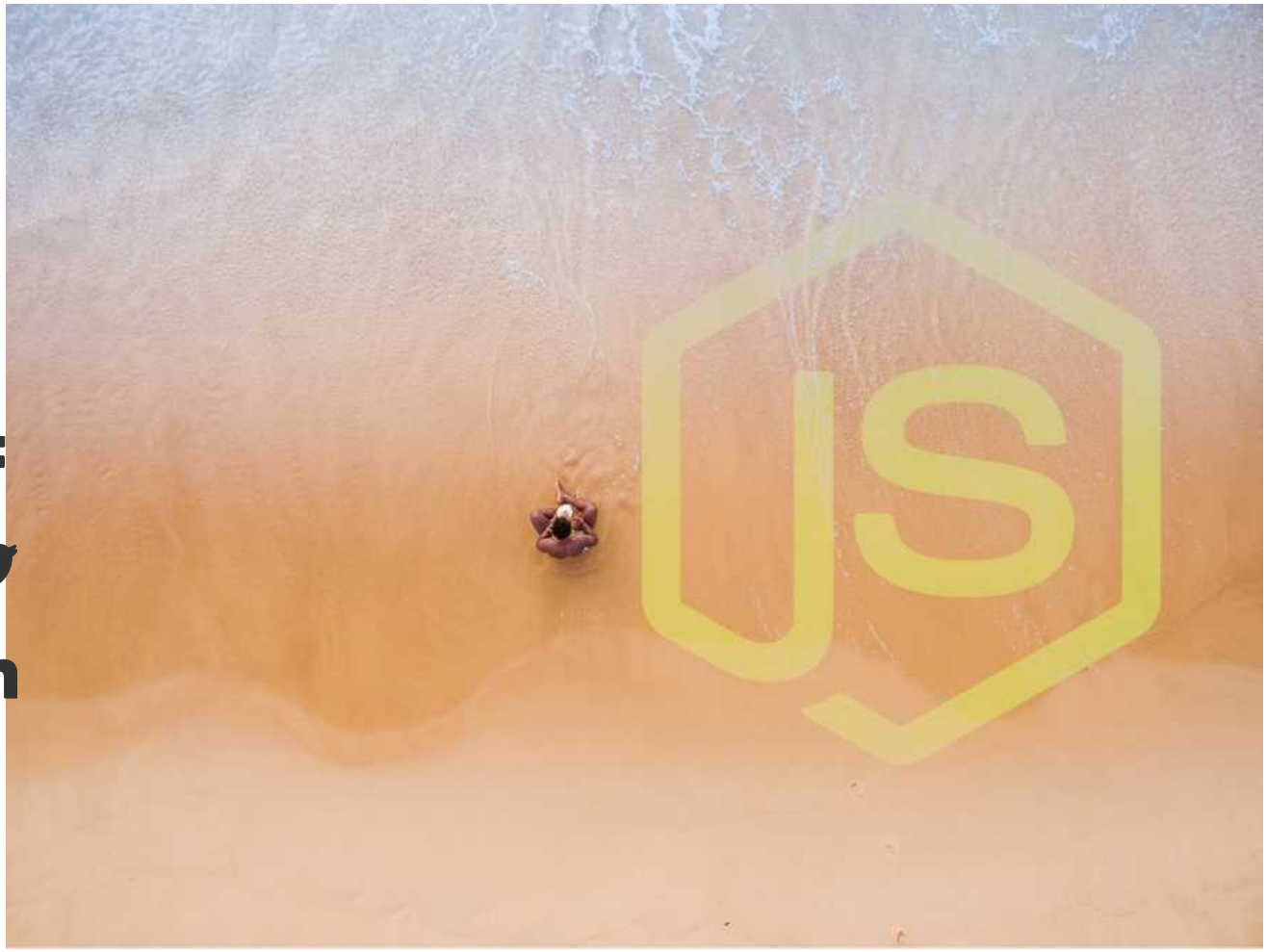
How to setup Express.js in Node.js

APRIL 23, 2020 BY ROBIN WIERUCH - EDIT THIS POST



This tutorial is part 2 of 2 in this series.

