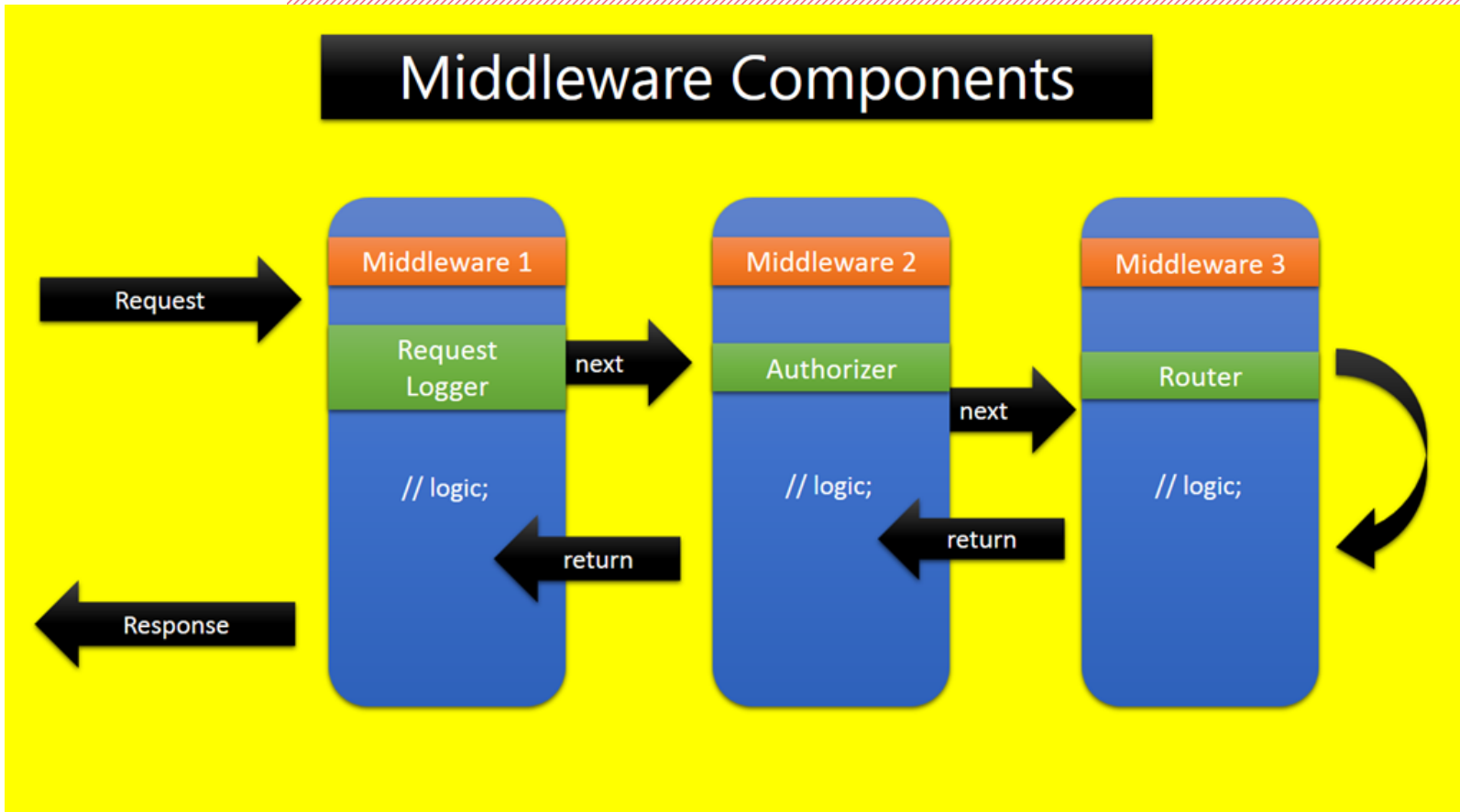
Node.js

Ch 5: Middleware functions

hogeschool vives



D. HOSTENS & M. DIMA

What are middleware functions?

- A middleware function is a function that takes a request object and either sends a response to the client or passes control to another middleware function.
- We have already seen two examples:
- **Example 1:** Every route handler function (Express API):

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

- It receives a *request* from the client and sends a *response* back. This means it ends the *request-response* cycle.

