How to create a REST API with Express.js in Node.js

APRIL 24, 2020 BY ROBIN WIERUCH - EDIT THIS POST

Follow on Twitter 17k Follow on Facebook

This tutorial is part 3 of 3 in this series.

Part 1: The minimal Node.js with Babel Setup
Part 2: How to setup Express.js in Node.js

An Express application is most often used as a backend application in a client-server architecture whereas the client could be written in React.js or another popular frontend solution and the server could be written in Express. Both entities result in a client-server architecture (frontend and backend relationship) whereas the backend would be needed for (A) business logic that shouldn't be exposed as source code to the frontend application -- otherwise it would be accessible in the browser -- or for (B) establishing connections to third-party data sources (e.g. database(s)).

However, don't mistake client application *always* for frontend and server application *always* for backend here. These terms cannot be exchanged that easily. Whereas a frontend application is usually something seen in the browser, a backend usually performs business logic that shouldn't be exposed in a browser and often connects to a database as well.

```
Frontend -> Backend -> Database
```

But, in contrast, the terms client and server are a matter of perspective. A backend application (Backend 1) which *consumes* another backend application (Backend 2) becomes a client application (Backend 1) for the server application (Backend 2). However, the same backend application (Backend 1) is still the server for another client application which is the frontend application (Frontend).

```
Frontend -> Backend 1 -> Backend 2 -> Database

// Frontend: Client of Backend 1
// Backend 1: Server for Frontend, also Client of Backend 2
// Backend 2: Server for Backend 1
```

If you want to answer the client-server question if someone asks you what role an entity plays in a client-server architecture, always ask yourself who (server) is serving whom (client) and who (client) consumes whom's (backend) functionalities?

That's the theory behind client-server architectures and how to relate to them. Let's get more practical again. How do client and server applications communicate with each other? Over the years, there existed a few popular communication interfaces (APIs) between both entities. However, the most popular one is called REST defined in 2000 by Roy Fielding. It's an architecture that leverages the HTTP protocol to enable communication between a client and a server application. A server application that offers a REST API is also called a RESTful server. Servers that don't follow the REST architecture a 100% are rather called RESTish than RESTful. In the following, we are going to implement such REST API for our Express server application, but first let's get to know the tooling that enables us to interact with a REST API.

Exercises:

- What's a client-server architecture?
- Read more about REST APIs and other APIs.

CURL FOR REST APIS

If you haven't heard about cURL, this section gives you a short excursus about what's cURL and how to use it to interact with (REST) APIs. The definition taken from Wikipedia says: "*cURL [...] is a computer software project providing a library and command-line tool for transferring data using* *various protocols.*" Since REST is an architecture that uses HTTP, a server that exposes a RESTful API can be consumed with cURL, because HTTP is one of the various protocols.

First, let's install it one the command line. For now, the installation guide is for MacOS users, but I guess by looking up "curl for windows" online, you will find the setup guide for your desired OS (e.g. Windows) too. In this guide, we will use Homebrew to install it. If you don't have Homebrew, install it with the following command on the command line:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/i
```

If you haven't heard about Homebrew, read more about it over here. Next, install cURL with Homebrew:

```
brew install curl
```

Now, start your Express server from the previous sections. Once your application is started, execute `curl http://localhost:3000` in another command line window. Make sure the port matches your port and the Express server is running. After executing the command, you should see the "Hello World!" printed on the command line. Congratulations, you just have consumed your Express server as a client with something else than a browser.

