UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

WEB TECHNOLOGIES — LECTURE 13

WEB FRAMEWORKS: EXPRESS

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

https://docenti.unina.it/luigiliberolucio.starace

https://luistar.github.io

PREVIOUSLY, ON WEB TECHNOLOGIES

We developed a few web apps using server-side programming

- We implemented our To-do list app using CGI and Bash (💀)
- We implemented our To-do list app using PHP 8.2
- We implemented our To-do list app using Node.js

There were a few **pain points**

- **Lots of tedious stuff** (e.g.: parse cookies, read and parse request bodies, …)
- **No out-of-the-box session management** (except for PHP)
- **No out-of-the-box middleware support** (e.g.: authentication)
- **No out-of-the-box templating**

COMMON TASKS IN WEB DEVELOPMENT

- Managing Routing
- Rendering templates
- Managing Web Sessions
- Managing Authentication/Authorization

We do not want to re-invent the wheel in every project, right?

WEB FRAMEWORKS

- A **framework** is a pre-defined set of software components, tools, and best practices that serve as a **foundation** for developing software
- **Web frameworks** are specifically designed to simplify and streamline the development of web applications
- They provide a (structured) way to build and organize web applications, reducing the need for developers to start from scratch and re-invent the wheel
- Frameworks and Software Libraries are different things!

SOFTWARE LIBRARY

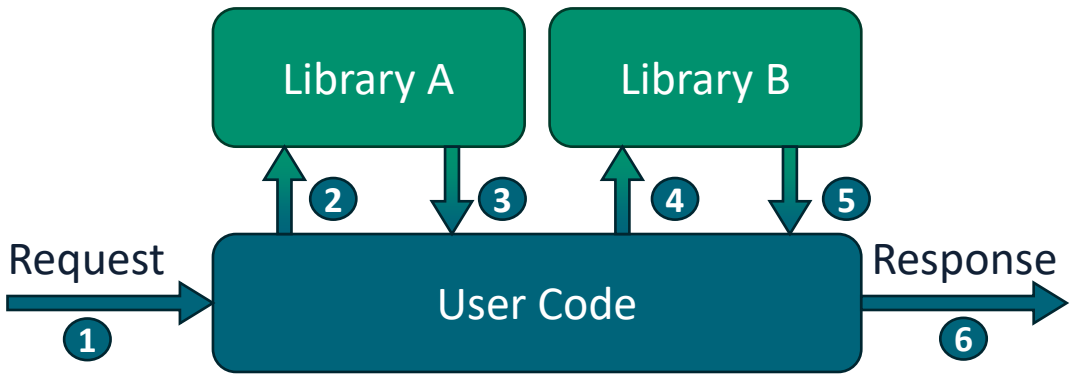
- Essentially a set of functions we can call to get some job done
- Each call does some work and then returns control to the caller
- User code (e.g.: our **main** method) is in control of the execution flow

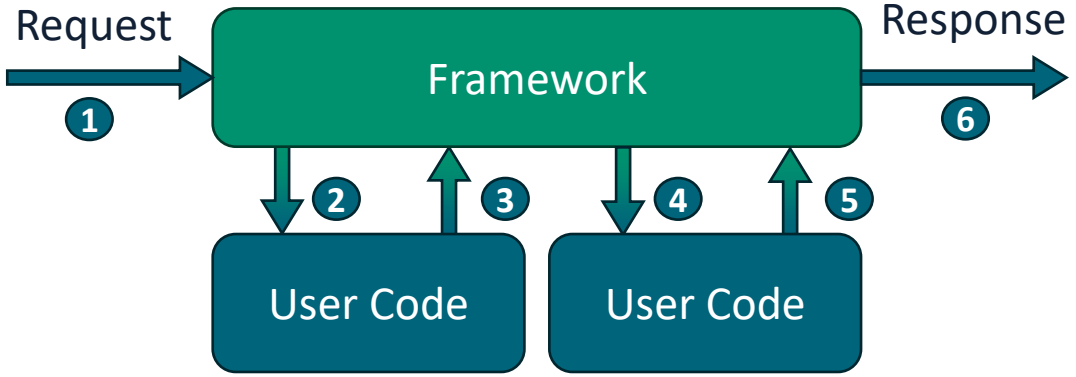
FRAMEWORK

- Embody some abstract design, with more **behaviour** built-in
- User-written code can be **plugged-in** into the framework
- The framework is in control of the execution flow and calls user code when appropriate

INVERSION OF CONTROL (IoC) PRINCIPLE

- **IoC** is a defining feature of frameworks
- The control flow is **inversed** from traditional imperative programming
  - The responsibility of managing the control flow is shifted from the developer to a framework
- A.k.a. Hollywood's Law: "*Don't call us, we'll call you*".





KEY COMPONENTS OF WEB FRAMEWORKS

Web frameworks typically include:

- **Core functionalities:**
  - Routing; Request parsing; Input validation; Cookie management; Sessions…
- **Template Engine:**
  - Helps in rendering dynamic content, separating code logic from the presentation layer.
- **ORM (Object-Relational Mapping) Mechanism:**
  - Simplifies database interactions by allowing developers to work with objects instead of SQL queries.