— Example 2 (Ch. 4 Express API)

```
app.use(express.json());
```

- The `express.json()` method returns a middleware function.
- The task of this middleware function is to read the request, and if a JSON object is found in the body of the request, it will be parsed. Then, the `req.body` property is set.
- Only after this can you read the JSON from `req.body`.

— Another example: Global middleware function

- Middleware is a function/program/… that runs in the time between when the server receives a request and when it sends a response back to the client (e.g., a route handler function).
- We are creating a new app (Express boilerplate code).

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

app.get('/', (req, res) => {
    res.send('Homepage')
});

app.listen(3000);
```

—

- We are creating our own middleware function logger().

```
function logger(req, res, next){
    console.log('log');
}
```

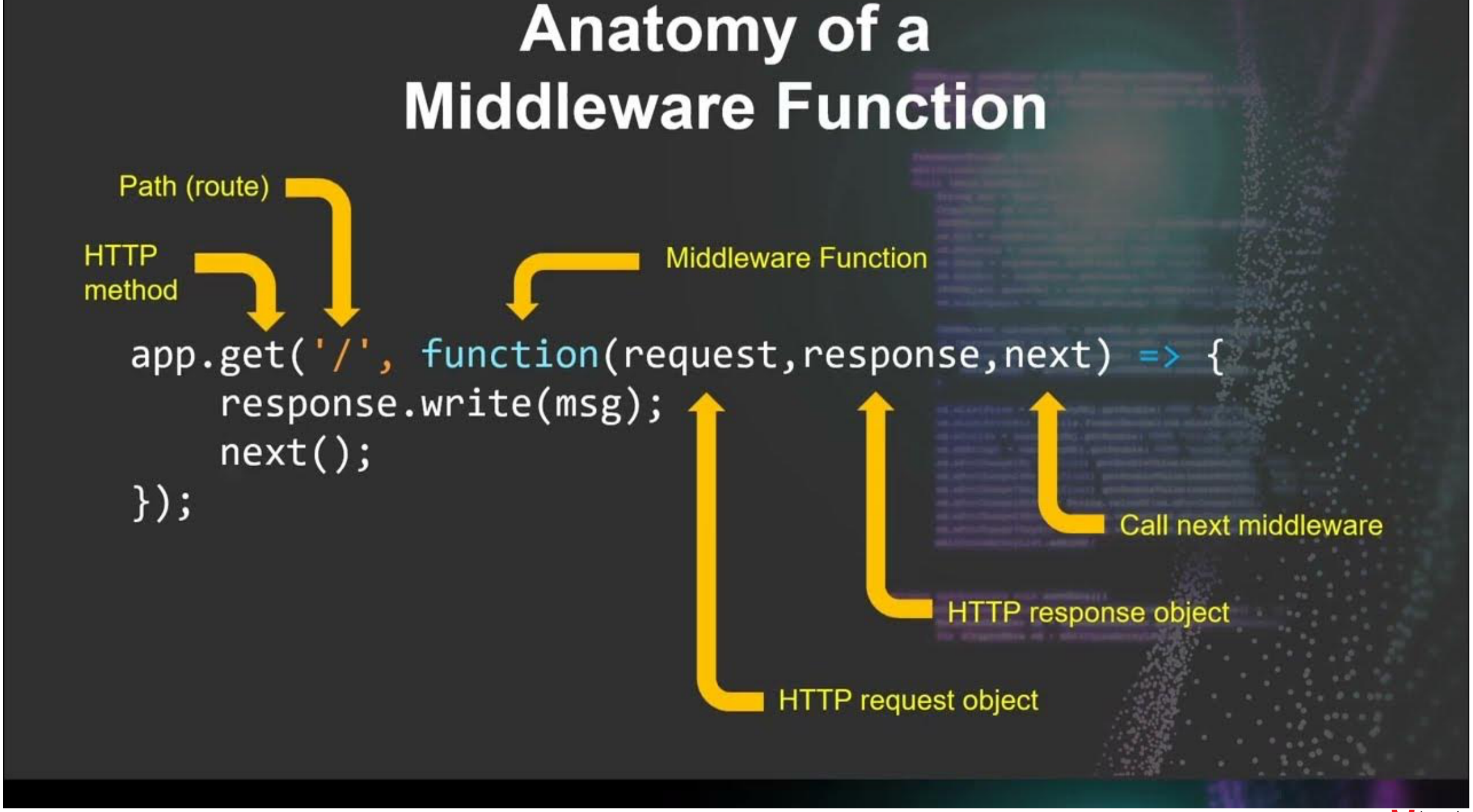
- The function takes three parameters: `request`, `response`, and `next`.
- `next` is a function that simply gets called, allowing the next middleware function to be executed.