```
Browser (Client) -> Express Server
cURL (Client) -> Express Server
```

Whether you access your Express application on `http://localhost:3000` in the browser or via the command line with cURL, you should see the same result. Both tools act as clients whereas the Express application is your server. You will see in the next sections how to use cURL to verify your Express application's REST API, that we are going to implement together, on the command line instead of in the browser.

Exercises:

- Get yourself more familiar with the terms client/server and frontend/backend.
- If you want to have an alternative for cURL which works in the browser, check out Postman or Isomnia.

EXPRESS ROUTES: HTTP METHODS ARE REST OPERATIONS

Express is a perfect choice for a server when it comes to creating and exposing APIs (e.g. REST API) to communicate as a client with your server application. Previously you have already implemented one Express route, which sends a "Hello World!", that you have accessed via the browser and cURL. Let's set up more routes to accommodate a RESTful API for your Express application eventually. Add the following routes to your Express application whereas the URI itself doesn't change, but the method used from your Express instance:

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

const app = express();

...

app.get('/', (req, res) => {
  return res.send('Received a GET HTTP method');
});

app.post('/', (req, res) => {
  return res.send('Received a POST HTTP method');
});

app.put('/', (req, res) => {
  return res.send('Received a PUT HTTP method');
});

app.delete('/', (req, res) => {
  return res.send('Received a DELETE HTTP method');
});

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

Every Express instance's method maps to a HTTP method. Let's see how this works: Start your Express server on the command line again, if it isn't running already, and execute four cURL commands in another command line window. You should see the following output for the commands:

```
curl http://localhost:3000
-> Received a GET HTTP method

curl -X POST http://localhost:3000
-> Received a POST HTTP method

curl -X PUT http://localhost:3000
-> Received a PUT HTTP method

curl -X DELETE http://localhost:3000
-> Received a DELETE HTTP method
```

By default cURL will use a HTTP GET method. However, you can specify the HTTP method with the `-X` flag (or `--request` flag). Depending on the HTTP method you are choosing, you will access different routes of your Express application -- which here represent only a single API endpoint with an URI so far. You will see later other additions that you can add to your cURL requests.

That's one of the key aspects of REST: It uses HTTP methods to perform operations on URI(s). Often these operations are referred to as CRUD operations for create, read, update, and delete operations. Next you will see on what these operations are used on the URIs (resources).

Exercises:

- Confirm your source code for the last section.
  - Confirm your changes from the last section.
- Read more about CRUD operations.
- Try some more cURL commands yourself on the command line.

EXPRESS ROUTES: URIS ARE REST RESOURCES

Another important aspect of REST is that every URI acts as a resource. So far, you have only operated on the root URI with your CRUD operations, which doesn't really represent a resource in REST. In contrast, a resource could be a user resource, for example. Change your previously introduced routes to the following:

```
...

app.get('/users', (req, res) => {
  return res.send('GET HTTP method on user resource');
});

app.post('/users', (req, res) => {
  return res.send('POST HTTP method on user resource');
});

app.put('/users', (req, res) => {
  return res.send('PUT HTTP method on user resource');
});

app.delete('/users', (req, res) => {
  return res.send('DELETE HTTP method on user resource');
});

...
```

With cURL on your command line, you can go through the resource -- represented by one URI `http://localhost:3000/users` -- which offers all the CRUD operations via HTTP methods:

```
C for Create: HTTP POST
R for Read: HTTP GET
U for Update: HTTP PUT
D for Delete: HTTP DELETE
```

You will see a similar output as before, but this time you are operating on a user resource. For example, if you want to create a user, you hit the following URI:

```
curl -X POST http://localhost:3000/users
-> POST HTTP method on user resource
```

Obviously we don't transfer any information for creating a user yet, however, the API endpoint for creating a user would be available now. One piece is missing to make the PUT HTTP method (update operation) and DELETE HTTP method (delete operation) RESTful from a URI's point of view:

```
...

app.get('/users', (req, res) => {
  return res.send('GET HTTP method on user resource');
});

app.post('/users', (req, res) => {
  return res.send('POST HTTP method on user resource');
});

app.put('/users/:userId', (req, res) => {
  return res.send(
    `PUT HTTP method on user/${req.params.userId} resource`,
  );
});

app.delete('/users/:userId', (req, res) => {
  return res.send(
    `DELETE HTTP method on user/${req.params.userId} resource`,
  );
});