Part 1: The minimal Node.js with Babel Setup

Express.js is the most popular choice when it comes to building web applications with Node.js. However, when saying web applications with Node.js, it's often not for anything visible in the browser (excluding server-side rendering of a frontend application). Instead, Express.js, a web application framework for Node.js, enables you to build server applications in Node.js. As a backend application, it is the glue between your frontend application and a potential database or other data sources (e.g. REST APIs, GraphQL APIs). Just to give you an idea, the following is a list of tech stacks to build client-server architectures:

- React.js (Frontend) + Express.js (Backend) + PostgreSQL (Database)

- Vue.js (Frontend) + Koa.js (Backend) + MongoDB (Database)
- Angular.js (Frontend) + Hapi.js (Backend) + Neo4j (Database)

Express.js is exchangeable with other web application frameworks for the backend the same way as React.js is exchangeable with Vue.js and Angular.js when it comes to frontend applications. The Node.js ecosystem doesn't offer only one solution, but various solutions that come with their strengths and weaknesses. However, for this application we will use a Express server, because it is the most popular choice when it comes to building JavaScript backend applications with Node.js.

The Node.js application from before comes with a watcher script to restart your application once your source code has changed, Babel to enable JavaScript features that are not supported in Node.js yet, and environment variables for your application's sensitive information. That's a great foundation to get you started with Express.js in Node.js. Let's continue by installing Express.js in your Node.js application from before on the command line:

```
npm install express
```

Now, in your *src/index.js* JavaScript file, use the following code to import Express.js, to create an instance of an Express application, and to start it as Express server:

```
import express from 'express';

const app = express();

app.listen(3000, () =>
  console.log('Example app listening on port 3000!'),
);
```

Once you start your application on the command line with `npm start`, you should be able to see the output in the command line:

```
Example app listening on port 3000!
```

Your Express server is up and running. Everything that should happen after your Express application has started goes into the callback function. The method itself takes another parameter as first parameter which is the port of the running application. That's why after starting it, the application is available via `http://localhost:3000` in the browser, although nothing should be available at this URL yet when you visit it in your browser.

Routes in Express.js

Routes in web applications for the backend are used to map URIs to middleware. These URIs could serve a text message, a HTML page, or data in JSON via REST or GraphQL. In a larger application, this would mean having several routes (middleware) which map to several URIs. In

Express, a middleware is everything needed for a route, because routes are just another abstraction on top. Let's set up such a single route with Express:

```
import express from 'express';

const app = express();

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(3000, () =>
  console.log('Example app listening on port 3000!'),
);
```

The route points to the root (/) of your domain. In the browser, you can visit this route with `http://localhost:3000/` or `http://localhost:3000` without the trailing slash. Once you save the file, the application should restart automatically due to our setup. You can verify it on the command line. Afterward, visit the browser to see what it outputs for you. You should see the printed "Hello World!" there. In our code, we are using the `res` object's `send` method to send something back to our client application. Whereas the `res` object is everything we need related to specifiyng a *response* for our client, the `req` object is everything we get from this particular *request* from our client. That's it for your first route in Express.js. We will learn more about routes and how to interact with them later.

Essentially every Express application is a just a series of routing and middleware function calls. You have seen the former, the routing with a single route, previously for the `http://localhost:3000` URL or `/` route. You can extend the application with additional URIs (e.g. `http://localhost:3000/example`) by using routes in Express.js (e.g. `/example`) as shown before. Try it yourself!

Middleware in Express.js

If an Express application consists of routing and middleware function calls as mentioned before, what about the middleware function calls then? There are two kinds of middleware in Express.js: application-level middleware and router-level middleware. Let's explore an application-level middleware in this section with a neat use case and dive deeper into the other aspects of both application-level and router-level middleware later.