```
function logger(req, res, next){
    console.log('log');
    next();
}
```

- Route handlers can also accept next as a parameter, but since they are often the final step, it is usually unnecessary.

```
app.get('/', (req, res, next) => {
    res.send('Homepage')
    next();
});
```

Anatomy of a Middleware Function



— How do we use the middleware logger?

- We add the following line at the top of our file:

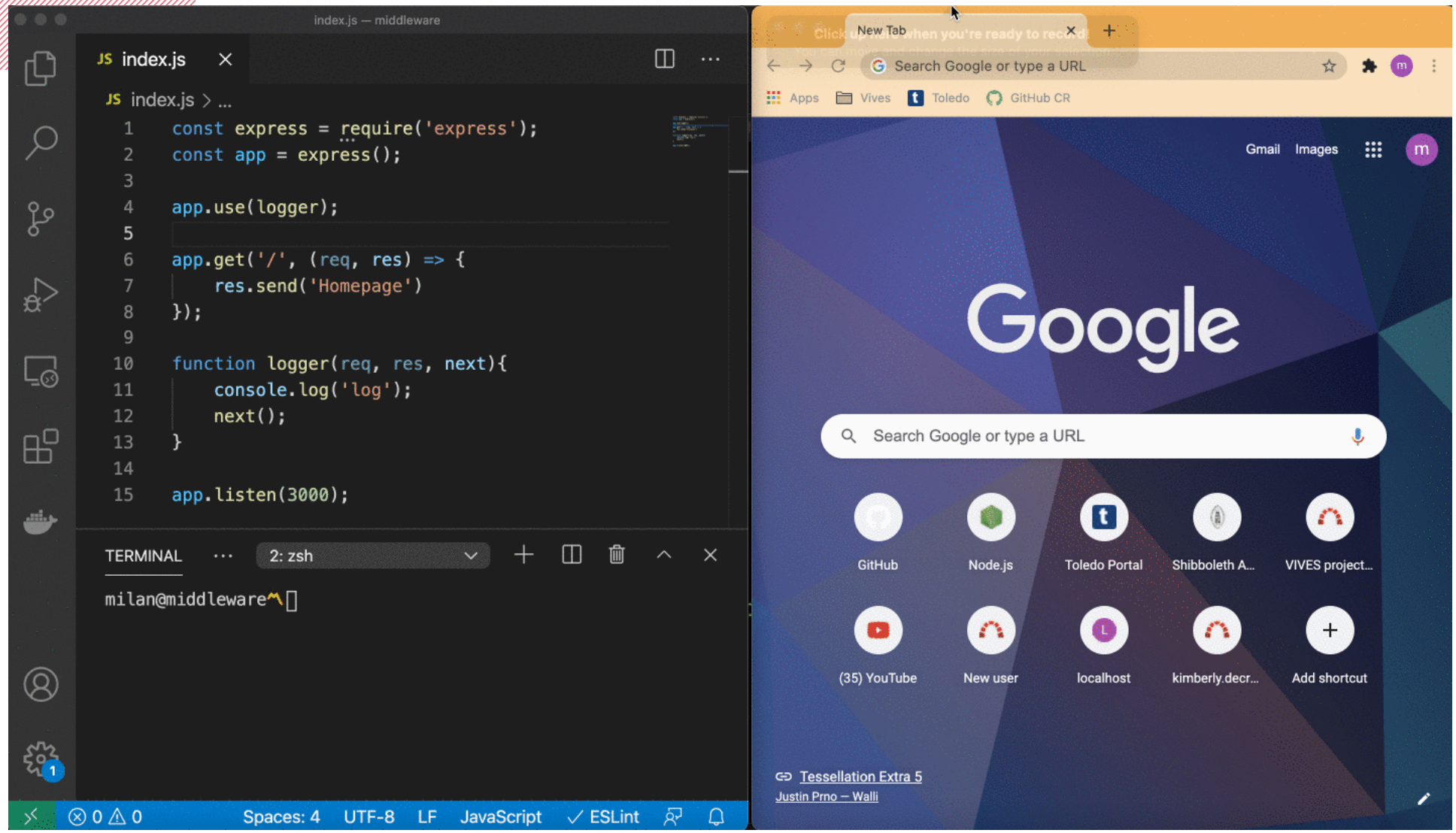
```
app.use(logger);
```

- We run our file: `node index.js` (or nodemon ...)
- We open the browser and go to: http://localhost:3000/