OPINIONATED FRAMEWORKS

- Frameworks can be more or less **opinionated**
- A (strongly) **opinionated** framework comes with a rigid set of pre-defined conventions, best practices, and decisions made by the framework's creators. It **enforces** a specific way of doing things, and leaves less wiggle room to devs.
- Unopinionated frameworks do not enforce a specific way of doing things

Keep in mind, this framework has really strong opinions…

Artwork generated using DALL-E 3

OPINIONATED FRAMEWORKS

Pros

- Ensure higher levels of consistency across different projects
- Can speed up development, reducing decision fatigue

Cons

- May have a steeper learning curve
- May not be flexible enough for a certain project

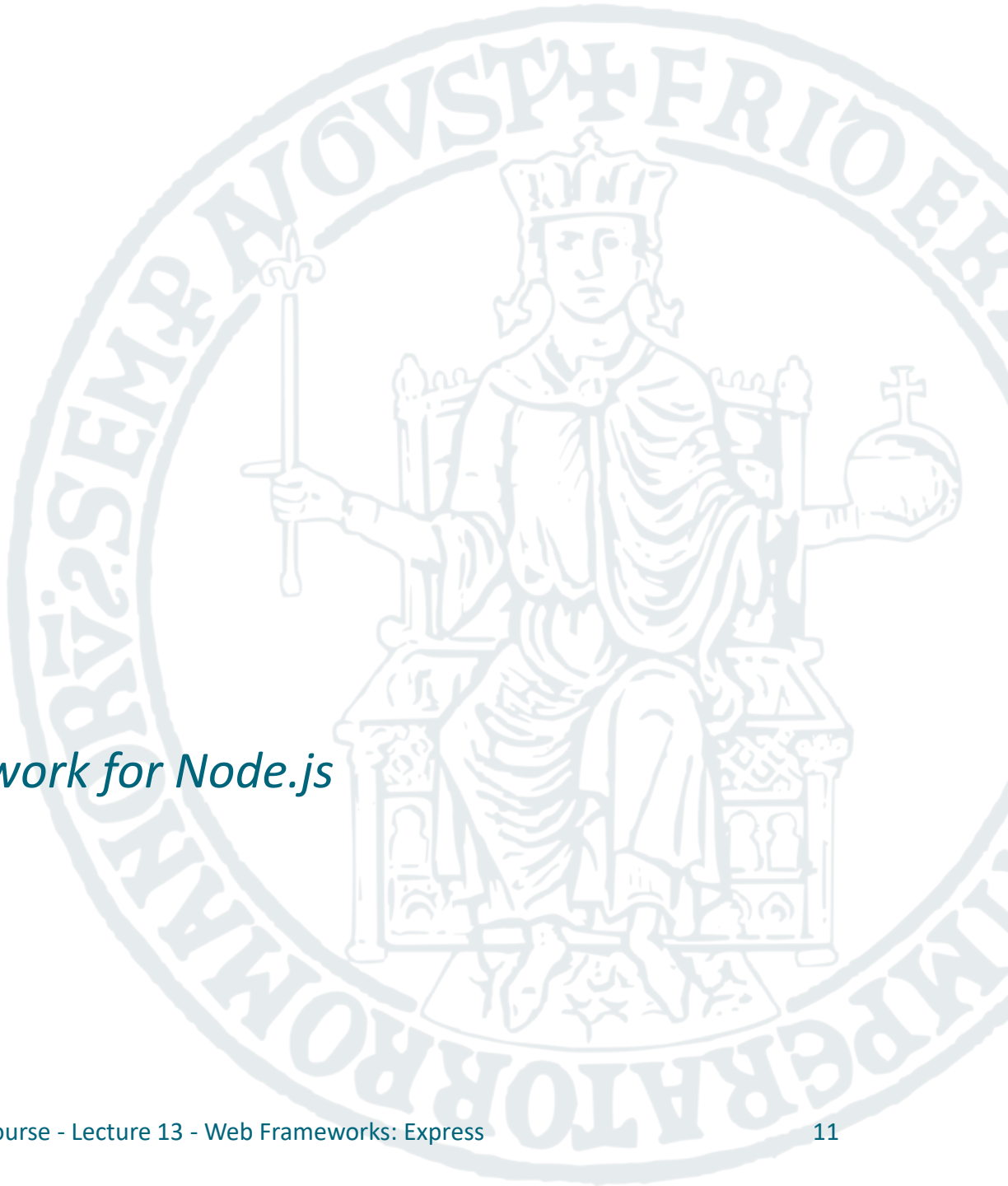
WEB FRAMEWORKS (SERVER-SIDE)

- Many web frameworks to choose from (e.g.: see Wikipedia)

One of the above is not a real framework, but a Pokemon. Can you tell which one?