...
```

In order to delete or update a user resource, you would need to know the exact user. That's where unique identifiers are used. In our Express routes, we can assign unique identifiers with parameters in the URI. Then the callback function holds the URI's parameter in the request object's properties. Try again a cURL operation on `/users/1`, `/users/2` or another identifier with a DELETE or UPDATE HTTP method and verify that the identifier shows up in the command line as output.

Exercises:

- Confirm your source code for the last section.
  - Confirm your changes from the last section.
- Try to delete or update a user by identifier with cURL.
- Read more about basic routing in Express.

MAKING SENSE OF REST WITH EXPRESS

You may be still wondering: *What value brings the combination of URIs and HTTP methods* -- which make up the majority of the REST philosophy -- *to my application?*

Let's imagine we wouldn't just return a result, as we do at the moment, but would act properly on the received operation instead. For instance, the Express server could be connected to a database that stores user entities in a user table. Now, when consuming the REST API as a client (e.g. cURL, browser, or also a React.js application), you could retrieve all users from the database with a HTTP GET method on the `/users` URI or, on the same resource, create a new user with a HTTP POST method.

```
// making sense of the naming

Express Route's Method <=> HTTP Method <=> REST Operation
Express Route's Path <=> URI <=> REST Resource
```

Suddenly you would be able to read and write data from and to a database from a client application. Everything that makes it possible is a backend application which enables you to write a interface (e.g. REST API) for CRUD operations:

```
Client -> REST API -> Server -> Database
```

Whereas it's important to notice that the REST API belongs to the server application:

```
Client -> (REST API -> Server) -> Database
```

You can take this always one step further by having multiple server applications offering REST APIs. Often they come with the name microservices or web services whereas each server application offers a well-encapsulated functionality. The servers even don't have to use the same programming language, because they are communicating over a programming language agnostic interface (HTTP with REST). Although the interfaces (APIs) don't have to be necessary REST APIs.

```
       -> (GraphQL API -> Server) -> Database
Client
```

```
        -> (REST API -> Server) -> Database
```

Let's take everything we learned in theory, so far, one step further towards a real application by sending real data across the wire. The data will be sample data, which will not come from a database yet, but will be hardcoded in the source code instead:

```
...

let users = {
  1: {
    id: '1',
    username: 'Robin Wieruch',
  },
  2: {
    id: '2',
    username: 'Dave Davids',
  },
};

let messages = {
  1: {
    id: '1',
    text: 'Hello World',
    userId: '1',
  },
  2: {
    id: '2',
    text: 'By World',
    userId: '2',
  },
};

...
```

Next to the user entities, we will have message entities too. Both entities are related to each other by providing the necessary information as identifiers (e.g. a message has a message creator). That's how a message is associated with a user and how you would retrieve the data from a database, too, whereas each entity (user, message) has a dedicated database table. Both are represented as objects that can be accessed by identifiers.

Let's start by providing two routes for reading the whole list of users and a single user by identifier:

```
...

let users = { ... };

let messages = { ... };

app.get('/users', (req, res) => {
  return res.send(Object.values(users));
});
```

```
app.get('/users/:userId', (req, res) => {
  return res.send(users[req.params.userId]);
});

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

Whereas we pick a user from the object by identifier for the single users route, we transform the user object to a list of users for the all users route. The same should be possible for the message resource:

```
...

let users = { ... };

let messages = { ... };

...

app.get('/messages', (req, res) => {
  return res.send(Object.values(messages));
});

app.get('/messages/:messageId', (req, res) => {
  return res.send(messages[req.params.messageId]);
});

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

Try all four routes with cURL on the command line yourself. That's only about reading data. Next, we will discuss the other CRUD operations to create, update and delete resources to actually write data. However, we will not get around a custom Express middleware and a Express middleware provided by the Express ecosystem. That's why we will discuss the subject of the Express middleware next while implementing the missing CRUD operations.

Exercises:

- Confirm your source code for the last section.
  - Confirm your changes from the last section.
- Read more about REST.
- Read more about GraphQL as popular alternative to REST.

APPLICATION-LEVEL EXPRESS MIDDLEWARE

Before we jump into Express middleware again, let's see how a scenario for creating a message could be implemented in our Express application. Since we are creating a message without a database ourselves, we need a helper library to generate unique identifiers for us. Install this helper library on the command line:

```
npm install uuid
```

Next import it at the top of your *src/index.js* file:

```
import { v4 as uuidv4 } from 'uuid';
```

Now, create a message with a new route that uses a HTTP POST method:

```
...

app.post('/messages', (req, res) => {
  const id = uuidv4();
  const message = {
    id,
  };

  messages[id] = message;

  return res.send(message);
});

...
```

We generate a unique identifier for the message with the new library, use it as property in a message object with a shorthand object property initialization, assign the message by identifier in the messages object -- which is our pseudo database --, and return the new message after it has been created.

However, something is missing for the message. In order to create a message, a client has to provide the `text` string for the message. Fortunately a HTTP POST method makes it possible to send data as payload in a body. That's why we can use the incoming request (`req`) to extract a payload from it:

```
...

app.post('/messages', (req, res) => {
  const id = uuidv4();
  const message = {
    id,
    text: req.body.text,
  };

  messages[id] = message;
  return res.send(message);
});

...
```

Accessing the payload of an HTTP POST request is provided within Express with its built-in middleware which is based on body-parser. It enables us to transform body types from our request object (e.g. json, urlencoded):

```
...
import express from 'express';

const app = express();

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

...
```

This extracts the entire body portion of an incoming request stream and makes it accessible on `req.body`. Now the body with the message's text is accessible in the request whether it is send by a regular POST request or a POST request from a HTML form. Both options should work, because all data should be received and send as JSON payload now. That's another aspect of REST, which itself is no opinionated about the payload format (JSON, XML), but once you have chosen a format (here JSON), you should stick to it for your entire API.

Note that all data that comes with the request object's body tag isn't typed yet. Everything comes as a JSON string. In the case of the message's `text`, we are doing fine with just keeping it as a string. However, for other types you would have to convert the JSON string:

```
const date = Date.parse(req.body.date);
const count = Number(req.body.count);
```

In this last step, we have used a built-in Express middleware and made it available on an application-level -- which means that each request that arrives at one of our Express routes goes through the middleware. Therefore, all data send by a client to our server is available in the incoming request's body. Try it by creating a message yourself: In a cURL request you can specify HTTP headers with the `-H` flag -- that's how we are saying we want to transfer JSON -- and data as payload with the `-d` flag. You should be able to create messages this way:

```
curl -X POST -H "Content-Type:application/json" http://localhost:3000/messa
```

You should see the created messaged returned to you on the command line. You can double check whether the message was really created in your messages object (aka pseudo database) by performing another cURL requests on the command line:

```
curl http://localhost:3000/messages
```

There you should see the new message which has been created for you. In addition, you should also be able to request your new message by identifier. Perform the following cURL request to get a single message entity, but use your actual message identifier for it, because my identifier is different from yours:

```
curl http://localhost:3000/messages/849d9407-d7c6-4712-8c91-1a99f7b22ef5
```

That's it. You have created your first resource (message) via your REST API and requested the same resource (message(s)) from your REST API. On top of that, you have used a built-in Express middleware to make the data available in the request's body object.

So far, we have only imported third-party Express middleware (CORS) or used a built-in Express middleware (body parser) -- both on an application-level. Now, let's build a custom Express middleware ourselves, which will be used on an application-level too. The blueprint for a middleware is similar to the Express functions we have seen before:

```
...

app.use((req, res, next) => {
  // do something
  next();
});