```
milan@middleware〽 node index.js
log
```

- We see that “log” appears in the terminal.
- This means that the logger function is executed first.
- The `next()` function ensures that `app.get()` is also executed afterward.
- Demo screencast: see toledo

— see gif Toledo



— Order

```
const express = require('express');
const app = express();
app.use(logger);
app.get('/', (req, res) => {
    res.send('Homepage')
    console.log('The GET request is handled HERE.')
});
function logger(req, res, next){
    console.log('log');
    next();
}
app.listen(3000);
```

- milan@middleware〽node index.js

```
log
The GET request is handled HERE.
```

— Position in the code is important!

- If we place `app.use()` afther `app.get()` it will not be called because `app.get()` does not invoke the `next()` function!

```
...
app.get('/', (req, res) => {
    res.send('Homepage')
    console.log('The GET request is handled HERE.')
});
app.use(logger);                <- !!!
function logger(req, res, next){
    console.log('log');
    next();
}...
```

```
milan@middleware〽node index.js
The GET request is handled HERE.
```

— Single action middleware

- Is placed in another function, e.g., `app.get()`
- `app.get()` accepts a URL parameter and a list of middleware functions.
- So far, we have only passed req and res.
- We create a new function `auth()`:

```
function auth(req, res, next){
    console.log('auth');
    next();
}
```

- We add it to the middleware list of `app.get()`:

```
app.get('/', auth, (req, res) => {...
```

— the complete file

```
const express = require('express');
const app = express();
app.use(logger);
app.get('/', auth, (req, res) => {
    res.send('Homepage')
    console.log('The GET request is handled HERE.')
});
function logger(req, res, next){
    console.log('log');
    next();
}
function auth(req, res, next){
    console.log('auth');
    next();
}
app.listen(3000);
```

— Run

- In the Terminal

```
milan@middleware〽node index.js
```

- In the browser
- http://localhost:3000/
- Output

```
log
auth
The GET request is handled HERE.
```

— Middleware has access to req.body

- We change our middleware function `auth()`

```
function auth(req, res, next){
    if (req.query.admin === 'true') next();
    else res.send('No Auth!');
}
```

- In the browser: http://localhost:3000/
- Browser Output: `No auth!`
- In the browser: http://localhost:3000/?admin=true
- Browser output: `Homepage`

— Pass data

- Suppose we want to inform `app.get()` that the user is an admin. We cannot do this via `next()`, like `next(admin)`.
- How do we solve this?
- By adding variables to our request body.
- In the `auth()` function:

```
req.admin = true;
```

- In the `app.get()` function, we now have access to the req.admin variable:

```
console.log(`User is admin = ${req.admin}`);
```

```
milan@middleware node index.js
log
The GET request is handled HERE.
User is admin = true
```

— Warning!

- `next()` does not equal to `return`!
- next() invokes the next middleware but also returns to execute the rest of the function.

```
function auth(req, res, next){
    if (req.query.admin === 'true')
        req.admin = true;
        next();
    console.log('This will also be executed!');
}
```

- We refreshen de browser

```
milan@middleware〽node index.js
log
The GET request is handled HERE.
User is admin = true
This will also be executed!
```

— Complete index.js

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

app.use(logger);

app.get('/', auth, (req, res) => {
    res.send('Homepage')
    console.log('The GET request is handled HERE')
    console.log(`User is admin = ${req.admin}`);
});

function logger(req, res, next){
    console.log('log');
    next();
}

function auth(req, res, next){
    if (req.query.admin === 'true')
        req.admin = true;
        next();
    console.log('This will also be executed!');
}

app.listen(3000);
```

Request processing pipeline

- request --> [next m]--> [next m]--> [next m]--> response
- next m = next middleware function
- In our app from Ch. 4
- `request` --> [`json()`]--> [`route()`]--> `response`

Good practice

- It's better to extract the middleware functions to separate modules and not keep them inside the index.js (or app.js)

— We use the code frrom Ch. 4 as starter

- You can also pull it from Github:

https://github.com/MilanVives/Node_les4

```
git clone https://github.com/MilanVives/
Node_les4
npm install
```

