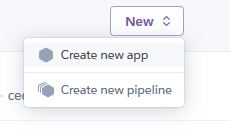
Open ./package.json and add the following to the scripts entry

"start": "node server.js",  
"heroku-postbuild": "cd client && npm install && npm install --only=dev --no-shrinkwrap && npm run build"

Heroku will run the start script by default and this will serve our app. Then we want to instruct Heroku to build our client app, we do so with heroku-postbuild script.

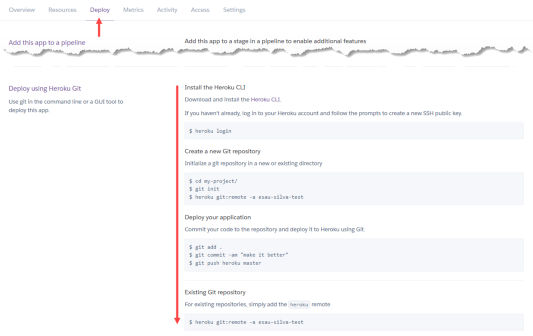
Now, head over to [Heroku](https://www.heroku.com/) and log in (or open an account if you don’t have one).

Create a new app and give it a name



Create new app on Heroku

Click on the **Deploy** tab and follow the deploy instructions (which I think they are pretty self-explanatory, no point on replicating them here 😁)



Deploy an app to Heroku

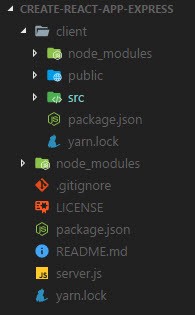
And that is it, you can open your app by clicking on the **Open app** button at the top right corner within the Heroku dashboard for your app.

Visit the deployed app for this tutorial: <https://cra-express.herokuapp.com/>

I write about other deployments options here:

* [Netlify](https://blog.bitsrc.io/react-production-deployment-part-1-netlify-703686631dd1)
* [Now](https://blog.bitsrc.io/react-production-deployment-part-2-now-c81657c700b7)

### Project Structure

This will be the final project structure.

Get the full code on the [GitHub repository](https://github.com/esausilva/example-create-react-app-express).

Thank you for reading and I hope you enjoyed it. Any question, suggestions let me know in the comments below!