...
```

A middleware is just a JavaScript function which has access to three arguments: `req`, `res`, `next`. You already know `req` and `res` -- they are our request and response objects. In addition, the next function should be called to signalize that the middleware has finished its job. In between of the middleware function you can do anything now. We could simply `console.log()` the time or do something with the request (`req`) or response (`res`) objects.

In our particular case, when creating a message on the message resource, we need to know who is creating the message to assign a `userId` to it. Let's do a simple version of a middleware that determines a pseudo authenticated user that is sending the request. In the following case, the authenticated user is the user with the identifier `1` which gets assigned as `me` property to the request object:

```
app.use((req, res, next) => {
  req.me = users[1];
  next();
});
```

Afterward, you can get the authenticated user from the request object and append it as message creator to the message:

```
app.post('/messages', (req, res) => {
  const id = uuidv4();
  const message = {
    id,
    text: req.body.text,
    userId: req.me.id,
  };

  messages[id] = message;

  return res.send(message);
});
```

You can imagine how such middleware could be used later to intercept each incoming request to determine from the incoming HTTP headers whether the request comes from an authenticated user or not. If the request comes from an authenticated user, the user is propagated to every Express route to be used there. That's how the Express server can be stateless while a client always sends over the information of the currently authenticated user.

Being a stateless is another characteristic of RESTful services. After all, it should be possible to create multiple server instances to balance the incoming traffic evenly between the servers. If you heard about the term load balancing before, that's exactly what's used when having multiple servers at your hands. That's why a server shouldn't keep the state (e.g. authenticated user) -- except for in a database -- and the client always has to send this information along with each request. Then a server can have a middleware which takes care of the authentication on an application-level and provides the session state (e.g. authenticated user) to every route in your Express application.

Now, that you have learned the essentials about application-level middleware in Express, let's implement the last routes to complete our application's routes. What about the operation to delete a message:

```
...

app.delete('/messages/:messageId', (req, res) => {
  const {
    [req.params.messageId]: message,
    ...otherMessages
  } = messages;

  messages = otherMessages;
  return res.send(message);
});

...
```

Here we used a dynamic object property to exclude the message we want to delete from the rest of the messages object. You can try to verify the functionality with the following cURL command:

```
curl -X DELETE http://localhost:3000/messages/1
```

The update operation on a message resource is for you to implement yourself as an exercise. I will spare it for a later section, because it quickly raises a new topic: permissions. The question: Who is allowed to edit a message? It should only be possible for the authenticated user (`me`) who is the creator of the message.

Last, since you have already the pseudo authenticated user at your hands due to the application-wide middleware, you can offer a dedicated route for this resource too:

```
...

app.get('/session', (req, res) => {
  return res.send(users[req.me.id]);
});

...
```

It's the first time you break the rules of being entirely RESTful, because you offer an API endpoint for a very specific feature. It will not be the first time you break the laws of REST, because most often REST is not fully implemented RESTful but rather RESTish. If you want to dive deeper into REST, you can do it by yourself. HATEOAS and other REST related topics are not covered in detail and implemented here.

Exercises:

- Confirm your source code for the last section.
  - Confirm your changes from the last section.
- Read more about using middleware in Express.
  - Focus on the application-level middleware, the built-in middleware, and the third-party middleware.
- Read more about writing middleware in Express.

MODULAR MODELS IN EXPRESS AS DATA SOURCES

At the moment, all of our implementation sits in the *src/index.js* file. However, at some point you may want to modularize your implementation details and put them into dedicated files and folders whereas the *src/index.js* file should only care about putting everything together and starting the application. Before we dive into modularizing the routing, let's see how we can modularize our sample data in so called models first. From your root folder type the following commands to create a folder/file structure for the models.

