Web Technologies 1

Back-end server applications with Express.js

Maxime Guériau & Alexandre Pauchet

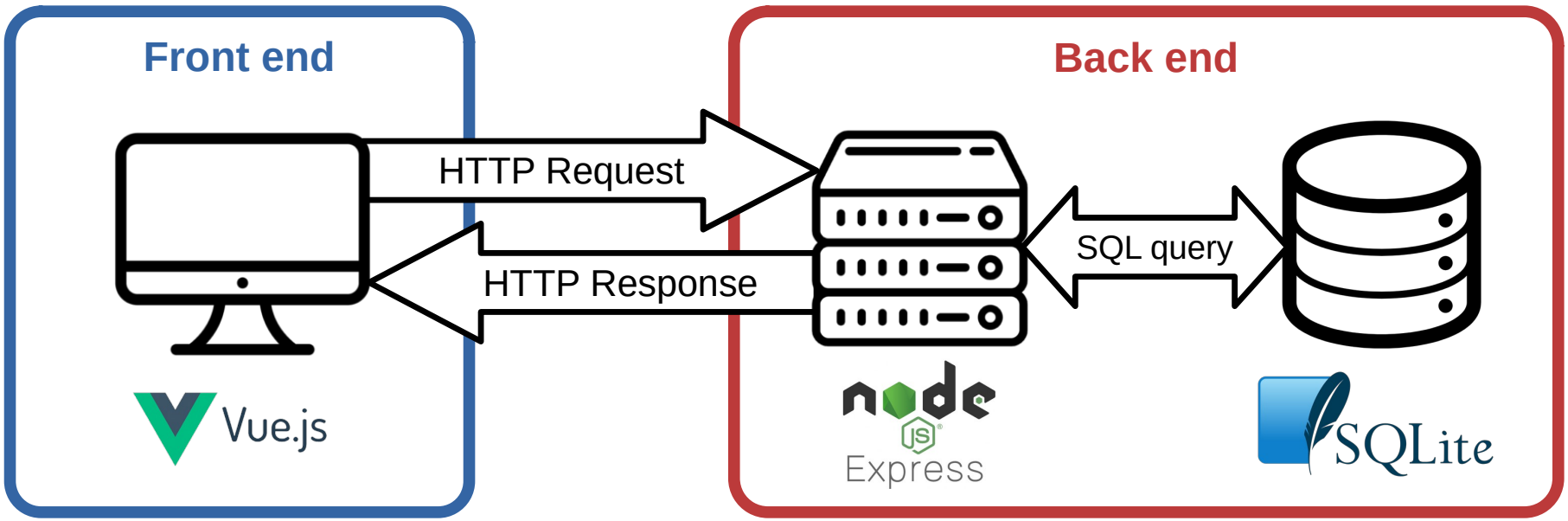
INSA Rouen - Département ASI

BO.B.RC.18, pauchet@insa-rouen.fr

Plan

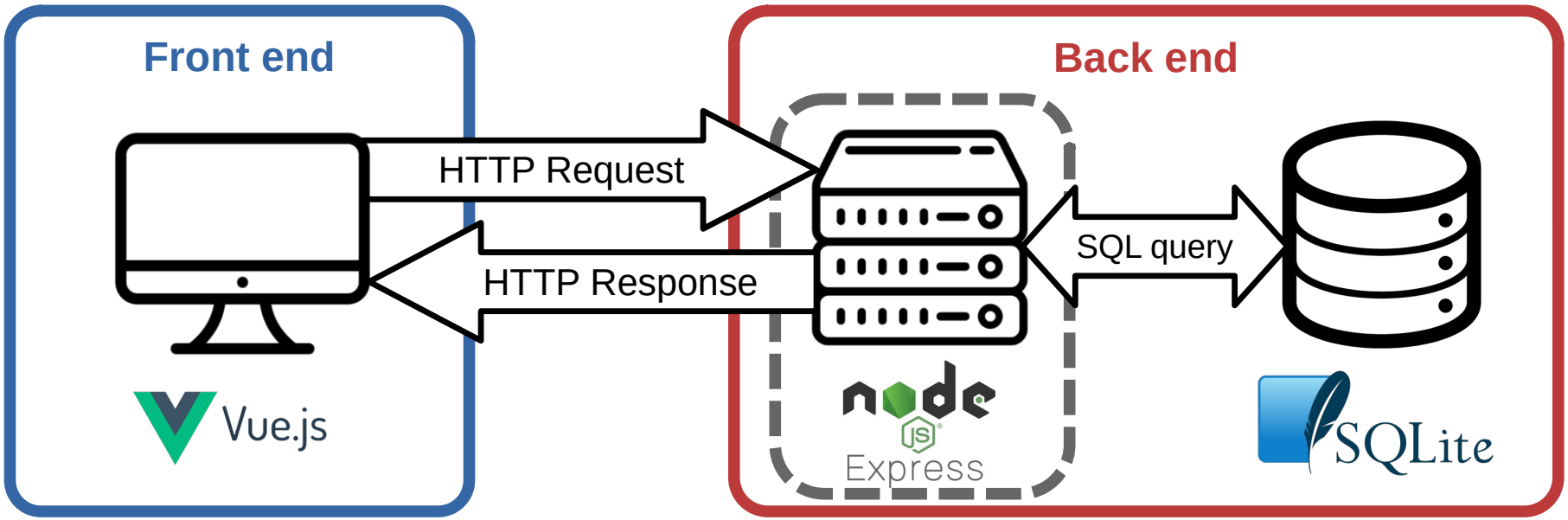
