

Implementing WebAPIs in Express.js

Software Engineering - Lab

Marco Robol - marco.robol@unitn.it

Academic year 2024/2025 - Second semester

Contents

- How to implement a web service with [Express.js](#) web framework.

Where are we headed? ... a web service backend

```
// https://github.com/unitn-software-engineering/EasyLib/blob/master/app/books.js
router.get('', async (req, res) => {
  // https://mongoosejs.com/docs/api.html#model_Model.find
  let books = await Book.find({});
  books = books.map( (book) => {
    return {
      self: '/api/v1/books/' + book.id,
      title: book.title
    };
  });
  res.status(200).json(books);
});
```

Creating a Node.js server with the *http module*

We can create a basic web server in Node.js using the standard *http module*.

```
var http = require('http');
var port = 3000;

var requestHandler = function(request, response) {
  const { method, url, headers } = request;
  console.log(request.url);
  response.end('Hello World!');
}

var server = http.createServer(requestHandler);
server.listen(port);
```

Open <http://localhost:3000> in a browser (`ctrl+c` in the terminal to end the execution).

<https://nodejs.org/en/docs/guides/anatomy-of-an-http-transaction/>

Express

How to develop a service exposed through web APIs using Express.

```
$ npm install express --save
```

Express

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. (Source: <https://expressjs.com/>).

Let's rewrite our web server using *Express*:

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

// Handling GET requests
app.get('/', function(req, res){
  res.send('Hello World!');
});

app.listen(port, function() {
  console.log('Server running on port ', 3000);
});
```

Routing with Express

There are a few interesting concepts that we can highlight in this trivial example:

- we can listen to specific http verbs (`app.get`)
- we can listen to specific routes (`'/'`)

Route definition takes the following structure: `app.METHOD(PATH, HANDLER)`

<https://expressjs.com/en/starter/basic-routing.html>

We can focus on the services that we want to implement, without worrying about the logic for handling the request (e.g., checking manually that the request method is GET, and that the request url is '/').

Handling requests in Express

<https://expressjs.com/en/4x/api.html#req>

Handling requests headers, url and query parameters can be done easily:

```
// Handling GET requests
app.get('/search', function(req, res){
  console.log(util.inspect(req.headers, {showHidden: false, depth: null}))
  console.log(util.inspect(req.url, {showHidden: false, depth: null}))
  console.log(util.inspect(req.query, {showHidden: false, depth: null}))
  // res.status(200).send('These are the items found!');
});

// Handling POST requests
app.post('/subscribe', function(req, res){
  console.log(util.inspect(req.headers, {showHidden: false, depth: null}))
  console.log(util.inspect(req.params, {showHidden: false, depth: null}))
  // res.status(201).send('You are now subscribed!');
});
```


Parsing body of the request

We cannot directly access the body of a request. To access the body of a request we need a middleware that concatenate chunks from the stream and parse it for us. Since v4.16+ Express comes with built-in `body-parser` middleware.

- Parsing **JSON** body:

```
app.use(express.json()); //Used to parse JSON bodies
```

- Parsing **URL encoded data**, which is how browsers send *form data*:

```
app.use(express.urlencoded()); //Parse URL-encoded bodies
```

After doing this, form data can be accessed directly using `req.body` as a keys-values object;

Handling response in Express

<https://expressjs.com/it/api.html#res>

```
app.get('/items', function(req, res){  
  ...  
  res.status(200).send('These are the items found!');  
});
```

Body encoding

```
// text of html
res.send('text')
// json
res.json({ user: 'tobi' })
// file
res.sendFile('/absolute/path/to/404.png')
```

Header of the response

In REST, the response of a POST request should provide an empty body and an HTTP header 'Location' with a link to the newly created resource. For example:

```
app.post('/api/products', function (req, res) {
  res.location("/api/products/" + product.id);
})
```

Sending the correct HTTP status codes

```
res.sendStatus(404) // status code  
res.status(404).sendFile('/absolute/path/to/404.png') // chainable status code
```

<https://restfulapi.net/http-status-codes/>

1xx: Informational; 2xx: Success; 3xx: Redirection; 4xx: Client Error; 5xx: Server Error

Using status codes in RESTful APIs?

<https://www.restapitutorial.com/lessons/httpmethods.html>

```
app.post('/api/products', function (req, res) {  
  ...  
  res.status(201).json(body-of-the-response);  
})
```

A complete list of HTTP status code: [restapitutorial.com/httpstatuscodes.html](https://www.restapitutorial.com/lessons/httpstatuscodes.html)

Notes on Implementing RESTful WebAPIs

The response of a POST request should provide an empty body and an HTTP header 'Location' with a link to the newly created resource. For example:

```
app.post('/api/products', function (req, res) {  
  ...  
  res.status(201).location("/api/products/" + product.id);  
  ...  
})
```

Send the correct HTTP status codes:

<https://www.restapitutorial.com/lessons/httpmethods.html>

A complete list of HTTP status code: [restapitutorial.com/httpstatuscodes.html](https://www.restapitutorial.com/httpstatuscodes.html)

Using Postman to test our web server

<https://www.postman.com/>

Play with Postman, submit example requests and analyse what arrives to the server.

- Build your request in Postman - Consider that we are listening to two different routes `subscribe` and `search`, on different HTTP verbs (post and get).
- Specify different **headers** `req.headers` and **query** `req.query` parameters.
- What if we want to send a **body** with our request? Can we directly access `req.body` ?
- Send **body** with different encodings and try different body-parser to access `req.body`

What are middlewares, and how do they work? ...

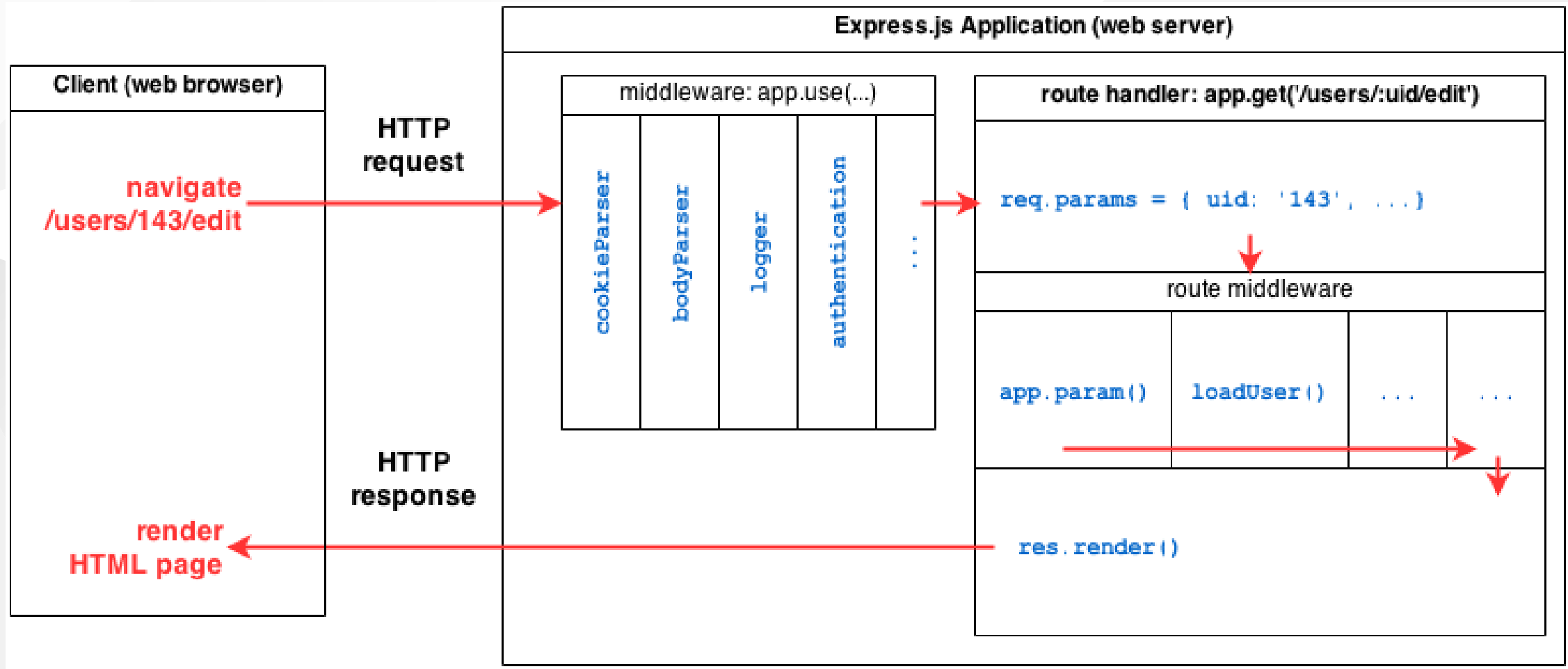


Figure by [hannahhoward](#)

Serving static files

If we had to implement a way to serve static files, one way would be to:

1. Check the request URL
2. Look for the file in the local file system
3. Read the file
4. Check the type/format, and set response headers manually

This requires quite some work, fortunately express provides some standard way of managing common features like this one. Look at the example `mid-static`.

<https://expressjs.com/en/starter/static-files.html>

Serving static files

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

// Serving static files
app.use(express.static('public'));

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

app.listen(port, function() {
  console.log('Server running on port ', 3000);
});
```

Serving static files

What the above does is to mount the built-in `static` middleware, which facilitates the task of servicing static assets.

