Object Based Routing in Express.js

On the other day, I was looking over the source code of the Sails.js framework. Sails.js framework has a nice routing configuration based on the `routes.js` file. In this file, you can have the exported object in the following way.

```
module.exports = {
  'get /': 'HomeController.index',
  'get /pages/contact': 'PagesController.contact',
  'post /users': 'UsersController.create',
};
```

When the user visit `GET /`, the framework maps the `HomeController.js` file from the `controllers` folder and executes the `index` method or an action (as in terms of MVC). The key in the above object is HTTP (`GET`) verb, followed by a space, and then route(`/`) and the value (of the object key) is the file name without extension(`HomeController`) followed by dot(`.`) and then action name(`index`). Nice and clean mapping!

Sails.js framework is based on Express. In other words, the above code at the end should be converted into standard `app.get()`, `app.post()`, etc. methods. So, I thought about the possibility of adding this object-based routing in a simple Express application. Rest of the article is walk-through on how one can add. I do not yet recommend trying this in a production application.

Create a folder with the name `routing-object`. Go inside, and then create a `package.json` file to mark this folder as a package/module.

```
$ mkdir routing-object
$ cd routing-object
$ npm init -y
```

Install the Express(`express`) as the production dependency and Nodemon(`nodemon`) as the development dependency(`-D`).

```
$ npm i express
$ npm i -D nodemon
```

Nodemon is used in development mode to restart the server when we do the changes. Let's add the `dev` and `start` scripts in `package.json` file to run the application in development and production mode respectively.

```
{
  "name": "routing-object",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "dev": "nodemon app.js",
```

```
    "start": "node app.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "4.18.2"
  },
  "devDependencies": {
    "nodemon": "3.0.0"
  }
}
```

Finally, create an `app.js` file and write the following skeleton Express code.

```
const express = require('express');
const app = express();

app.listen(3000, () => {
  console.log('Application is up and running on port 3000');
});
```

We can easily run this application in development mode using `dev` script.

```
$ npm run dev
[nodemon] starting `node app.js`
Application is up and running on port 3000
```

With these changes, the base Express application is up and running.

Let's add three routes `GET /`, `GET /pages/contact`, and `POST /users` as mentioned in the object at starting of this article. I'll just return simple messages for all these routes.

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.json({ message: 'home page' });
});

app.get('/pages/contact', (req, res) => {
  res.json({ message: 'contact us' });
});

app.post('/users', (req, res) => {
  res.json({ message: 'user created' });
});
```

```
app.listen(3000, () => {
  console.log('Application is up and running on port 3000');
});
```

Test these routes in the browser and/or Insomnia.

With these changes, our application is now supporting the above-mentioned routes.

Now, we're going to do the changes in the application in such a way that it should not affect the output but the structure of the application.

Go ahead and create a `routes.js` file in the route folder.

```
$ touch routes.js
```

Open this file and copy the object mentioned in this article at the beginning.

```
module.exports = {
  'get /': 'HomeController.index',
  'get /pages/contact': 'PagesController.contact',
  'post /users': 'UsersController.create',
};
```

The plan is when the user called `POST /users`, we should look for the file name `UserController.js`. Even further we'll look for this file in the `controllers` folder. So, let's create the `controllers` folder first in the root folder.

```
$ mkdir controllers
```

In this folder, based on the above routes object, we need to create three files - `HomeController.js`, `PagesController.js`, and `UserController.js`.

```
$ touch controllers/HomeController.js
$ touch controllers/PagesController.js
$ touch controllers/UsersController.js
```

`HomeController.js` should have the `index` method.

```
module.exports = {
  index: () => {},
};
```

When the user visit `GET /`, this `index` method ultimately runs. In other words, if we map the following code with the above code,

```
app.get('/', (req, res) => {
  res.json({ message: 'home page' });
});
```

Then

```
(req, res) => {
  res.json({ message: 'home page' });
};
```

should be within the `index` method in `HomeController.js` file as follow.