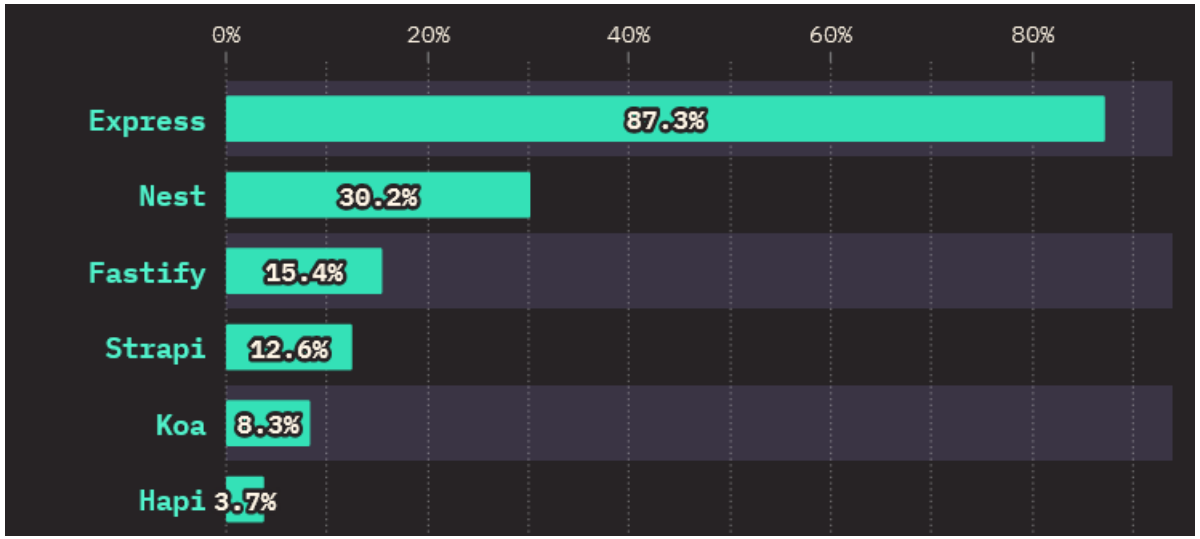
EXPRESS

Fast, unopinionated, minimalist web framework for Node.js



THE EXPRESS FRAMEWORK

- In the web technologies course, we'll see Express as an example
- *Fast, unopinionated, minimalist web framework for Node.js*
- By far, the most popular Node.js backend framework



State of JavaScript 2022 Report. Source here.

HELLO EXPRESS

```
@luigi ➜ D/O/T/W/2/e/express-hello-world$ npm install express
```

```
import express from "express";

const app = express(); // creates an express application
const PORT = 3000;

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

app.listen(PORT);
```

EXPRESS: ROUTING

- Routing refers to the way the framework determines how to handle a request to a given endpoint
- A **route** is defined using appropriate methods on the **app** object
- Typically using statements of the form

```
app.<METHOD>(<PATH>, callback)
```

- Where <METHOD> and <PATH> identify an endpoint, and callback is a function to execute to handle requests to that endpoint

```
app.get("/hello", (req, res) => {
  res.send("Hello");
});
```

```
app.post("/hello", (req, res) => {
  res.send("Hello Post");
});
```

EXPRESS: ROUTE PATHS

Route paths can be strings, string patterns, or full regular expressions

- In case of multiple matches, the first matched route applies

```
app.get("/webtech", (req, res) => {
  res.send("Web Technologies!");
});

//matches /webtech, /web-tech, /web-technologies, /webtechnologies, etc...
app.get("/web(-)?tech*", (req, res) => {
  res.send("Web Technologies Pattern!");
});

//matches /webtech, /wirelesstech, /wtech, /whatever-tech, etc...
app.get(/^\/w.*tech$/, (req, res) => {
  res.send("Web Technologies Regex!");
});
```

EXPRESS: ROUTE PARAMETERS

- Express routes can also capture named URL segments called parameters, declared in the PATH by using **«:paramName»**
- The captured parameters are made available in the **req.params** obj.

```
//if a get request is made to /student/42/exams/TecWeb
app.get("/student/:student_id/exams/:exam_name", (req, res) => {
  res.send(req.params); //{"student_id":"42","exam_name":"TecWeb"}
});

//finer control over admissible param values can be defined using regex in
parentheses. The below route requires :student_id to be a series of digits.
It matches /student/42/exams/TecWeb, but not /student/bob/exams/TecWeb.
app.get("/student/:student_id([0-9]+)/exams/:exam_name", (req, res) => {
  res.send(req.params); //{"student_id":"42","exam_name":"TecWeb"}
});
```

EXPRESS: MULTIPLE ROUTE HANDLERS

- It is also possible to specify multiple handler functions for a route
- These functions behave like **middleware** (we'll see soon enough!)
- **next** is a callback to the next handler function to run

```
let f1 = function(req, res, next) {console.log("f1"); next();}
let f2 = function(req, res, next) {console.log("f2"); next();}
let f3 = function(req, res, next) {console.log("f3"); next();}

app.get("/handlers", [f1, f2, f3], (req, res) => {
  res.send("Done with all handlers");
})
```

EXPRESS: WORKING WITH ROUTES

- We can create chainable routes using **app.route()**
- This way, the path is specified in a **single** location (**less redundancy**), and method-specific handlers are chained.

```
app.route('/book')
  .get((req, res) => {
    res.send('Get a random book')
  })
  .post((req, res) => {
    res.send('Add a book')
  })
  .delete((req, res) => {
    res.send('Delete a book')
  });
```

EXPRESS: MODULAR ROUTERS

- We can also create a modular router, which can be loaded as needed

```
//router.js
import express from "express";

export const router = express.Router();

router.get("/echo/:value", (req, res) => {
  res.send(req.params.value);
});
```

```
import { router as echoRouter } from "./router.js";
const app = express();
//other routes can be defined here as done earlier...
app.use("/custom", echoRouter); //load echoRouter (optionally in a certain path)
//requests to /custom/echo/:value will be handled by the router
app.listen(3000);
```

EXPRESS: RESPONSE METHODS

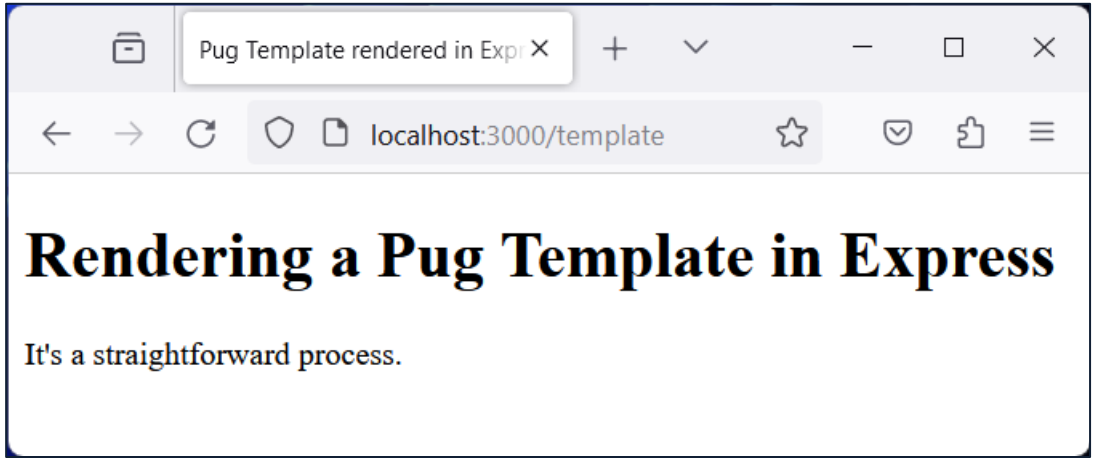
Appropriate methods on the response object (res) should be called to send a response to the client, or the request will remain hanging.

