**Handle CRUD in express.js -** [Inpyo Lee](https://medium.com/@1992season?source=post_page-----d49b5f85d4d0--------------------------------) Aug 14, 2022

RESTful API and Back-End on a Front-End Perspective

[***View Source Codes***](https://github.com/WilleLee/crud-in-express)

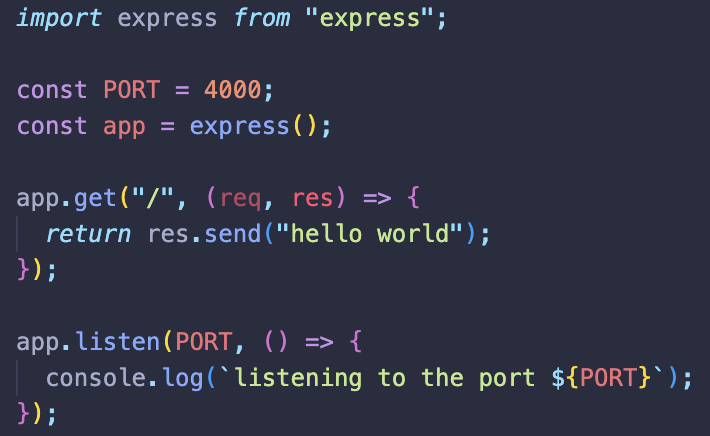
REST(Representational State Transfer) is a software architectural design that describes basic operations of computational storage, which are CRUD(Create, Read, Update and Delete), in a way of following HTTP protocol. APIs that are designed based on the idea of REST, thus are called RESTful.

The core role of front-end development, I think, is to understand the logic of REST of target servers/APIs and to write codes that help clients reach interested/appropriate web pages and resources in terms of that logical process.

Here I want to build a simple express application that deals with resources RESTfully, because having experience in building CRUD interfaces on a back-end side has increased my understanding of what is RESTful and how to deal with it.

**“Hello World” in express.js**

The first thing to do is, of course, to install express.js on your project. You can install express.js with “npm install express” command to be able to use express modules on your project. Let’s import them into your app.js or app.ts.



**./src/app.ts**

Since I installed TypeScript compiler on the project, I could use the latest version of JavaScript and modules here. Or you could call them using *require()* function as the official document shows like;

const express = require("express");  
const app = express();// ...

Codes above become an app that listens to port number 4000 of the host computer, sending a string “hello world” when the client tries to “GET/READ” a page on the path “/”. As such, APIs created by express use “post”, “get”, “put”, and “delete” to **represent** “Create”, “Read”, “Update”, and “Delete” in HTTP protocol, which is very RESTful.

**Add apiRouter**

Having root codes and different API routers in one file makes things complicated so it would be nice to handle routers other than the root file in varied folders. First, let’s create “routes” folder and add apiRouter.ts or .js there.

Text

Description automatically generated with medium confidence

**./src/routes/apiRouter.ts**

Router methods of express.js enable you to create a new router object. Cool thing is that they also have HTTP method routes inside to post, get, put or delete resources responding to clients’ commands or approaches.

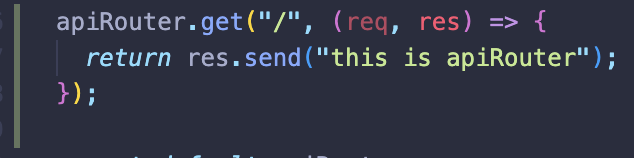
Now your application can **use** that router as a middleware. Here *app.use()* method makes it possible.

Text

Description automatically generated

**./src/app.ts**

I just imported apiRouter that is exported as default, and use it on the application. *use* method receives two arguments, path and callback function, which is a middleware so that the middleware’s invoked on a specific path of the application. Thus, if you let apiRouter gets something on a path “/”, that resource would show up when clients reach the path of “…/api/” like below.



**./src/routes/apiRouter.ts**

Graphical user interface, text, application, chat or text message

Description automatically generated

**localhost:4000/api**

**GET Data**

Although CRUD starts with creating/posting resources to databases, let me begin with reading data from the server, which looks more intuitive, simpler, and closer to what clients do with web pages for the first time. To do so, I created a simple database inside the application, since I don’t use a database like mongo on this project.

Text

Description automatically generated

**./src/database/posts.ts**

Text

Description automatically generated

**./src/routes/apiRouter.ts**

Graphical user interface, text, application

Description automatically generated

**tried using GET method by postman**

Here the *get* method of apiRouter receives the path of “api/posts/:id” and a callback function that takes requests(*req* object) and sends responses(*res* object). I added a colon in front of “id” so that any values on the URL behind “posts/” will be saved as a property *id* inside the property *params* inside the *req* object, which enables me to get any post inside the database by typing the corresponding id on the URL. You can use something like *find* method to catch the target resource and send it with *res* object.

**POST Data**

It’s time to post something to the database.

The path for creating a new post doesn’t seem to require any specific id in it so “api/posts” without any other parameter would be fine to do the job. And I thought it’d be cool if we could read the whole database to check if the new post is added properly with the same path. *route()* method could help you in this case by receiving the path as the only argument, and then it lets us use multiple HTTP method routes on that path.

Text

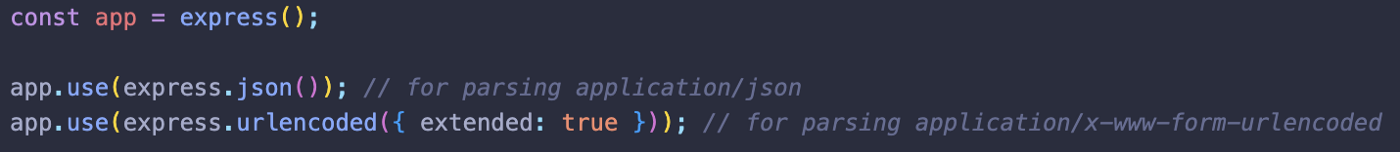
Description automatically generated

**./src/routes/apiRouter.ts**

The id for the new post will be created with *Date.now()* method, while titles and texts must be sent from the client’s request. Using *try … catch …* and *res.status()* method, you could manage unexpected errors and send status codes to the client. Let’s see if it works.

TypeError: Cannot destructure property 'title' of 'req.body' as it is undefined.

Unfortunately, a new post written in JSON was not added to the database with the error message above, which seems to say that *req.body* is an empty object.As the *req.body* section of the express document describes, req.body is undefined by default, and the application requires to use body-parsing middlewares such as *express.json()* or *express.urlencoded()*, so that the server understands JSON files.



**./src/app.ts**

Graphical user interface, text, application, email

Description automatically generated

**POST request sent**

Graphical user interface, text, application, email

Description automatically generated

**database with new posts**

**PUT and DELETE**

Now updating or deleting data would be done quite easily. Those methods require, I thought, to call specific posts based on their *id*s so that such route needs to be added to the same path, “/api/posts/:id”.

Text

Description automatically generated

Since putting and updating posts require posts that already exist, those routes have to contain codes that handle errors that happen when there’s no post with the target *id*. In *put* route, *find()* method was used to see if there was the requested post, and *forEach()* method found the target post and updated data.

On the other hand, I created a new function using *for* loop to have the index of the post that needs to be deleted so that it’s been possible to use *splice()* method to modify the database which is an array. And it worked so well like below.

Graphical user interface, text, application, email

Description automatically generated

**PUT request sent**

Graphical user interface, text, application

Description automatically generated

**database with an updated post**

Graphical user interface, application

Description automatically generated

**DELETE request sent**

Graphical user interface, application

Description automatically generated

**database without the deleted post**