— We continue building our Express API from Ch. 4

- We create a new file `logger.js`

```
function log(req, res, next) {
console.log('logging...');
next();
}
module.exports = log;
```

- `index.js`

```
const logger = require('./logger');
app.use(logger);
```

- In the browser: http://localhost:3000/
- Terminal output:

```
milan@les5 nodemon index.js
...
[nodemon] starting `node index.js`
Listening on port 3000...
logging...
```

Built-in middleware - `json()` and `urlencoded()`

- Express already has a few built-in middleware functions
- We already know one: `express.json()`
- Another one is `express.urlencoded()`
- Sometimes the data will be passed in the URL e.g. in http forms (older technology). `key=value&key=value`
- `urlencoded()` takes this data and appends it to the `req.body`
- we add an extra line in our `index.js`:

```
app.use(express.urlencoded());
```

— Built-in middleware - `express.urlencoded()`

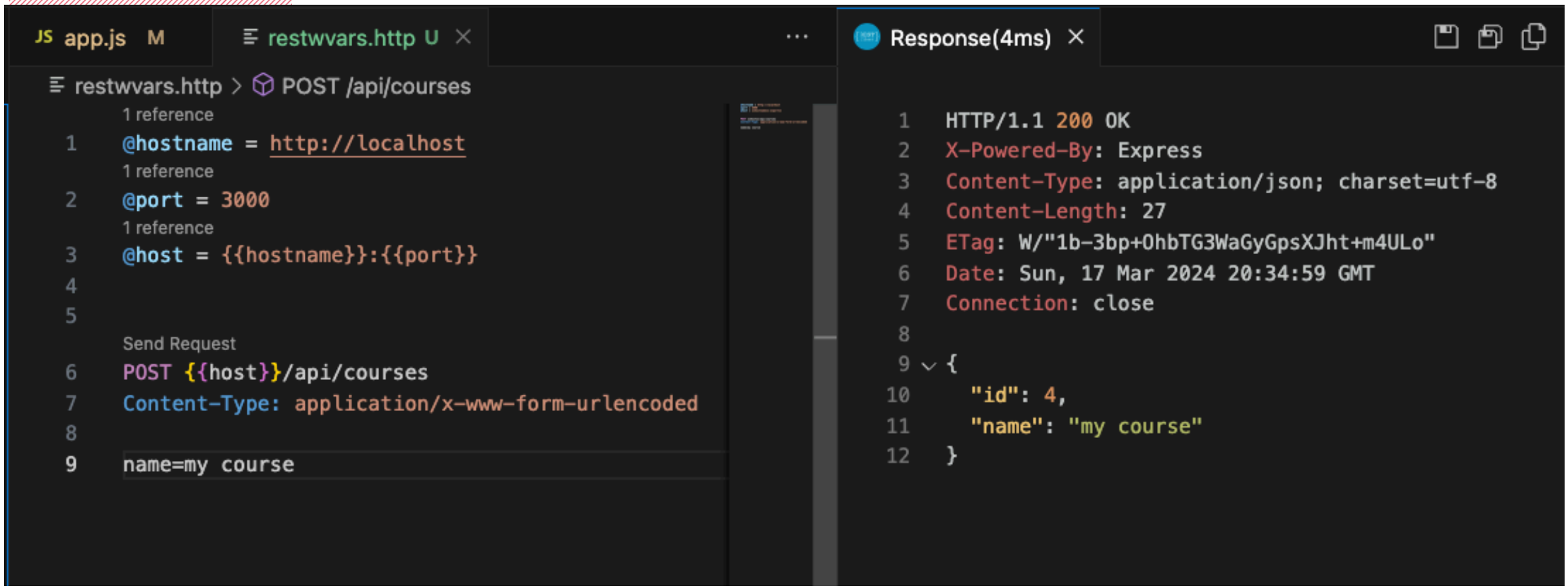
- In Postman send a POST request to:
  http://localhost:3000/api/courses
- Body > x-www-form-urlencoded
  Inside KEY we put `name and` VALUE = `"my course"`
- Clock on End
- Postman response

```
{
  "id": 4,
  "name": "my course"
}
```

- Course was created!
- Terminal will give you a warning: body-parser deprecated undefined extended: provide extended option index.js:8:17

```
app.use(express.urlencoded({extended: true}));
```

With Rest Client



Built-in middleware - `express.static()`

- Servs static files
- We create a folder '`public`' in our project
- Inside that folder a file `readme.txt` with `'This is a readme file'` as content.
- We add to our `index.js`:

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

- in the browser
- http://localhost:3000/readme.txt
- Browser output:

```
This is a readme file!
```

- Please note: **`public`** will not be appended to the URL!

