

# WebAPIs with Express (Node.js) and apiary.io

Software Engineering - Part 2 - Lab

Marco Robol - [marco.robol@unitn.it](mailto:marco.robol@unitn.it)

*Academic year 2021/2022 - Second semester*

# Plan for the second part of the course

- April 19-21: Design thinking, project arch, API
- ***April 26-28: Foundations JS, Node.js, git***
- May 2-5: Agile Methodology, MongoDB, API
- May 9 - May 22: Sprint #1
  - More on agile methodology, testing, git branching
- May 23 - June 7: Sprint #2
  - More on testing, devops/CI

# Class outline for this week

## Tuesday

- Web2.0 - Technologies (AJAX, XMLHttpRequest, ...)
- Javascript - Basics of the language
- [Node.js](#) - Asynchronous programming

## *Wednesday - today*

- [Express](#) - WebAPIs with Node.js
- [apiary.io](#) - Documenting WebAPIs

## Thursday

- [Git](#) - Versioning and collaboration with Git and Github

# Contents of today class

- Express <https://expressjs.com/>
- WebAPIs
- [apiary.io](https://apiary.io)

Lab repository: <https://github.com/unitn-software-engineering/2022-se-lab.git>

# Node.js

**Node.js** is a server-side platform built on Google Chrome's JavaScript Engine (V8 Engine). Node.js uses an **event-driven, single threaded, non-blocking I/O model** that makes it lightweight and efficient, perfect for **data-intensive real-time applications** that run across distributed devices.

You can install Node.js by following the instructions from the Node.js project webpage (<https://nodejs.org/en/>).

If you're using a package manager in your OS, you might find ports already available. For example:

- [Installing nodejs using MacPorts.](#)
- [Installing nodejs in Ubuntu](#)
- If you're using anything else, you probably know what you're doing :)

# 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/>

# WebAPIs development with Express

How to develop a backend server exposing a REST APIs using Express.

# 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 specify 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 '/').

# The Request object

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!');
});
```

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

# The response object

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

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

## What about HTTP status codes?

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

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

## 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` ?

# Request body

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.

- We can use the body-parser library. To install `npm install --save body-parser`.

```
// Load the body-parser module
var bodyParser = require('body-parser');
// Mount body-parser middleware, and instruct it to process form url-encoded data
app.use(bodyParser.urlencoded());
app.use(bodyParser.json());
```

- If you are using Express 4.16+ you can now replace body-parser with:

```
app.use(express.json()); //Used to parse JSON bodies
app.use(express.urlencoded()); //Parse URL-encoded bodies
```

After doing this, we should be able to access the form data directly using `req.body`

## Parsing request body contents

Notice that in this case we were parsing form data, but depending on the type of data you want your service to handle, you'll need a different type of parsing.

- **bodyParser.raw():** Doesn't actually parse the body, but just exposes the buffered up contents from before in a Buffer on req.body.
- **bodyParser.text():** Reads the buffer as plain text and exposes the resulting string on req.body.
- **bodyParser.urlencoded():** Parses the text as URL encoded data (which is how browsers tend to send form data from regular forms set to POST) and exposes the resulting object (containing the keys and values) on req.body. For comparison; in PHP all of this is automatically done and exposed in \$\_POST.
- **bodyParser.json():** Parses the text as JSON and exposes the resulting object on req.body.

## Use Postman to test our web server - part 2

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

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

- Send **body** with different encodings and try different body-parser to access `req.body`

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

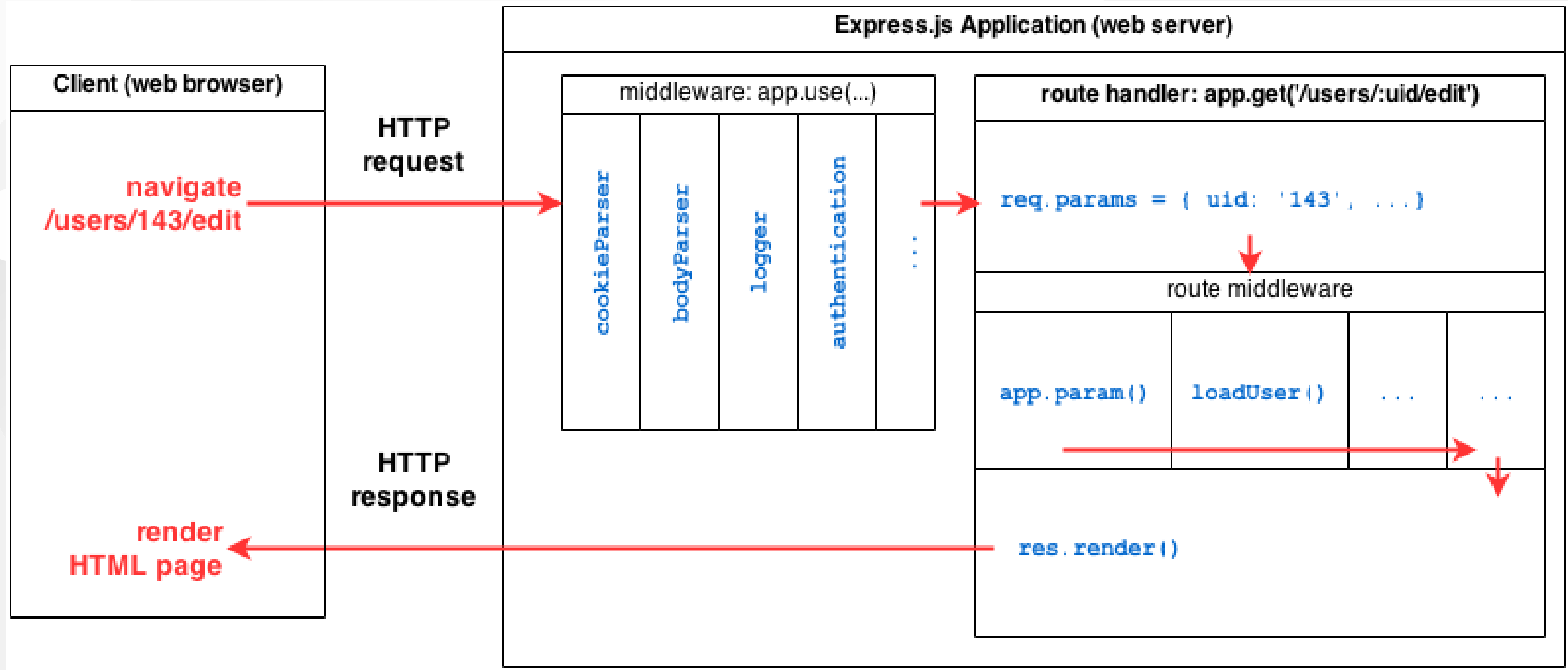


Figure by [hannahhoward](#)



## 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'));
```