Method	Description
res.download()	Prompt a file to be downloaded.
res.end()	End the response process.
res.json()	Send a JSON response.
res.jsonp()	Send a JSON response with JSONP support.
res.redirect()	Redirect a request.
res.render()	Render a view template.
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.

EXPRESS: RENDERING A TEMPLATE

```
const app = express();
app.set("view engine", "pug");

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

app.listen(3000);
```

```
//- file: /views/template.pug
doctype html
html
  head
    title Pug Template rendered in Express
  body
    h1 Rendering a Pug Template in Express
    p It's a straightforward process.
```

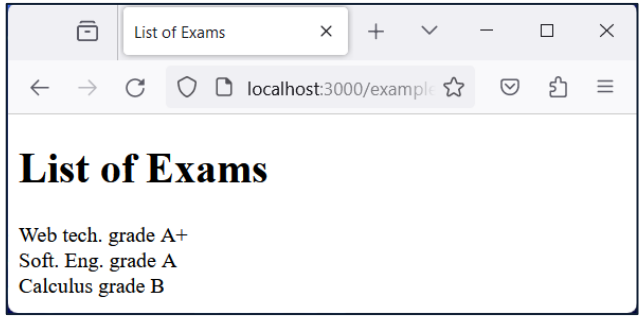


EXPRESS: RENDERING A TEMPLATE

- Moreover, data set in the **res.locals** object is available to all views
- Data set in the **res.locals** is available to views rendering **res**

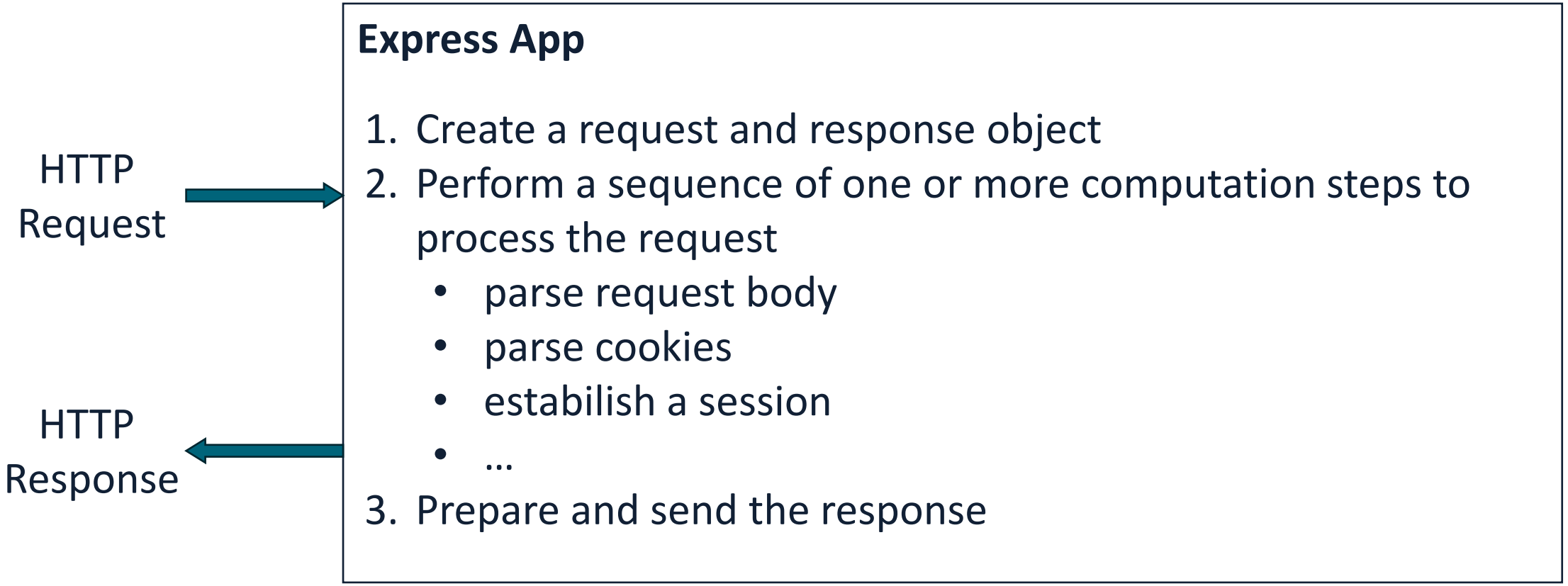
```
app.get("/example", (req, res) => {
  res.locals.items = [
    {name: "Web tech.", grade: "A+"},
    {name: "Soft. Eng.", grade: "A"},
    {name: "Calculus", grade: "B"},
  ]
  res.render("example");
});
```

```
//- file: /views/example.pug
doctype html
html
  head
    title List of Exams
  body
    h1 List of Exams
    each exam in items
      li #{exam.name} grade #{exam.grade}
```