Full stack web development

Spoil (TW2) and today's plan



Full stack web development

Spoil (TW2) and today's plan



Back-end frameworks

Why using a framework?

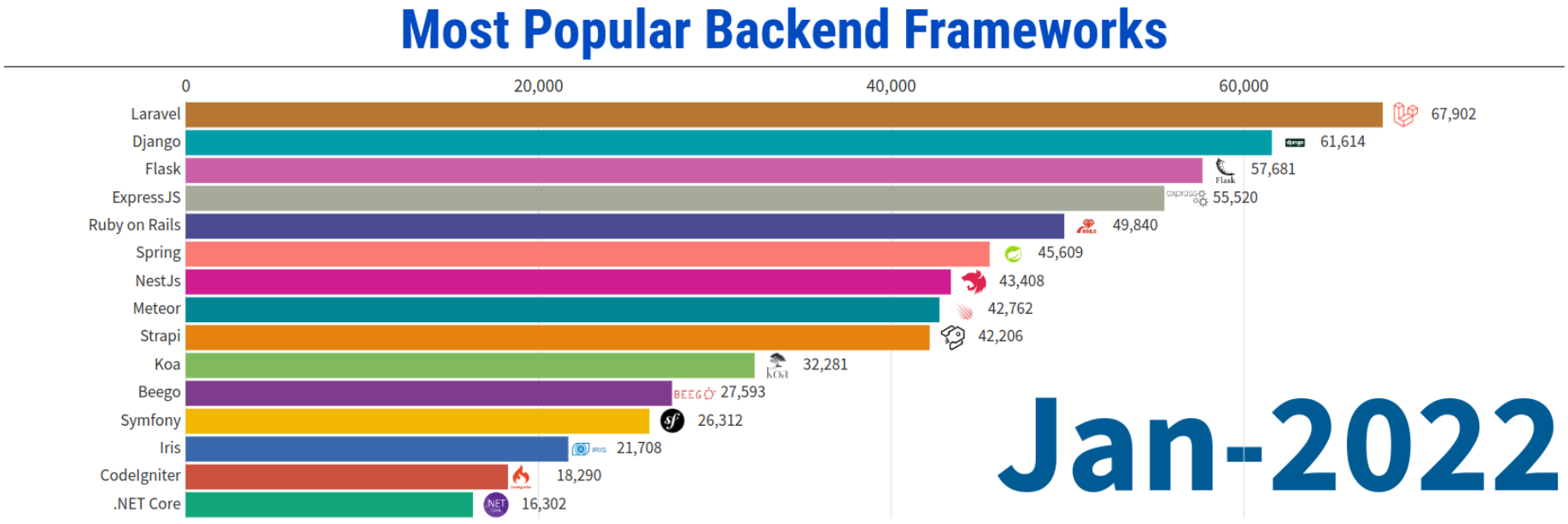
Problem: NodeJS server APIs are actually pretty low-level:

- you build/read the request manually
- you write/build the response manually
- you need a lot of (boring) processing code

Solution: Choose and use an existing framework!

Back-end frameworks

Which one?



Source: [1]

Your first Express.js server

NodeJS vs. ExpressJS (rappel)

NodeJSMini/server.js (adapted from [2])

```
1  // import http module; documentation: https://nodejs.org/api/http.html
2  const http = require('http');
3
4  // set the server port
5  const port = 3000;
6
7  // create a http server endpoint
8  const server = http.createServer((req, res) => {
9      // set the response of your endpoint
10     res.end('Hello World!');
11 });
12
13 // run the server
14 server.listen(port, () => {
15     // callback executed when the server is launched
16     console.log(`Server running on port ${port}`);
17 });
```

ExpressJSMini/server.js (from [3])

```
1  // import express module and create your express app
2  const express = require('express');
3  const app = express();
4
5  // set the server host and port
6  const port = 3000;
7
8  // create your express server endpoint
9  app.get('/', function (req, res) {
10     // set the response of your endpoint
11     res.send('Hello World!');
12 });
13
14 // run the server
15 app.listen(port, () => {
16     // callback executed when the server is launched
17     console.log(`Express app listening on port ${port}`);
18 });
```

Your first Express.js server

Installation (rappel)

Remarque

Express is not part of the NodeJS APIs

⇒**We need to install Express via npm!**

- To install Express, simply run:

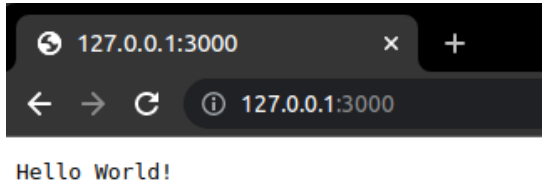
```
npm install express
```

- and then run your (express app) server:

```
node server.js
```

output:

```
Express app listening on port 3000
```



Your first Express.js server

Hello world example [4]

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

The `require()` lets us load the ExpressJS module.

The module actually contains a function that creates a new Express Application object.

Your first Express.js server

Hello world example [4]

```
app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
})
```

The ExpressJS `listen()` is identical to the NodeJS `listen()` function:

- This binds the server process to the given **port number**.
- Now messages sent to the OS's port 3000 will be routed to this server process.
- The function parameter is a callback that will execute when it starts listening for HTTP messages (when the process has been bound to port 3000)

Your first Express.js server

Hello world example [4]

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

app.***method***(***path***, ***handler***)

- Specifies how the server should handle HTTP ***method*** requests made to URL/***path***
- The function callback will fire every time there's a new response.
- This example is saying: When there's a GET request to http://localhost:3000/, respond with the text "Hello World!"

Your first Express.js server

Hello world example [4]

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

Express has its own `Request` and `Response` objects:

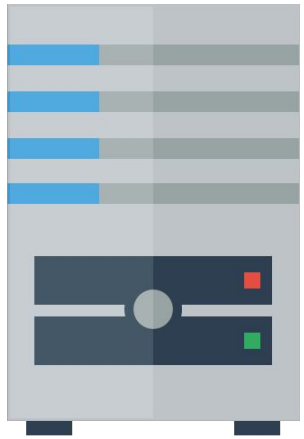
- `req` is a Request object
- `res` is a Response object
- `res.send()` sends an HTTP response with the given content
  - Sends content type "text/html" by default

Building web APIs with ExpressJS

API endpoints [4]

Sometimes when you type a URL into your browser, the URL represents **an API endpoint**.