```
cd src
mkdir models
cd models
touch index.js
```

The models folder in an Express application is usually the place where you define your data sources. In our case, it's the sample data, but in other applications, for instance, it would be the interfaces to the database. In our case of refactoring this, let's move our sample data over to the new *src/models/index.js* file:

```
let users = {
  1: {
    id: '1',
    username: 'Robin Wieruch',
  },
  2: {
    id: '2',
    username: 'Dave Davids',
  },
};

let messages = {
  1: {
    id: '1',
    text: 'Hello World',
    userId: '1',
  },
  2: {
    id: '2',
    text: 'By World',
    userId: '2',
  },
};

export default {
  users,
  messages,
};
```

Remove the sample data afterward in the *src/index.js* file. Also import the models in the *src/index.js* file now and pass them in our custom application-level middleware to all routes via a dedicated context object. That's where the `me` user (authenticated) user can be placed as well.

You don't need necessarily the context object as container, but I found it a good practice to keep everything that is passed to the routes at one place.

```
...

import models from './models';

const app = express();

...

app.use((req, res, next) => {
  req.context = {
    models,
    me: models.users[1],
  };
  next();
});

...
```

Then, instead of having access to the sample data in all routes from outside variables as before -- which is an unnecessary side-effect and doesn't keep the function pure --, we want to use the models (and authenticated user) from the function's arguments now:

```
...

app.get('/session', (req, res) => {
  return res.send(req.context.models.users[req.context.me.id]);
});

app.get('/users', (req, res) => {
  return res.send(Object.values(req.context.models.users));
});

app.get('/users/:userId', (req, res) => {
  return res.send(req.context.models.users[req.params.userId]);
});

app.get('/messages', (req, res) => {
  return res.send(Object.values(req.context.models.messages));
});

app.get('/messages/:messageId', (req, res) => {
  return res.send(req.context.models.messages[req.params.messageId]);
});

app.post('/messages', (req, res) => {
  const id = uuidv4();
  const message = {
    id,
    text: req.body.text,
    userId: req.context.me.id,
  };
  req.context.models.messages[id] = message;

  return res.send(message);
});

app.delete('/messages/:messageId', (req, res) => {
  const {
    [req.params.messageId]: message,
    ...otherMessages
  } = req.context.models.messages;

  req.context.models.messages = otherMessages;

  return res.send(message);
});

...
```

We are using the application-wide middleware to pass the models to all our routes in a context object now. The models are living outside of the *src/index.js* file and can be refactored to actual database interfaces later. Next, since we made the routing independent from all side-effects and pass everything needed to them via the request object with the context object, we can move the routes to separated places too.

Exercises:

- Confirm your source code for the last section.
  - Confirm your changes from the last section.

MODULAR ROUTING WITH EXPRESS ROUTER

So far, you have mounted routes directly on the Express application instance in the *src/index.js* file. This will become verbose eventually, because this file should only care about all the important topics to start our application. It shouldn't reveal implementation details of the routes. Now the best practice would be to move the routes into their dedicated folder/file structure. That's why we want to give each REST resource their own file in a dedicated folder. From your root folder, type the following on the command line to create a folder/file structure for the modular routes:

```
cd src
mkdir routes
cd routes
touch index.js session.js user.js message.js
```

Then, assumed the routes would be already defined, import the all the modular routes in the *src/index.js* file and *use* them to mount them as modular routes. Each modular route receives a URI which in REST is our resource:

```
...

import routes from './routes';

const app = express();

...

app.use('/session', routes.session);
app.use('/users', routes.user);
app.use('/messages', routes.message);

...
```

In our *src/routes/index.js* entry file to the routes module, import all routes form their dedicated files (that are not defined yet) and export them as an object. Afterward, they are available in the *src/index.js* file as we have already used them.

```
import session from './session';
import user from './user';
import message from './message';

export default {
  session,
  user,
  message,
};
```

Now let's implement each modular route. Start with the session route in the *src/routes/session.js* file which only returns the pseudo authenticated user. Express offers the Express Router to create such modular routes without mounting them directly to the Express application instance. That's how we can create modular routes at other places than the Express application, but import them later to be mounted on the Express application's instance as we already have done in a previous step.