When using Express.js, people often run into the following error in the browser when accessing their Express application:

```
"Cross-Origin Request Blocked: The Same Origin Policy disallows reading the
```

It most likely happens because we are accessing a domain from a foreign domain. Cross-origin resource sharing (CORS) was invented to secure web applications on a domain level. The idea: It shouldn't be possible to access data from other domains. For instance, a web application with the domain `https://example.com` shouldn't be allowed to access another web application with `https://website.com` by default. CORS is used to restrict access between web applications.

Now, we can allow CORS by adding this missing CORS header, because we will run eventually into this error ourselves when implementing a consuming client application for our Express server. However, since we don't want to do this manually for every route, we can use an application-level middleware to add the CORS HTTP header to every request by default. Therefore, we could write a middleware ourselves -- we will see how this works later -- or use an off the shelf Express.js middleware library which is doing the job for us:

```
npm install cors
```

Next use it as a application-wide middleware by providing it to the Express instance's `use` method:

```
import 'dotenv/config';
import cors from 'cors';
import express from 'express';

const app = express();

app.use(cors());

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(3000, () =>
  console.log(`Example app listening on port 3000!`),
);
```

The Express application can literally *use* a middleware, coming from an external library or built by yourself, to extend all its routes (application-level middleware). In this case, all routes are extended with CORS HTTP headers. By default all routes are accessible for all domains now. This includes later our development domains from our consuming client application too. After all, this was only a sneak peak into an Express middleware. We will learn more about application-level and router-level middleware, and how to write a middleware yourself, later.

Note: Don't worry about the CORS configuration if you didn't fully grasp its purpose yet. It's one of the things many first time Express users run into, have to deal with by installing this neat library, and often never look back why they had to install and use it. If you didn't understand it yet, no worries, but at the time you deploy your application to production, you should set up a whitelist of domains which are allowed to access your Express server application. The CORS library offers this kind of configuration. Take some time to look into it yourself.

Environment Variables in Express.js

Before you have set up environment variables for your Node.js application. Let's use one environment variable to set up your port instead of hardcoding it in the source code. If there isn't such file, create a new *.env* file in your project. Otherwise use the *.env* file that's already there. Give it a new key value pair to define your port:

```
PORT=3000
```

Now in your *src/index.js* file, import the node package which makes the environment variables available in your source code and use the `PORT` environment variable for starting your Express application:

```
import 'dotenv/config';
import cors from 'cors';
import express from 'express';

const app = express();

app.use(cors());

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(process.env.PORT, () =>
  console.log(`Example app listening on port ${process.env.PORT}!`),
);
```

Instead of exposing the port that is used in the source code, you have stored it at a more sensitive place in your environment variables. If you are using Git with something like GitHub, you can exclude the *.env* from being uploaded to the GitHub repository by adding it to your *.gitignore* file. That's how sensitive data is kept away from public repositories like GitHub. If you deploy your application to production eventually, you can add the environment variables as *.env* file on your web server which is serving your application then.

Exercises:

- Confirm your source code for the last section.
- Define for yourself: What's a frontend and a backend application?
- Ask yourself: How do frontend and backend application communicate with each other?
- Optional: Checkout the configuration that can be used with the CORS library.
- Optional: Upload your project to GitHub with Git.
  - Exclude the *.env* file from Git with a *.gitignore* file.
- Explore alternatives for Express.

This tutorial is part 2 of 3 in this series.

Part 1: The minimal Node.js with Babel Setup
Part 3: How to create a REST API with Express.js in Node.js

Show Comments

KEEP READING ABOUT NODE

HOW TO DOCKER WITH NODE.JS

Just recently I had to use Docker for my Node.js web application development. Here I want to give you a brief walkthrough on how to achieve it. First of all, we need a Node.js application. Either take...

CREATING A REST API WITH EXPRESS.JS AND MONGODB

Node + Express + MongoDB is a powerful tech stack for backend applications to offer CRUD operations. It gives you everything to expose an API (Express routes), to add business logic (Express...

PORTFOLIO

Online Courses

Open Source

Tutorials

ABOUT

About me

What I use

How to work with me

How to support me

Contact Me Privacy & Terms