That is, the URL represents **a parameterized request**, and the web server dynamically generates a response to that request.



That's how our NodeJS server treats routes defined like this:

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

Building web APIs with ExpressJS

API endpoints [4]

Server-side web API [5]

”A server-side web **Application Programming Interface** is a programmatic interface consisting of one or more publicly exposed *endpoints* to a defined request–response message system, typically expressed in JSON or XML, which is exposed via the web—most commonly by means of an HTTP-based web server.”

API endpoint [6]

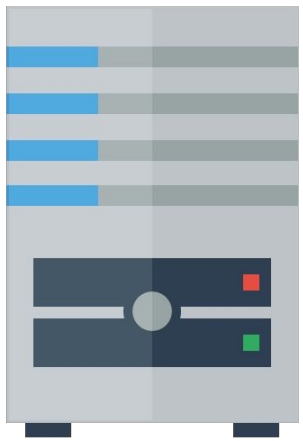
”An API endpoint is the point of entry in a communication channel when two systems are interacting. It refers to touchpoints of the communication between an API and a server. The endpoint can be viewed as the means from which the API can access the resources they need from a server to perform their task. **An API endpoint is basically a fancy word for a URL of a server or service.**”

Building web APIs with ExpressJS

API endpoints [4]

Other times when you type a URL in your browser, the URL is a **path to a file** on the hard drive of the server:

- The web server software grabs that file from the server's local file system, and sends back its contents to you



We can make our NodeJS server also sometimes serve files "statically," meaning instead of treating **all** URLs as API endpoints, some URLs will be treated as file paths.

Building web APIs with ExpressJS

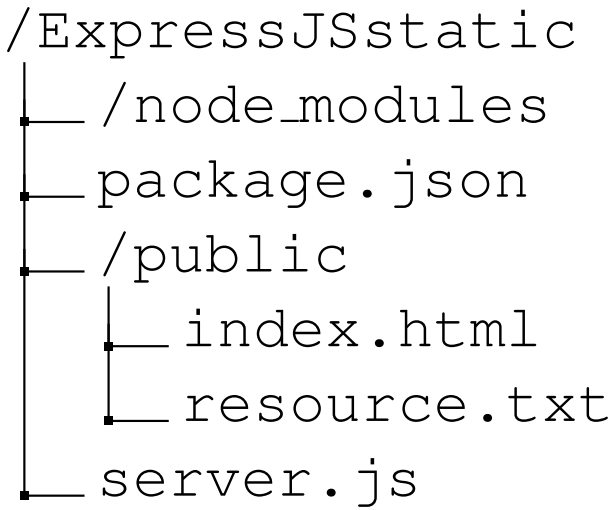
Serving static files [4]

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

app.use(express.static('public'));

app.get('/', function (req, res) {
  res.send('Main page!');
});
```

This line of code makes our server now start serving the files in the 'public' directory directly.

Building web APIs with ExpressJS

Example: ExpressJSstatic



ExpressJSstatic/server.js ...

```
5 // enable your express app to serve static files located in
      ”public” directory
6 app.use(express.static('public'));
```

...

Building web APIs with ExpressJS

Managing dependencies (rappel) [4]

When you upload NodeJS code to a GitHub repository (or any code repository), you should **not** upload the `node_modules` directory:

- You shouldn't be modifying code in the `node_modules` directory, so there's no reason to have it under version control
- This will also increase your repo size significantly

Q: But if you don't upload the node_modules directory to your code repository, how will anyone know what libraries they need to install?

Building web APIs with ExpressJS

Managing dependencies (rappel) [4]

If we don't include the `node_modules` directory in our repository, we need to somehow tell other people what `npm` modules they need to install.

`npm` provides a mechanism for this: `package.json`

Building web APIs with ExpressJS

Managing dependencies (rappel) [4]

You can put a file named `package.json` in the root directory of your NodeJS project to specify metadata about your project.

Create a `package.json` file using the following command:

```
$ npm init
```

This will ask you a series of questions then generate a `package.json` file based on your answers.

Building web APIs with ExpressJS

Managing dependencies (rappel)

Example of an auto-generated `package.json`:

```
{
    "name": "expressjsstatic",
    "version": "1.0.0",
    "description": "",
    "main": "server.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1",
        "start": "node server.js"
    },
    "author": "Alexandre Pauchet",
    "license": "ISC",
    "dependencies": {
        "express": "^4.18.1"
    }
}
```

Building web APIs with ExpressJS

Managing dependencies (rappel) [4]

Now when you install packages, you should pass in the --save parameter:

```
$ npm install --save express
$ npm install --save body-parser
```

This will also add an entry for this library in package.json.

```
"dependencies": {
  "body-parser": "^1.17.1",
  "express": "^4.15.2"
},
```

Building web APIs with ExpressJS

Managing dependencies (rappel) [4]

If you remove the node_modules directory:

```
$ rm -rf node_modules
```

You can install your project dependencies again via:

```
$ npm install
```

- This also allows people who have downloaded your code from GitHub to install all your dependencies with one command instead of having to install all dependencies individually.

Building web APIs with ExpressJS

Managing dependencies (rappel) [4]

Your package.json file also defines scripts:

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "node server.js"
},
```

You can run these scripts using `$ npm` ***scriptName***

E.g. the following command runs "node server.js"

```
$ npm start
```

Building web APIs with ExpressJS

Managing dependencies (rappel): fixing `nodemon`

An other example of dependency is `nodemon` module, that you can install as a local development dependency:

```
npm install --save-dev nodemon
```

Doing so will add a few specific lines in your `package.json`:

```
...
"devDependencies": {
    "nodemon": "^2.0.20"
},
...
```

You can not use `nodemon` directly as a command line but you can update your `package.json` so `npm` can now use `nodemon`:

```
...
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon server.js"
},
...
```

Express middleware functions

Definitions [7]

Express

Express is a **routing and middleware web framework** that has minimal functionality of its own: an Express application is essentially a series of middleware function calls

Middleware

Middleware functions are functions that have access to:

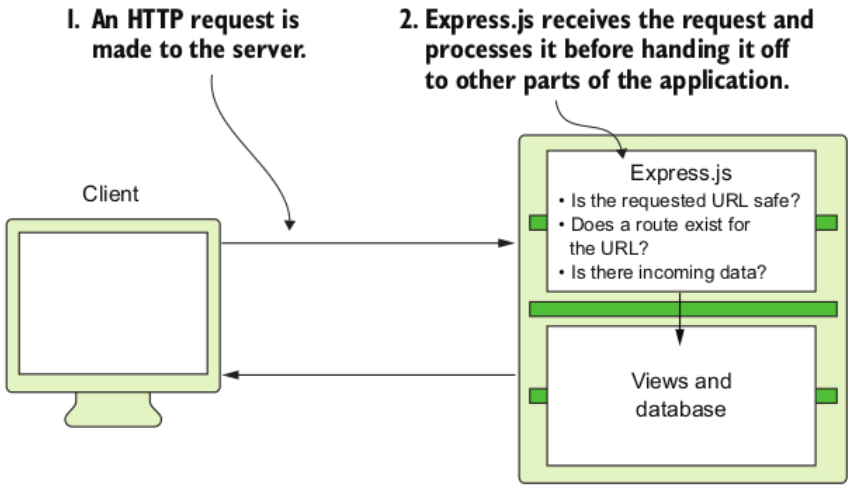
- the request object (`req`),
- the response object (`res`),
- and the next middleware function in the application's request-response cycle.

Express middleware functions

Definitions [7]

Middleware functions can perform the following tasks:

- Execute any code;
- Make changes to the request and the response objects;
- Call the next middleware function in the stack (using `next()`);
- End the request-response cycle.



