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());

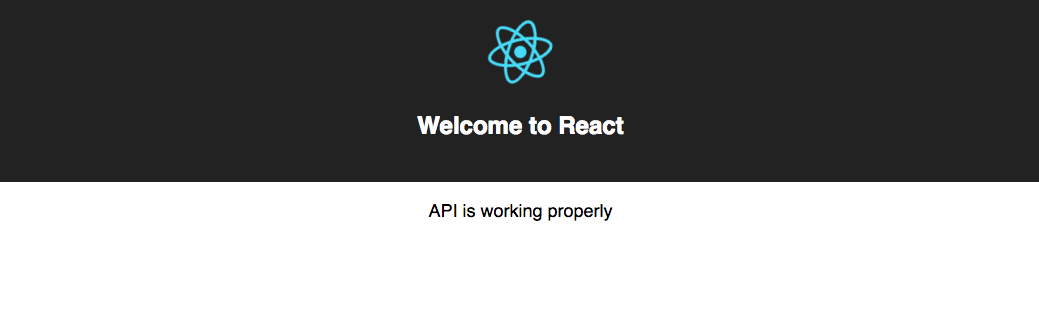
The API **app.js** file should end up something like this:



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* => {

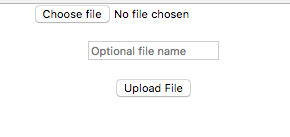
*this*.setState({ imageURL: `http://localhost:4000/${body.file}` });

});

});

}

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/>