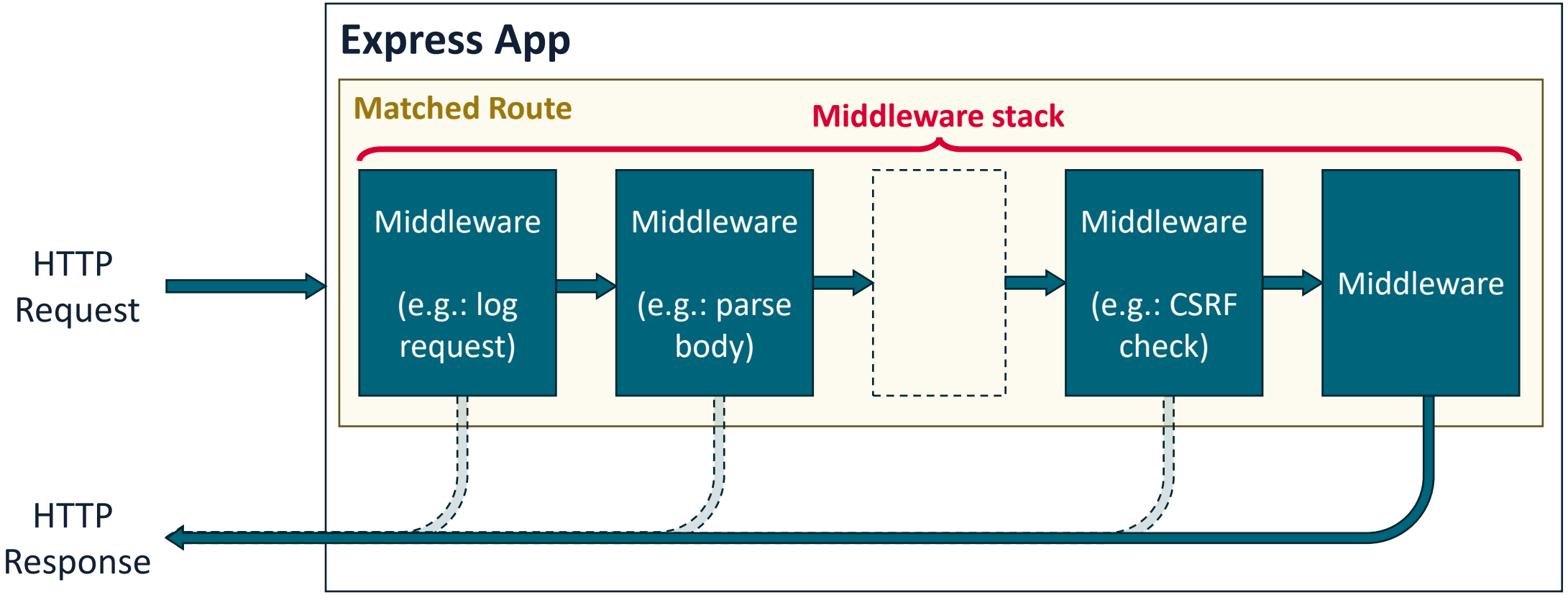
EXPRESS: THE REQUEST–RESPONSE CYCLE

- The Request-Response cycle is the core of the Express framework



EXPRESS: THE REQUEST–RESPONSE CYCLE

- The Request-Response cycle is the core of the Express framework



MIDDLEWARES

- In Express, **middlewares** are functions that can manipulate the current request and response object or execute any arbitrary code
- They’re called that because they are executed **in between** receiving a request and sending back its response
- Middlewares are executed in the same order they are declared in
- They are functions that take as input arguments:
  - **req** (the HTTP request object) and **res** (the HTTP response object)
  - **next** (a callback function that calls the next middleware in the stack)

OUR FIRST EXPRESS MIDDLEWARE

```
// defined in some module
export function logger(req, res, next) {
  console.log("LOGGER HAS BEEN CALLED!");
  next();
}
```

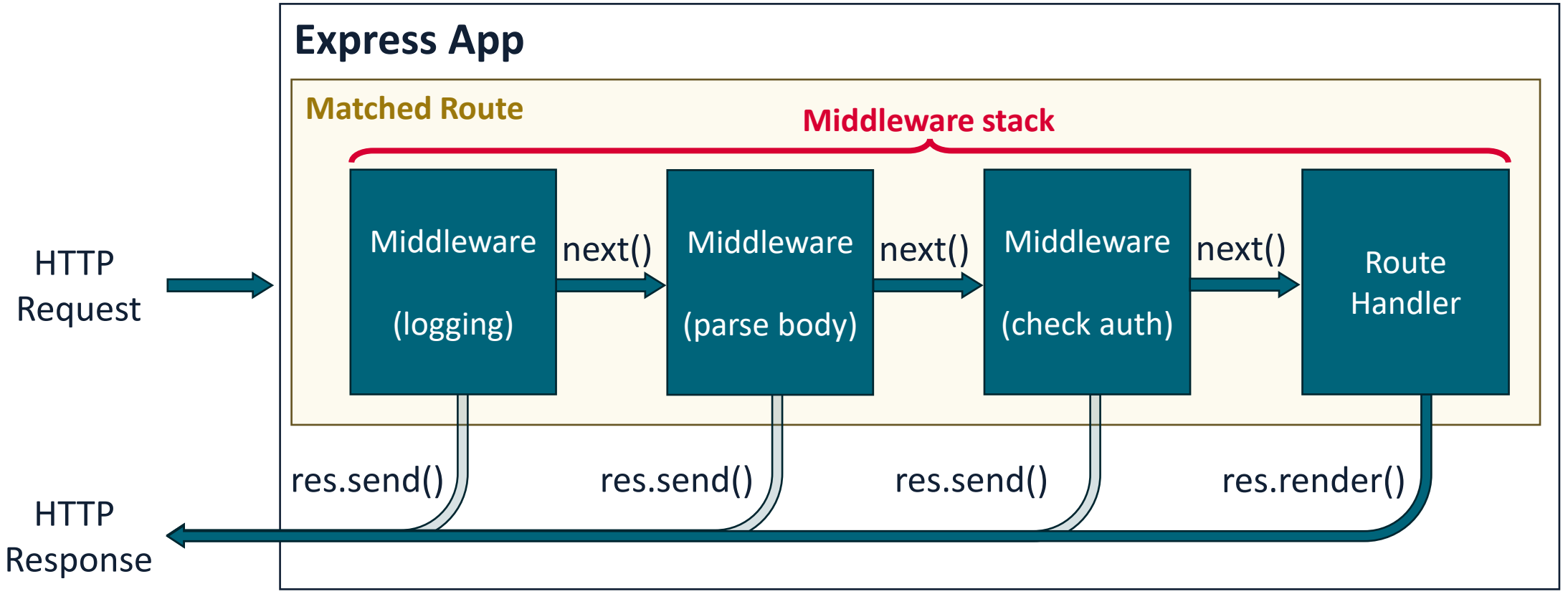
```
// index.js file starting the express app
app.use(logger); // mounts the middleware function "logger" in the "/" path

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

app.listen(PORT);
```