Express middleware functions

Definitions [7]

An Express application can use the following types of middleware:

1. **Application-level** middleware
2. **Error-handling** middleware
3. **Built-in** middleware
4. **Third-party** middleware
5. **Router-level** middleware

Express middleware functions

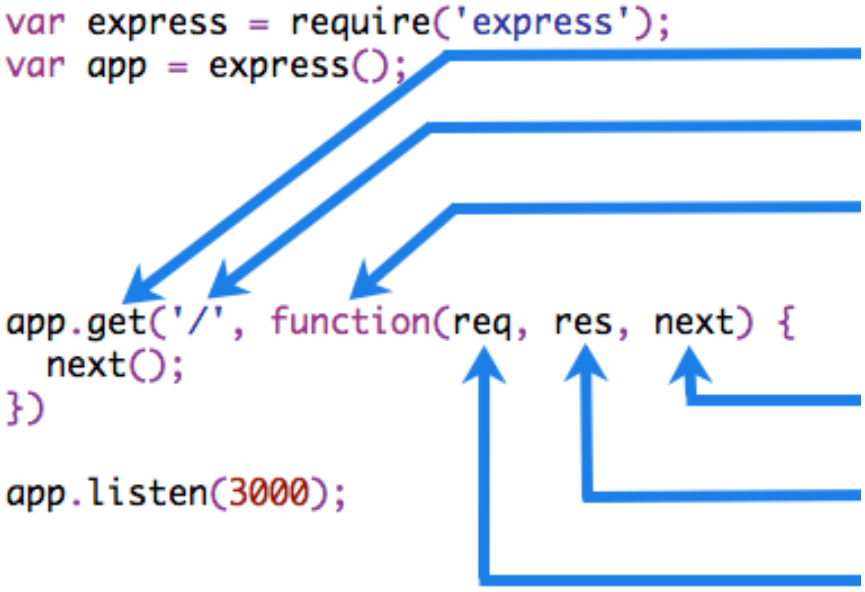
Application-level middleware [7]

You can bind **application-level** middleware to an instance of the `app` object by using:

- `app.use()` for any type of HTTP request;
- `app.<method>()` for a specific type of HTTP request:
  - `app.get()`
  - `app.post()`
  - `app.put()`
  - `app.delete()`

Express middleware functions

Application-level middleware [8]



HTTP method for which the middleware function applies.

Path (route) for which the middleware function applies.

The middleware function.

Callback argument to the middleware function, called `next` by convention.
HTTP response argument to the middleware function, called `res` by convention.
HTTP request argument to the middleware function, called `req` by convention.

Express middleware functions

Application-level middleware: ExpressJSMiddleware**Use**/server.js


```
...
// endpoint executed for all requests
app.use(function (req, res, next) {
        console.log(req.url);
        // there is no response sent, we mush call next()
        next();
});

// endpoint with a specific route
app.use('/home', function (req, res) {
        res.send('Welcome home!');
        // we sent a response, no need to call next()
});

// endpoint with a general route
app.use('/', function (req, res) {
        res.send('Hello World!');
        // we sent a response, no need to call next()
});
...
```

Express middleware functions

Application-level middleware: ExpressJSMiddleware**Method**/server.js


```
...
// enable your express app to serve static files located in ”public” directory
app.use(express.static('public'));
// enable your express app to read request body (for POST requests)
app.use(express.urlencoded({ extended: true }));
...
// endpoint executed for all requests
app.use(function (req, res, next) {
        console.log(req.url);
        console.log(req.method);
        console.log(req.query); // shows GET params
        console.log(req.body); // show POST params
        next();
});
...
```

Express middleware functions

Application-level middleware: ExpressJSMiddleware**Method**/server.js


```
...
// endpoint with a specific route
app.get('/data', function (req, res) {
        res.send('GET /data');
});

// endpoint with a specific route
app.post('/data', function (req, res) {
        res.send('POST /data');
});
...
```

Express middleware functions

Error-handling middleware [7]

You can 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)`, for instance:

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

Note: In addition to handling the error, it is a good practice to inform the user of the type or error, by sending the corresponding standard HTTP Status Code [9] in your HTTP response.

