Chapter 4

Web and REST Development in Node.js API

Business Training

1

Outline

- Getting Started with Express
- Understanding routing
- Rendering Web pages using View Engines
- Exposing REST Resources
- Validation, logging, and Error Handling

2

Getting Started with Express

3

What is Express ?

- **Express.js is a web application framework for Node.js. It provides various features that make web application development fast and easy which otherwise takes more time using only Node.js.**
  - Express.js is based on the Node.js middleware module called connect which in turn uses http module. So, any middleware which is based on connect will also work with Express.js
- **Installing Express in an application**
  - `npm install express –save`
  - It is possible to use express without installing it, through the **npx** command

4

Web and REST Development

Hello World Express

```
const express = require('express')
const app = express()
const port = 3000

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

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`)
})
```

- **This app starts a server and listens on port 3000 for connections. The app responds with "Hello World!" for requests to the root URL (/) or route. For every other path, it will respond with a 404 Not Found.**
  - The req (request) and res (response) are the exact same objects that Node provides
- **Run the app with the following command:**
  - $ node app.js
  - Then, load http://localhost:3000/ in a browser to see the output.

M.Romdhani, 2021 **5**

5

Web and REST Development

Express Generator

- **Use the application generator tool, express-generator, to quickly create an application skeleton.**
- **Running Express Generator**
  - You can run the application generator with the npx command (available in Node.js 8.2.0).
    - `$ npx express-generator`
  - For earlier Node versions, install the application generator as a global npm package and then launch it.
    - `$ npm install -g express-generator`
    - `$ express`
- **Express Generator help**

```
$ express -h
  Usage: express [options] [dir]
  Options:
     -e, --ejs           add ejs engine support
         --hbs           add handlebars engine support
         --pug           add pug engine support
    -H, --hogan          add hogan.js engine support
         --no-view       generate without view engine
    -v, --view <engine> add view <engine> support (ejs|hbs|hjs|jade|pug|twig|vash) (defaults to jade)
    -c, --
css <engine>  add stylesheet <engine> support (less|stylus|compass|sass) (defaults to plain css)
```

6

Generating an app using Express

- **Generate the app**
  - $ express --view=pug myapp
- **Then install dependencies:**
  - $ cd myapp
  - $ npm install
- **On Windows Command Prompt, use this command:**
  - > set DEBUG=myapp:* & npm start
- **On Windows PowerShell, use this command:**
  - PS> $env:DEBUG='myapp:*'; npm start
- **Then load `http://localhost:3000/` in your browser to access the app.**

7

Understanding Routing

8

Basic Routing

- **Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).**
  - Each route can have one or more handler functions, which are executed when the route is matched.
- **Route definition takes the following structure:**
  - `app.METHOD(PATH, HANDLER)`
  - Where:
    - app is an instance of express.
    - METHOD is an HTTP request method, in lowercase.
    - PATH is a path on the server.
    - HANDLER is the function executed when the route is matched.

9

Rendering Web pages using View Engines

10

Serving static files in Express

- **To serve static files such as images, CSS files, and JavaScript files, use the express.static built-in middleware function in Express.**
  - The function signature is:
    - express.static(root, [options])

    The root argument specifies the root directory from which to serve static assets. For more information on the options argument, see express.static.

11

Express View Engines

- **There's a wide variety of template engines that work with Express. The default template engine found in Express is Jade, which is now known as Pug. However, the default Jade installed in Express is still using the old version.**
- **These template engines work "out-of-the-box" with Express:**
  - Pug: Haml-inspired template engine (formerly Jade).
  - EJS: Embedded JavaScript template engine.
  - hbs: Adapter for Handlebars.js, an extension of Mustache.js template engine.

12

Using EJS View Engine

- **Install EJS**
  - $ npm install ejs
- **Call it into your Express app:**

```
const express = require('express');
const app = express();
app.set('view engine', 'ejs');

app.get('/home', (req, res) => {
  res.render('home');
});
```

- **You can create a template in the views directory under views/home.ejs. EJS uses regular HTML for its templates:**

```
<h2>This is the home page</h2>
<p>It's a test view</p>
```


Using EJS View Engine

- **We can pass an array of animals through our Express application to our EJS template.**

```
app.get('/home', (req, res) => {
  let animals = [
    { name: 'Alligator' },
    { name: 'Crocodile' }
  ];
  res.render('home', { animals: animals });
});
```

- **In your views/home.ejs file, you can loop through the data using .forEach:**

```
<h2>This is the home page</h2>
<p>It's a test view</p>
<ul>
  <% animals.forEach((animal) => { %>
  <li><%= animal.name %></li>
  <% }); %>
</ul>
```