```
module.exports = {
  index: (req, res) => {
    res.json({ message: 'home page' });
  },
};
```

Similarly, `PagesController.js` should have the `contact` method.

```
module.exports = {
  contact: (req, res) => {
    res.json({ message: 'contact us' });
  },
};
```

And finally, the `UsersController.js` file should have the `create` method.

```
module.exports = {
  create: (req, res) => {
    res.json({ message: 'user created' });
  },
};
```

With these changes, the routes are ready and the associated code is implemented. Next, we need to map these two.

In the `app.js` file, go ahead and remove three routes code as we have that code in other places.

```
const express = require('express');
const app = express();

app.listen(3000, () => {
  console.log('Application is up and running on port 3000');
});
```

To call the method from the files or controller's files, we need to first import these files. For this, we can use require-all npm package. This package requires all the files and makes them available for use to directly call. Go ahead stop the server and install this package.

```
$ npm i require-all
```

As mentioned in the documentation of this package, we can write the following code.

```
const controllers = require('require-all')({
  dirname: __dirname + '/controllers',
  filter: /(.+Controller)\.js$/,
});
```

This `controllers` object now has reference to all the modules or files within `controllers` folder. To call `index` method from `HomeController` we can simply write

```
controllers.HomeController.index();
```

Let's use the `require-all` package in the `app.js` file and require all the controllers.

```
const express = require('express');
const app = express();

const controllers = require('require-all')({
  dirname: __dirname + '/controllers',
  filter: /(.+Controller)\.js$/,
});

app.listen(3000, () => {
  console.log('Application is up and running on port 3000');
});
```

With these changes, we just figure out what to do with the values of the routes object. Next, we need to fetch the routes from `routes.js` file and attach the respected method to each of the routes by loop through.

Let's simply require the `routes.js` file to fetch all the routes.

```
const express = require('express');
const app = express();

const controllers = require('require-all')({
  dirname: __dirname + '/controllers',
  filter: /(.+Controller)\.js$/,
});

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

app.listen(3000, () => {
  console.log('Application is up and running on port 3000');
});

app.listen(3000, () => {
```

```
  console.log('Application is up and running on port 3000');
});
```

We're going to use `for...in` loop to loop through the object.

```
const express = require('express');
const app = express();

const controllers = require('require-all')({
  dirname: __dirname + '/controllers',
  filter: /(.+Controller)\.js$/,
});

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

for (let route in routes) {
}

app.listen(3000, () => {
  console.log('Application is up and running on port 3000');
});
```

The `route` is the key and it contains two things - HTTP verb and route itself. We need to fetch both the details from route. To fetch the HTTP verb, we can simply use the RegEx as follow.

```
const verbExpr = /^(get|post|put|delete)\s+/;
const verb =
  key.match(verbExpr || [])[key.match(verbExpr || []).length - 1] || null;
```

The first line is the RegEx for four HTTP verbs and the second line check for the verbs and return and saved them into the `verb` variable.

Let's add these lines to our `app.js` file.

```
const express = require('express');
const app = express();

const controllers = require('require-all')({
  dirname: __dirname + '/controllers',
  filter: /(.+Controller)\.js$/,
});

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

for (let route in routes) {
  const verbExpr = /^(get|post|put|delete)\s+/;
  const verb =
    key.match(verbExpr || [])[key.match(verbExpr || []).length - 1] || null;
```

```
}

app.listen(3000, () => {
  console.log('Application is up and running on port 3000');
});
```

Next, we need to fetch the route itself. Fetching the route is as simple as replacing the verb with an empty string as whatever remains must be the route.

```
let path = route;
path = path.replace(verbExpr, '');
```

Instead of directly changing the route iterator, I used the intermediate `path` variable and saved the route in it.

Let's add these lines to our `app.js` file.