OUR FIRST EXPRESS MIDDLEWARE

```
// defined in some module
export function addTimestamp(req, res, next){
  req.timestamp = new Date();
  next();
}
```

```
// index.js file starting the express app
app.use(addTimestamp);
app.use(logger); // mounts the middleware function "logger" in the "/" path

app.get('/', (req, res) => {
  res.send(`Request received at ${req.timestamp}`)
});

app.listen(PORT);
```

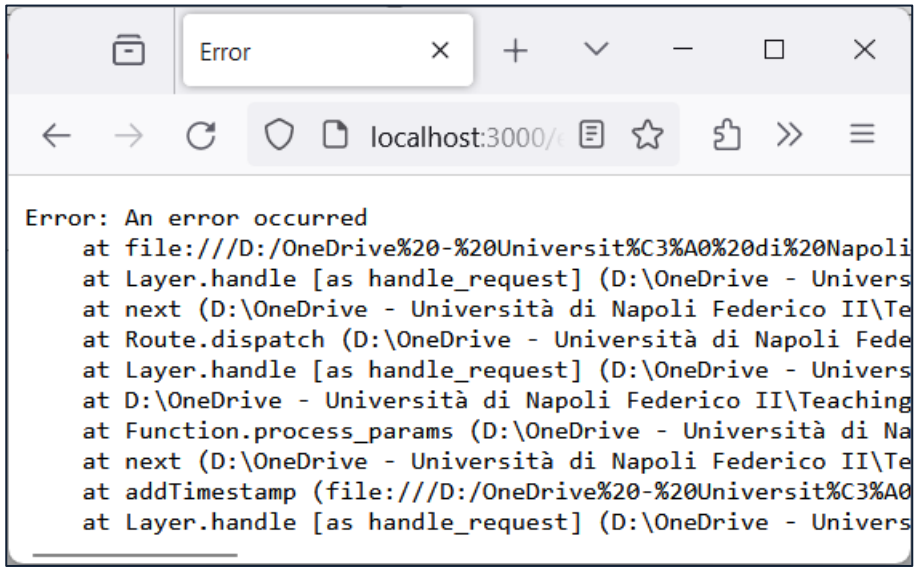
EXPRESS: THE REQUEST–RESPONSE CYCLE



HANDLING ERRORS

- It’s important to make sure that any error occurring when processing middlewares or routes is properly handled by Express.
- If an error is thrown during synchronous code execution it will be automatically caught by the Express default error handler

```
app.get("/error", (req, res) => {
  throw new Error("An error occurred");
});
```



HANDLING ERRORS

- Errors returned from asynchronous functions invoked by route handlers and middlewares must be passed to the **next()** function

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

HANDLING ERRORS (IN EXPRESS 5)

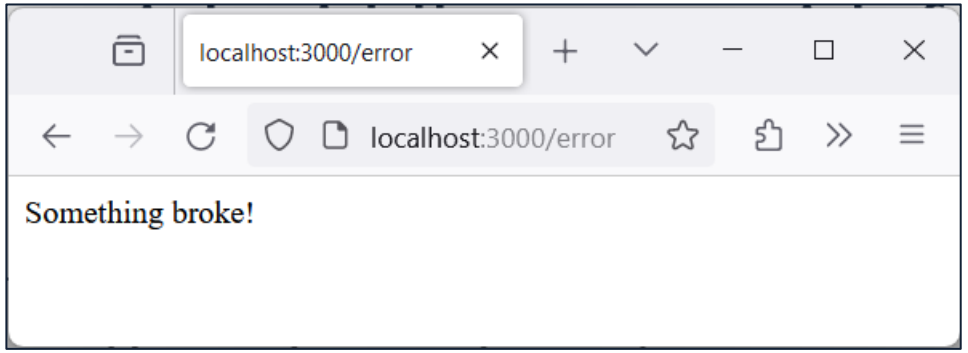
- Starting from Express 5 (currently still in beta), route handlers and middlewares that return a promise will automatically call **next(value)** when they reject or throw an error.
- So, we won't need to do it explicitly

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

ERROR HANDLERS

- Express includes a default error handler. The default handler prints the entire stack trace only when in development mode.
- We can define custom error handlers as well
- They are special middlewares with four input parameters