Express middleware functions

Error-handling example: ExpressJSMiddleware**Errors**/server.js, adapted from [10]


```
...
//import the fs module
const fs = require('fs');
...
app.get('/', (req, res, next) => {
        fs.readFile('./file.txt', { encoding: 'utf8' }, (err, data) => {
                if (err) {
                        next(err);
                } else {
                        res.send(''+ data);
                }
        })
})

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

Express middleware functions

Built-in middleware [7]

Express has many **built-in** middleware functions, for instance:

- `express.static`: serves static assets such as HTML files, images, etc.
- `express.json`: parses incoming requests with JSON payloads.
- `express.urlencoded`: parses incoming requests with URL-encoded payloads.
- and more[1].

[1] See the full list of built-in middleware functions available in express: `https://github.com/senchalabs/connect#middleware`

Express middleware functions

Built-in middleware [7] example: ExpressJSstatic


```
/ExpressJSstatic
├── /node_modules
├── package.json
├── /public
│   ├── index.html
│   └── resource.txt
└── server.js
```

ExpressJSstatic/server.js ...

```
5 // enable your express app to serve static files located in
      ”public” directory
6 app.use(express.static('public'));
```

...

Note: you have already seen this one!

Express middleware functions

Built-in middleware [7]

You can also use **third-party** middleware to add functionality to your Express app:

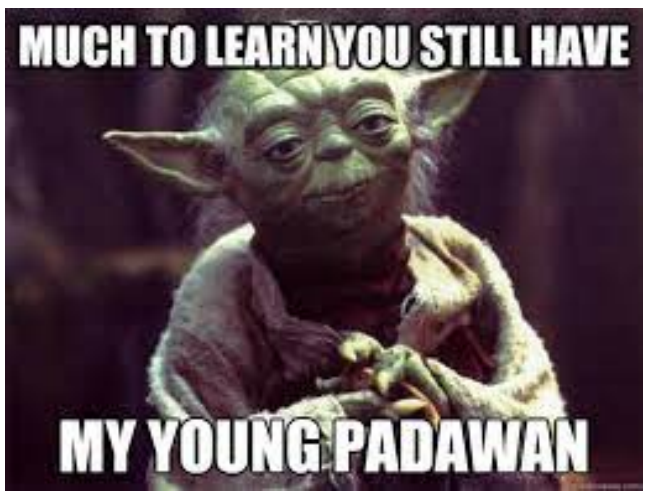
1. Install the Node.js module for the required functionality (using `npm`)
2. Load it in your app at the application level or at the router level (using `require()`)

For instance :

```
npm install sqlite3
```

and then in your code:

```
const sqlite3 = require('
    sqlite3');
```



Express middleware functions

An Express application can use the following types of middleware:

1. **Application-level** middleware
2. **Error-handling** middleware
3. **Built-in** middleware
4. **Third-party** middleware
5. **Router-level** middleware

Routing

Routes *vs.* endpoints

Routing [11]

"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)."

Routing in Express = defining middleware functions corresponding to each endpoint of your API.

Endpoint in Express = Request method + **Route path**

Routing
How to route

Routing in Express basically refers to associating a middleware function per API endpoint using:

```
app.method(path,handler)
```

where:

- (route) path: is the route path (URL or the resource requested) of the endpoint
- (route) handler: is the middleware function callback fired everytime there is a new request

Important: your callback function must use the `(req, res, next)` parameters and explicitly call `next()` if your endpoint is not sending a final response!

Routing

Route paths [13]

Express **route paths** can be:

- strings,
- string patterns,
- regular expressions.

Not an expert with regular expressions? Use the online Express Route Tester [12].

Routing
Route paths [13]

Examples of route paths based on **strings**[2]:

Route path	Target URL
`'/'`	root route `/`
`'/home'`	`/home`
`'/file.txt'`	`/file.txt`

[2] Again, you can test your route paths using Express Route Tester [12]

Routing

Route paths [13]

Examples of route paths based on **string patterns**[2]:

Route path	Target URL
`'/ab?cd'`	`/abd` and `/abcd`
`'/ab+cd'`	`/abcd`, `/abbcd`, `/abbbcd`, etc.
`'/ab*cd'`	`/abcd`, `/abxcd`, `/abRANDOMcd`, `/ab123cd`, etc.
`'/ab(cd)?e'`	`/abe` and `/abcde`

[2] Again, you can test your route paths using Express Route Tester [12]

Routing
Route parameters

Route parameters are named URL segments that are used to capture the values specified at their position in the URL. The name of route parameters must be made up of "word characters" ([A-Za-z0-9_]). Hyphen (-) and dot (.) characters are interpreted literally

Examples of route paths using **route parameters**[3]:

Route path	Target URL
`'/user/:user/movie/:movie'`	`/user/bob/movie/findingne` etc.
`/years/:from-:to`	`/years/1998-2022`, etc.

[3] Again, you can test your route paths using Express Route Tester [12]

Routing
Response methods

You can send a response to the client and terminate the request-response cycle by using a **response method** on the response object `res`:

Method	Target URL
`res.download()`	Prompt a file to be downloaded.
`res.end()`	End the response process.
`res.json()`	Send a JSON response.
`res.redirect()`	Redirect a request.
`res.send()`	Send a response of various types.
`res.sendFile()`	Send a file as an octet stream.
`res.sendStatus()`	Set the response status code and send its string representation as the response body.

Routing

And when there is no route?

Problem: How do I do if the requested path matches no route?

Important: If no response method is called from a route handler, the client request will be left hanging.

Routing

Serving a 404 error page example: ExpressJS404/server.js

...

```
13 // basic root route
14 app.get('/', function(req, res){
15         res.send('Hello world!');
16   });
17
18 // the 404 route (always keep this as the last route)
19 app.get('*', function(req, res){
20         res.status(404);
21         //serve a static html file (can be done using express.static)
22         res.sendFile("404.html", {root: "."});
23 });
```

...

Conclusion

- Key takeaways:
  - ✓ ExpressJS is a powerful (and popular) back-end framework ...
  - ✓ .. built on top of NodeJS, that helps at building complete web APIs.
  - ✓ In ExpressJS, you define middleware (callback) functions ...
  - ✓ ... for each endpoint of your API.
  - ✓ Routing enables your Express app to handle at the same time ...
  - ✓ ... the requested URL and the request method.
  - ✓ Using NodeJS modules in your Express app enables you to ...
  - ✓ ... decompose your code, routing and middleware in separate files ...
  - ✓ ... in order to create modular and reusable applications.

References and further reading/watching I

[1] Most popular backend frameworks 2012-2022. URL https://statisticsanddata.org/data/most-popular-backend-frameworks-2012-2022/.

[2] Go full-stack with node.js, express, and mongodb. URL https://openclassrooms.com/fr/courses/5614116-go-full-stack-with-node-js-express-and-mo

[3] Expressjs starter, . URL https://expressjs.com/en/starter/hello-world.html.

[4] Victoria Kirst. *CS193X Web Programming Fundamentals*, 2017. URL https://web.stanford.edu/class/archive/cs/cs193x/cs193x.1176/.

[5] Web api wikipedia. URL https://en.wikipedia.org/wiki/Web_API.

[6] Endpoint – what is an api endpoint? URL https://rapidapi.com/blog/api-glossary/endpoint/.

References and further reading/watching II

[7] Expressjs middleware, . URL https://expressjs.com/en/guide/using-middleware.html.

[8] Writing expressjs middleware, . URL https://expressjs.com/en/guide/writing-middleware.html.

[9] Http status codes. URL https://www.restapitutorial.com/httpstatuscodes.html.

[10] Express error handling, . URL https://expressjs.com/en/guide/error-handling.html.

[11] Express basic routing, . URL https://expressjs.com/en/starter/basic-routing.html.

[12] Express route tester, . URL http://forbeslindesay.github.io/express-route-tester/.

[13] Express routing, . URL https://expressjs.com/en/guide/routing.html.