— Third party middleware - helmet

- Go to: https://expressjs.com/en/resources/middleware.html
- Those are all 3rd party middleware functions
- Only use if necessary. They slow down the response
- An example: helmet Helps secure your apps by setting various HTTP headers.
- We install helmet: `npm i helmet`
- In our index.js:

```
const helmet = require('helmet');
...
app.use(helmet());
```

— Third party middleware - morgan

morgan HTTP request logger. express.logger
https://expressjs.com/en/resources/middleware/morgan.html

- We install it: `npm i morgan`
- In the `index.js` file:

```
const morgan = require('morgan')
...
app.use(morgan('tiny'));
```

- tiny is a format, see documentation.
- We run the app
- In the browser: http://localhost:3000/api/courses
- Output terminal:

```
milan@les5 nodemon index.js
[nodemon] 2.0.7
...
[nodemon] starting `node index.js`
GET /api/courses 200 79 - 2.059 ms
```

— Environments

- An additional middleware is another stop in your request pipeline, which impacts speed.
- It can be useful in certain situations
- e.g., only during development but not in production.
- Environments: Development - Test - Production
- In complex applications, different environments are set up.
- Some features work only in a specific environment,
- e.g., logging, test frameworks, etc.
- We use `process.env.NODE_ENV` for this.

— process.env.NODE_ENV

- Is a default variable which keeps track of the current environment we work in. Can be `undefined` if not set

```
console.log(`NODE_ENV: ${process.env.NODE_ENV}`);
```

- The app object also has a way to read the environment
- If NODE_ENV is not set it will by default return development

```
console.log(`app: ${app.get('env')}`);
```

- In the terminal

```
[nodemon] starting `node index.js`
NODE_ENV: undefined
app: development
Listening on port 3000...
```

— Environment dependent logging

- We only want logging inside the dev environment:

```
if (app.get('env') === 'development'){
    app.use(morgan('tiny'));
    console.log('Morgan enabled');
}
```

```
[nodemon] starting `node index.js`
Morgan enabled
Listening on port 3000...
```

- we change the env variable to production

```
^C%
milan@les5 export NODE_ENV=production
milan@les5 nodemon index.js
...
[nodemon] starting `node index.js`
Listening on port 3000...
```

— Keep Configuration settings

- Most populaire package is `rc`
- Less populair but user friendlier is: `config`
- We install config: `npm i config`
- Usage: you create a folder named `config`
- Inside that folder you create several configuration files: `default`, `production`, `development`....
- The content of those files is JSON and extension also e.g. `default.json`

```
{
    "name": "My Express App"
}
```

- development.json

```
{
    "name": "My Express App - Development",
    "mail": {
        "host": "dev-mail-server"
    }
}
```

- production.json

```
{
    "name": "My Express App - Production",
    "mail": {
        "host": "prod-mail-server"
    }
}
```

- In index.js

```
const config = require('config');
...
console.log('Application Name:', config.get('name'));
console.log('Application Name:', config.get('mail.host'));
```

```
[nodemon] starting `node index.js`
Application Name: My Express App - Development
Application Name: dev-mail-server
Morgan enabled
Listening on port 3000...
```

- We change the environment to development

```
milan@les5~export NODE_ENV=production
milan@les5~nodemon index.js
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Application Name: My Express App - Production
Application Name: prod-mail-server
Listening on port 3000...
```

- SECURITY RISC: Don't keep credentials inside config files!
- Save them in environment variables. (persistent if necessary)

— Passwords policy

- best practice is to save secrets/passwords in environment variables
- also see slides on environment variables in the folder Extra on Toledo
- we want to save a password for a mail server
- we create a new environment variable:

```
milan@les5 export les5_password=1234
```

Linux/macOs: export les5_password=1234
Windows cmd: set les5_password=1234
Windows PowerShell: $Env:les5_password = 1234

- we create a new file in the config folder
- `custom-environment-variables.json`
- Please note the name should be identical!

```
{
    "mail": {
        "password": "les5_password"
    }
}
```