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



USEFUL MIDDLEWARE

EXPRESS.STATIC

express.static is a built-in middleware designed to serve static files

```
// Register the built-in express.static middleware to serve static files.
// All files in the /public directory will be served as static files.
app.use(express.static("public"));
```

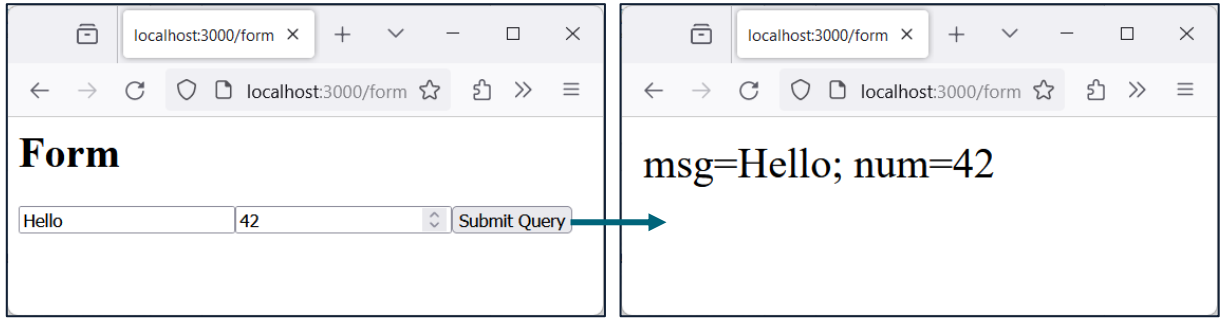
- Uses **req.url** to determine whether a request matches the specified root directory (e.g.: `public`)

EXPRESS.URLENCODED

express.urlencoded is a built-in middleware that parses the body of incoming requests with urlencoded payloads

```
// Parses url-encoded body (e.g.: those in requests submitted via forms)
// and makes the parameters available in the req.body object.
app.use(express.urlencoded({extended: false}))

app.get("/form", (req, res) => { res.render("form"); });

app.post("/form", (req, res) => {
  res.send(`msg=${req.body.msg}; num=${req.body.num}`);
});
```

```
h1 Form
form(action="/form" method="POST")
  input(type="text" name="msg")
  input(type="number" name="num")
  input(type="submit")
```



EXPRESS.JSON

express.json is a built-in middleware. Behaves similarly to express.urlencoded, but is designed to parse JSON request bodies.

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

app.post("/json", (req, res) => {
  res.send(`exam=${req.body.exam}; grade=${req.body.grade}`);
})
```

```
POST http://localhost:3000/json HTTP/1.1
Content-Type: application/json

{
    "exam": "Web Technologies",
    "grade": "A+"
}
```

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 31

exam=Web Technologies; grade=A+
```

EXPRESS – SESSION

express-session is a middleware available through npm

- It simplifies the management of a server-stored session
- Session data is stored on the server, session id is stored in a cookie
- **Notice**: the default server-side storage is not production ready! For production, you should setup a dedicated session store (e.g.: memcached or redis-based).

EXPRESS–SESSION: EXAMPLE

```
app.use(session({
  secret: "s3cr37", saveUninitialized: false, resave: false
}));

app.get("/session-counter", (req, res) => {
  if(! req.session?.count) {
    req.session.count = 1;
    res.send("It's the first time you visit this page.");
  } else {
    req.session.count = req.session.count + 1;
    res.send(`You visited this page ${req.session.count} times.`);
  }
});
```

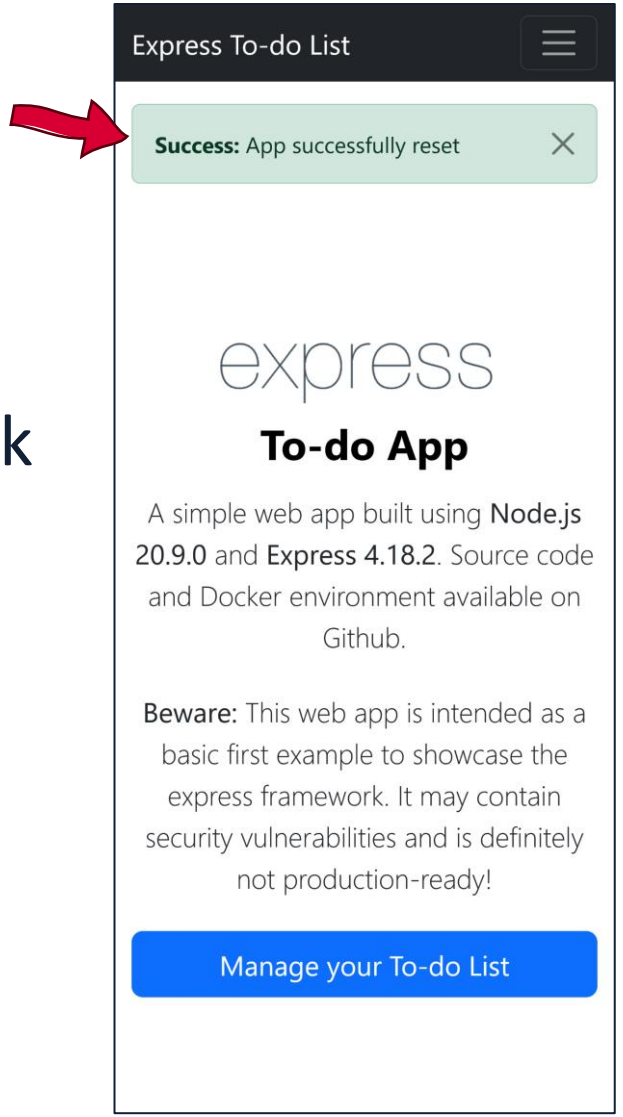
COOKIE-PARSER

- cookie-parser is a middleware available through npm
- It parses the Cookie header in incoming requests, and populates req.cookies with an object whose properties are the cookie names

```
app.use(cookieParser());

app.get("/cookie-counter", (req, res) => {
  if(! req.cookies?.count){
    res.cookie("count", 1, {maxAge: 10*60*1000});
    res.send("It's the first time you visit this page.");
  } else {
    res.cookie("count", parseInt(req.cookies.count) + 1, {maxAge: 10*60*1000});
    res.send(`You visited this page ${parseInt(req.cookies.count) + 1} times.`);
  }
});
```