Exposing REST Resources

15

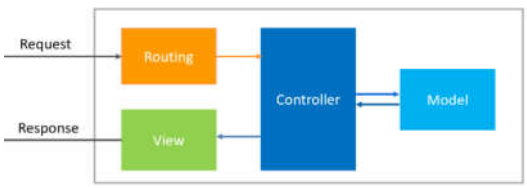
Express Routes

- **A route is a section of Express code that associates an HTTP verb (GET, POST, PUT, DELETE, etc.), a URL path/pattern, and a function that is called to handle that pattern.**
  - encoded in request URLs) to the appropriate controller functions.
  - Routes can be defined in separate route modules

```
app.method('<path>', callbackFunction)

// Examples
app.get('/', callbackFunction)
app.post('/create', callbackFunction)
```

  - The callbackFunction is what to extract from the route definition into a self-contained file and refer to as Controllers



16

API Routes

- **The router object is a utility method that is built on top of express to make RESTful APIs even easier.**
- **REST defines a clear and simple way to define the API endpoints which relies on the HTTP methods or verbs: GET, POST, PUT and DELETE.**
- **Steps for REST API development**
  - Create routes in a Separate module
  - Define the REST API Endpoints as routes

```
router.route('/api/users')
  /** GET /api/users - Get list of users */
  .get(userCtrl.list)

  /** POST /api/users - Create new user */
  .post(userCtrl.create);
```

  - Write the controller methods

CORS Middleware

- **Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served.**
- **How it works ?**
  - The browser sends the GET request with an extra Origin HTTP header to service.example.com containing the domain that served the parent page:Origin: http://www.example.com
  - The server at service.example.com may respond with:
    - The requested data along with an Access-Control-Allow-Origin (ACAO) header in its response indicating the requests from the origin are allowed. For example in this case it should be: **Access-Control-Allow-Origin:***
    - An error page if the server does not allow a cross-origin request
- **Simple Usage (Enable All CORS Requests)**

```
var express = require('express')
var cors = require('cors')
var app = express()
app.use(cors())
```

Validation, Logging and Error handling

19

Data Validation

- **express-validator is a set of express.js middlewares that wraps validator.js validator and sanitizer functions.**
- **Installation**
  - `npm install --save express-validator`
- **Let's get started by writing a basic route to create a user in the database:**

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

app.use(express.json());
app.post('/user', (req, res) => {
  User.create({
    username: req.body.username,
    password: req.body.password,
  }).then(user => res.json(user));
});
```


20

Data Validation

```
// ...rest of the initial code omitted for simplicity.
const { body, validationResult } = require('express-validator');

app.post(
  '/user',
  // username must be an email
  body('username').isEmail(),
  // password must be at least 5 chars long
  body('password').isLength({ min: 5 }),
  (req, res) => {
    // Finds the validation errors in this request and wraps
    //them in an object with handy functions
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }

    User.create({
      username: req.body.username,
      password: req.body.password,
    }).then(user => res.json(user));
  },
);
```


Logging

- **From official doc :**
  - Define error-handling middleware functions in the same way as other middleware functions, except with four arguments instead of three, specifically with the signature `(err, req, res, next))`
- **If you do this where your callback has four arguments:**
  - `app.use(function(error, req, res, next) {...});`
  - then Express assumes this is an error-only middleware handler and will only be called when there are errors.
- **Use morgan https://github.com/expressjs/morgan**
  - Install morgan
    - $ npm install morgan
  - Example:

```
var express = require('express')
var morgan = require('morgan')

var app = express()

app.use(morgan('combined'))

app.get('/', function (req, res) {
  res.send('hello, world!')
})
```

Web and REST Development

Error handling

- **Express comes with a default error handler so you don't need to write your own to get started.**
- **Errors that occur in synchronous code inside route handlers and middleware require no extra work. If synchronous code throws an error, then Express will catch and process it. For example:**

```
app.get('/', function (req, res) {
  throw new Error('BROKEN') // Express will catch this on its own.
})
```

- **For errors returned from asynchronous functions invoked by route handlers and middleware, you must pass them to the next() function, where Express will catch and process them. For example:**

```
app.get('/', function (req, res, next) {
    fs.readFile('/file-does-not-exist', function (err, data) {
      if (err) {
        next(err) // Pass errors to Express.
      } else {
        res.send(data)
      }
    })
  })
```


Web and REST Development

Error handling

- **Starting with Express 5, route handlers and middleware that return a Promise will call next(value) automatically when they reject or throw an error. For example:**

```
app.get('/user/:id', async function (req, res, next) {
    var user = await getUserById(req.params.id)
    res.send(user)
  })
```

Error handling

- **The default error handler**
  - Express comes with a built-in error handler that takes care of any errors that might be encountered in the app. This default error-handling middleware function is added at the end of the middleware function stack.
  - If you pass an error to next() and you do not handle it in a custom error handler, it will be handled by the built-in error handler; the error will be written to the client with the stack trace. The stack trace is not included in the production environment.
- **Writing error handlers**
  - Define error-handling middleware functions in the same way as other middleware functions, except error-handling functions have four arguments instead of three: (err, req, res, next). For example:

```
app.use(function (err, req, res, next) {
  console.error(err.stack)
  res.status(500).send('Something broke!')
})
```