— in index.js

- we add an additional console.log()

```
console.log('Application Name:', config.get('mail.password'));
```

- config will search in a few files
- will pick up the content of the environment variable
- upon commit the value of the environment variable will not be committed and pushed to a remote repository

```
[nodemon] starting `node index.js`
Application Name: My Express App - Development
Mail server: dev-mail-server
Password: 1234
1234
Morgan enabled
Listening on port 3000...
```

— Templating Engines

- Now the endpoints will only return JSON
- Sometimes you need to return dynamic HTML
- Templating engines generate html
- There are several available for Express apps:
- Pug (used to be Jade)
- Mustache
- EJS
- ...
- They all use a different syntax

— Pug

- We install Pug: `npm i pug`
- in `index.js` we set a view-engine for the app

```
app.set('view engine', 'pug');
```

- By doing this Express will internally load `pug`.
- No need for require!
- Templates will be saved in views.
- Exceptions on this: `app.set('views', './path');`
- We create a folder: `views`
- And a new file in `views: index.pug`

— `index.pug` syntax

```
html
    head
        title= title
    body
        h1= message
```

- in `index.js` we update the `app.get('/',...`

```
app.get('/', (req, res) => {
    res.render('index', { title: 'My Express App',
message: 'Hello'});
});
```

- In the browser: http://localhost:3000/
- Output browser:

Hello

— Project structure Express App

- Middleware in a separate folder: `middleware`
- Routes (endpoints) in a separate folder `routes` by URL
- All `/api/courses/...` endpoints grouped together is a file `courses.js`

- We begin with the routes
- We create a folder `routes`
- Inside the `routes` folder a file `courses.js`
- We move all courses endpoints and the `courses` array
- `const app = express()` will be replaced by
- `const router = express.Router();`

— in index.js

- we move all endpoints, the array and the `validateCourse` function to `courses.js`
- we importert the courses:

```
const courses = require('./routes/courses');
const home = require('./routes/home')
```

- we let Express know that all routes beginning with `/api/courses` should use the `courses` module

```
app.use('/api/courses', courses);
app.use('/', home);
```

- And all routes to the `root (/)` the `home` router

— courses.js

```
const express = require('express');
const router = express.Router();
const courses = [
    {id: 1, name: 'course1'},
    {id: 2, name: 'course2'},
    {id: 3, name: 'course3'},
    ];
router.get('/', (req, res) => {
        res.send(courses);
    });
router.post('/', (req,res) => {
    const result = validateCourse(req.body);
    if(result.error){
        res.status(400).send(result.error.details[0].message);
        return;
        }
    const course = {
        id: courses.length + 1,
        name: req.body.name
    };
    courses.push(course);
    res.send(course);
    });
router.get('/:id', (req, res) => {
    const course = courses.find(c => c.id === parseInt(req.params.id));
    if(!course) return res.status(404).send('the course with the given id was not
found');
    res.send(course);
    });
```

```
router.put('/:id', (req, res) => {
    const course = courses.find(c => c.id === parseInt(req.params.id));
    if(!course) return res.status(404).send('the course with the given id was not found');
    const result = validateCourse(req.body);
    if(result.error){
        res.status(400).send(result.error.details[0].message);
        return;
    }
    course.name = req.body.name;
    res.send(course);
    });
router.delete('/:id', (req, res) => {
    const course = courses.find(c => c.id === parseInt(req.params.id));
    if(!course) return res.status(404).send('the course with the given id was not found');
    const index = courses.indexOf(course);
    courses.splice(index, 1);
    res.send(course);
    });
function validateCourse(course){
    const schema = Joi.object({
        name: Joi.string().min(3).required()
    });
    return schema.validate(course);
    }
module.exports = router;
```

— home.js

```
const express = require('express');
const router = express.Router();

router.get('/', (req, res) => {
    res.render('index', { title: 'My Express
App', message: 'Hello'});
});

module.exports = router;
```

Extract middleware to different folder

- We create a folder: `middleware`
- We move `logger.js` to that folder
- We edit the links in the imports (automatically?)

```
config
  custom-environment-variables.json
  default.json
  development.json
  production.json
index.js
middleware
 logger.js
node_modules
package-lock.json
package.json
public
  readme.txt
routes
  courses.js
  home.js
views
  index.pug
```

— Lab

- See Github Classroom Ch. 5
- Link: see Toledo