```
import { Router } from 'express';

const router = Router();

router.get('/', (req, res) => {
  return res.send(req.context.models.users[req.context.me.id]);
});

export default router;
```

Next, the user route in the *src/routes/user.js* file. It's quite similar to the session route:

```
import { Router } from 'express';
```

```
const router = Router();

router.get('/', (req, res) => {
  return res.send(Object.values(req.context.models.users));
});

router.get('/:userId', (req, res) => {
  return res.send(req.context.models.users[req.params.userId]);
});

export default router;
```

Notice how we don't need to define the `/users` URI (path) but only the subpaths, because we did this already in the mounting process of the route in the Express application (see *src/index.js* file). Next, implement the *src/routes/message.js* file to define the last of our modular routes:

```
import { v4 as uuidv4 } from 'uuid';
import { Router } from 'express';

const router = Router();

router.get('/', (req, res) => {
  return res.send(Object.values(req.context.models.messages));
});

router.get('/:messageId', (req, res) => {
  return res.send(req.context.models.messages[req.params.messageId]);
});

router.post('/', (req, res) => {
  const id = uuidv4();
  const message = {
    id,
    text: req.body.text,
    userId: req.context.me.id,
  };

  req.context.models.messages[id] = message;

  return res.send(message);
});

router.delete('/:messageId', (req, res) => {
  const {
    [req.params.messageId]: message,
    ...otherMessages
  } = req.context.models.messages;

  req.context.models.messages = otherMessages;

  return res.send(message);
});

export default router;
```

Every of our modular routes from Express Router is mounted to our Express application with a dedicated URI in the *src/index.js* file now. The modular routes in the *src/routes* folder only take care of their sub paths and their implementation details while the mounting in the *src/index.js* file takes care of the main path and the mounted modular route that is used there. In the end, don't forget to remove all the previously used routes that we moved over to the *src/routes/* folder in the *src/index.js* file.

Exercises:

- Confirm your source code for the last section.
  - Confirm your changes from the last section.
- Read more about advanced routing in Express.

This tutorial is part 3 of 4 in this series.

Part 1: The minimal Node.js with Babel Setup
Part 2: How to setup Express.js in Node.js
Part 4: Setup PostgreSQL with Sequelize in Express Tutorial

This tutorial is part 3 of 4 in this series.

Part 1: The minimal Node.js with Babel Setup
Part 2: How to setup Express.js in Node.js
Part 4: Setup MongoDB with Mongoose in Express Tutorial

Show Comments

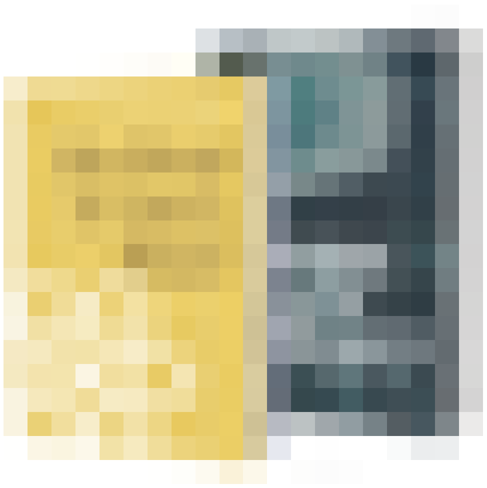
KEEP READING ABOUT DENO >

HOW TO CREATE A REST API WITH OAK IN DENO

An Oak application is most often used as a backend application in a client-server architecture whereas the client could be written in React.js or another popular frontend solution and the server could...

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...



TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

✔ Join 50.000+ Developers

✓ Learn Web Development

✓ Learn JavaScript

✓ Access Tutorials, eBooks and Courses

✓ Personal Development as a Software Engineer

SUBSCRIBE >

View our Privacy Policy.

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