```
const express = require('express');
const app = express();

const controllers = require('require-all')({
  dirname: __dirname + '/controllers',
  filter: /(.+Controller)\.js$/,
});

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

for (let route in routes) {
  const verbExpr = /^(get|post|put|delete)\s+/;
  const verb =
    key.match(verbExpr || [])[key.match(verbExpr || []).length - 1] || null;

  let path = route;
  path = path.replace(verbExpr, '');
}

app.listen(3000, () => {
  console.log('Application is up and running on port 3000');
});
```

We now have all the needed code but in de-composed mode.

Even though all the methods within controllers are available in the `controllers` object, we can not dynamically call them. We need to save the controllers and methods separately to call.

To save the controllers and method from object value, we can simply split by dot(`.`).

```
let location = routes[key];
location = location.split('.');
let controllerLocation = location[0];
let methodLocation = location[1];
```

The above code saves the value of the route object in `location` and we split it by dot(`.`). With this, we can have the controller in `controllerLocation` and the method in `methodLocation` variables in callable form.

Let's add these lines to our `app.js` file.

```
const express = require('express');
const app = express();

const controllers = require('require-all')({
  dirname: __dirname + '/controllers',
  filter: /(.+Controller)\.js$/,
});

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

for (let route in routes) {
  const verbExpr = /^(get|post|put|delete)\s+/;
  const verb =
    key.match(verbExpr || [])[key.match(verbExpr || []).length - 1] || null;

  let path = route;
  path = path.replace(verbExpr, '');

  let location = routes[key];
  location = location.split('.');
  let controllerLocation = location[0];
  let methodLocation = location[1];
}

app.listen(3000, () => {
  console.log('Application is up and running on port 3000');
});
```

The exact callable form is within `controllers` object. So, let's fetch specific controllers and methods from the controller per route.

```
let controller = controllers[controllerLocation];
let method = controller[methodLocation];
```

Let's add these lines to our `app.js` file.

```
const express = require('express');
const app = express();
```

```
const controllers = require('require-all')({
  dirname: __dirname + '/controllers',
  filter: /(.+Controller)\.js$/,
});

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

for (let route in routes) {
  const verbExpr = /^(get|post|put|delete)\s+/;
  const verb =
    key.match(verbExpr || [])[key.match(verbExpr || []).length - 1] || null;

  let path = route;
  path = path.replace(verbExpr, '');

  let location = routes[key];
  location = location.split('.');
  let controllerLocation = location[0];
  let methodLocation = location[1];

  let controller = controllers[controllerLocation];
  let method = controller[methodLocation];
}

app.listen(3000, () => {
  console.log('Application is up and running on port 3000');
});
```

With these changes, we now have the verb, route, and method to call.

The only code we now need to write is,

```
app[verb](path, method);
```

The above code attaches `.get()` or `.post()` method to `app` and the path and method as the argument to it.

Let's add these lines in our `app.js` file to complete the working Express application.

```
const express = require('express');
const app = express();

const controllers = require('require-all')({
  dirname: __dirname + '/controllers',
  filter: /(.+Controller)\.js$/,
```

```
});

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

for (let route in routes) {
  const verbExpr = /^(get|post|put|delete)\s+/;
  const verb =
    key.match(verbExpr || [])[key.match(verbExpr || []).length - 1] || null;

  let path = route;
  path = path.replace(verbExpr, '');

  let location = routes[key];
  location = location.split('.');
  let controllerLocation = location[0];
  let methodLocation = location[1];

  let controller = controllers[controllerLocation];
  let method = controller[methodLocation];

  app[verb](path, method);
}

app.listen(3000, () => {
  console.log('Application is up and running on port 3000');
});
```

Go ahead and re-run all three routes in the browser and/or Insomnia. You should see the identical output.

The above code is not complete. I intentionally left many scenarios and improvements. For example,

1. You can add all the HTTP verbs in verbExpr and even do case-insensitive matching.
2. You can place this code into a separate file.
3. You can have better names for the variables.
4. You can even implement your require-all package as this package is quite small as one file only.
5. You can add error checking as what if someone forgot to add. between controller and method name of added tab between HTTP verb and route instead of single space, etc.