# RESTful APIs

# Intro

*Representational State Transfer (REST) is an architectural style for creating web services. REST-compliant web services allow the requesting systems to access and manipulate textual representations of web resources by using a uniform and predefined set of stateless operations.* (Source [Wikipedia](#))

**Online:** A complete guide to RESTful [restapitutorial.com](https://restapitutorial.com)

- **web resource:** any resource on the web that can be identified by an URI (universal resource identifier - urls are the most common type of identifiers).
- **text representation:** json, xml, ...
- **operations:** In our case we are talking about HTTP operations (GET, POST, PUT, DELETE)

## Resources

Web resources can be identified by an URI (universal resource identifier - urls are the most common type of identifiers).

For example, let's say we want to implement a REST API to manage **products**.

```
{  
  "id" : 1,  
  "name" : "iPhone XL",  
  "description" : "Extra large"  
}
```

Complex APIs require special attention to the relationship between web resources, and ways of traversing the relationships. The same resource could be accessed under different URLs. For example, to get the list of products associated to a company ( `/company/:id/products` ).

To easily navigate through resources, we should use links in place of ids. For example:

```
{  
  "self" : "/products/1",  
  "name" : "iPhone XL",  
  "description" : "Extra large",  
  "producer" : "/company/1"  
}
```

*Read the following article:* [API design: Why you should use links, not keys, to represent relationships in APIs](#)

# Operations

CRUD operations are mapped to the standard HTTP verbs. In our example we will have that:

Operation	HTTP Verb	URI	Req body	Resp body	success
Search	GET	/products	Empty	[Product+]	200
Create	POST	/products	Product	Product	201
Read	GET	/products/:id	Empty	Product	200
Update	PUT / PATCH	/products/:id	Product*	Product	200
Delete	DELETE	/products/:id	Empty	Empty	204



To specify the body of the response in Express.js use:

```
res.send(body);
```

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

*Read the following article:* Using status code 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:  
[www.restapitutorial.com/httpstatuscodes.html](http://www.restapitutorial.com/httpstatuscodes.html).

# Documenting your APIs


*Online documentation:* <https://swagger.io/docs/specification/about/>

OpenAPI Specification (formerly Swagger Specification) is an API description format for REST APIs. An OpenAPI file allows you to describe your entire API, including: endpoints and operations, input and output parameters, authentication.

Use the following tools to document and test your APIs:

- <https://app.apiary.io/>
- <https://www.postman.com/>

# 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**  **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](https://github.com/Surnet/swagger-jsdoc)

### Homepage

 [github.com/Surnet/swagger-jsdoc](https://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));
```

## 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](mailto:marco.robol@unitn.it)