Run the script and then open <http://localhost:3000> in your browser. What happens when you request the following?:

- <http://localhost:3000/hello>
- <http://localhost:3000/index.html>
- <http://localhost:3000/image1.jpg>

You can decide where the static files will be served, by simply specifying the root as first parameter in `app.use` :

```
app.use('/static', express.static('public'));
```

EasyLib

Web service for the management of book lendings to students.


Repository: <https://github.com/unitn-software-engineering/EasyLib>

APIs documentation: <https://easylib.docs.apiary.io/#>


Questions?

marco.robol@unitn.it

Swagger UI Express

swagger-ui-express 

4.1.6 • Public • Published a year ago

 [Readme](#)

 [Explore](#) 

 1 Dependency

 1,126 Dependents

 51 Versions

Swagger UI Express

This module allows you to serve auto-generated **swagger-ui** generated API docs from express, based on a `swagger.json` file. The result is living documentation for your API hosted from your API server via a route.

Swagger version is pulled from npm module swagger-ui-dist. Please use a lock file or specify the version of swagger-ui-dist you want to ensure it is consistent across environments.

You may be also interested in:

- **swagger-jsdoc**: Allows you to markup routes with jsdoc comments. It then produces a full swagger yaml config dynamically, which you can pass to this module to produce documentation. See below under the usage section for more info.
- **swagger tools**: Various tools, including swagger editor, swagger code gen etc.

Usage

Install

```
> npm i swagger-ui-express
```

Repository

 [github.com/scottie1984/swagger-ui-exp...](https://github.com/scottie1984/swagger-ui-express)

Homepage

 [github.com/scottie1984/swagger-ui-exp...](https://github.com/scottie1984/swagger-ui-express)

Weekly Downloads

851,487



Version

4.1.6

License


MIT

This module allows you to serve auto-generated swagger-ui generated API docs from express, based on a swagger.json file. The result is living documentation for your API hosted from your API server via a route.


```
const express = require('express');
const app = express();
const swaggerUi = require('swagger-ui-express');
const swaggerDocument = require('./swagger.json');


app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerDocument));
```

swagger-jsdoc

swagger-jsdoc 

6.2.1 • Public • Published 4 days ago

 Readme

 Explore BETA

 6 Dependencies

 295 Dependents

 86 Versions

swagger-jsdoc

This library reads your **JSDoc**-annotated source code and generates an **OpenAPI (Swagger) specification**.

downloads **1.1M/month**  CI **failing**

Getting started

Imagine having API files like these:

```
/**
 * @openapi
 * /:
 *   get:
 *     description: Welcome to swagger-jsdoc!
```


Install

```
> npm i swagger-jsdoc
```

Repository

 github.com/Surnet/swagger-jsdoc

Homepage

 github.com/Surnet/swagger-jsdoc

Weekly Downloads

220.570

Version

6.2.1

License

MIT

This library reads your JSDoc-annotated source code and generates an OpenAPI (Swagger) specification. Imagine having API files like these:

```
/**
 * @openapi
 * /:
 *   get:
 *     description: Welcome to swagger-jsdoc!
 *     responses:
 *       200:
 *         description: Returns a mysterious string.
 */
app.get('/', (req, res) => {
  res.send('Hello World!');
});
```


The library will take the contents of @openapi (or @swagger) with the following configuration:

```
const swaggerJsdoc = require('swagger-jsdoc');

const options = {
  definition: {
    openapi: '3.0.0',
    info: {
      title: 'Hello World',
      version: '1.0.0',
    },
  },
  apis: ['./src/routes*.js'], // files containing annotations as above
};

const openapiSpecification = swaggerJsdoc(options);
```

The resulting openapiSpecification will be a swagger validated specification.

swagger-ui-express and swagger-jsdoc

Swagger specification auto-generated and served from express.

```
const swaggerUI = require('swagger-ui-express')
const swaggerJsDoc = require('swagger-jsdoc')

const swaggerOptions = {
  definition: {
    openapi: '3.0.0',
    info: {
      title: 'Hello World',
      version: '1.0.0',
    },
  },
  apis: ['./src/routes*.js'], // files containing annotations as above
};

const swaggerDocument = swaggerJsDoc(swaggerOptions);
app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerDocument));
```