FLASH MESSAGES

- Often, in web applications, users are redirected after performing some action (e.g.: redirect to homepage after deleting some data).
- It's a good rule of thumb to provide some feedback to users in these cases, so they know whether the operation they performed was successful or not.
- The «difficult» part is that, after redirecting, a whole new request-response cycle starts. How to keep track of the fact that we need to show some flash message?



FLASH MESSAGES

- Different variation of the same old problem of HTTP being stateless
- We can solve this in the same ways we did before: cookies, sessions…
- Some middlewares can be quite useful at that!
- One such example is connect-flash, available on npm

```
import flash from "connect-flash"; //simple middleware for flash messages

app.use(flash()); //register the middleware
```

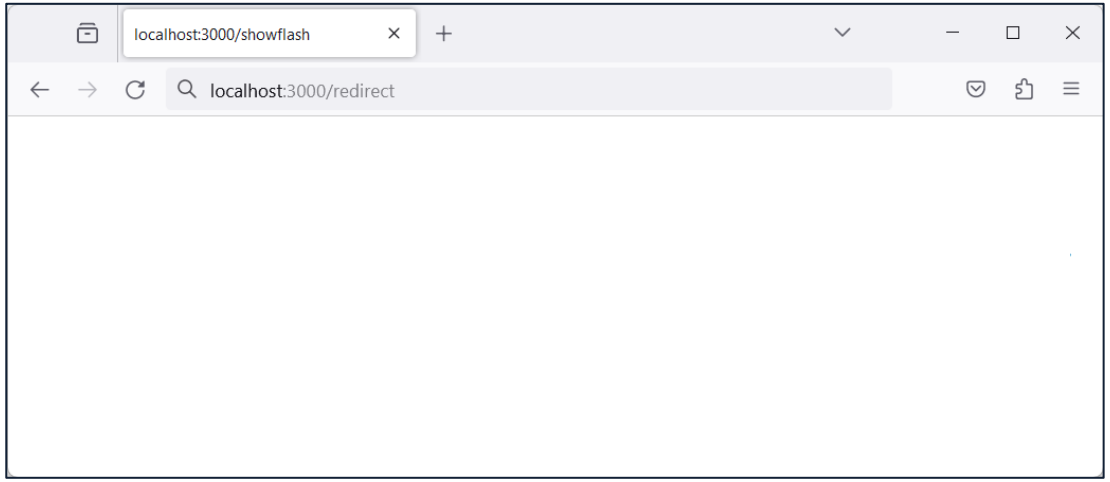
FLASH MESSAGES: EXAMPLE

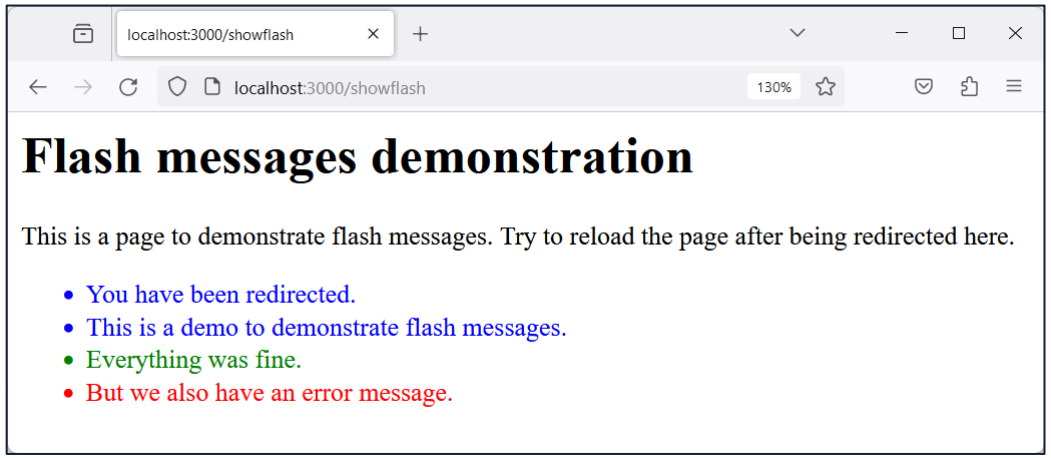
```
app.use(flash());

app.get("/redirect", (req, res) => {
  req.flash("info", "You have been redirected."); // Saves flash messages
  req.flash("info", "This is a demo to demonstrate flash messages.");
  req.flash("success", "Everything was fine.");
  req.flash("error", "But we also have an error message.");
  res.redirect("/showflash");
})
app.get("/showflash", (req, res) => {
  let infos = req.flash("info"); // Consumes flash messages with the given key
  let successes = req.flash("success");
  let errors = req.flash("error");
  res.render("flash", {infos: infos, successes: successes, errors: errors});
})
```

After being consumed, flash messages are deleted

FLASH MESSAGES: EXAMPLE

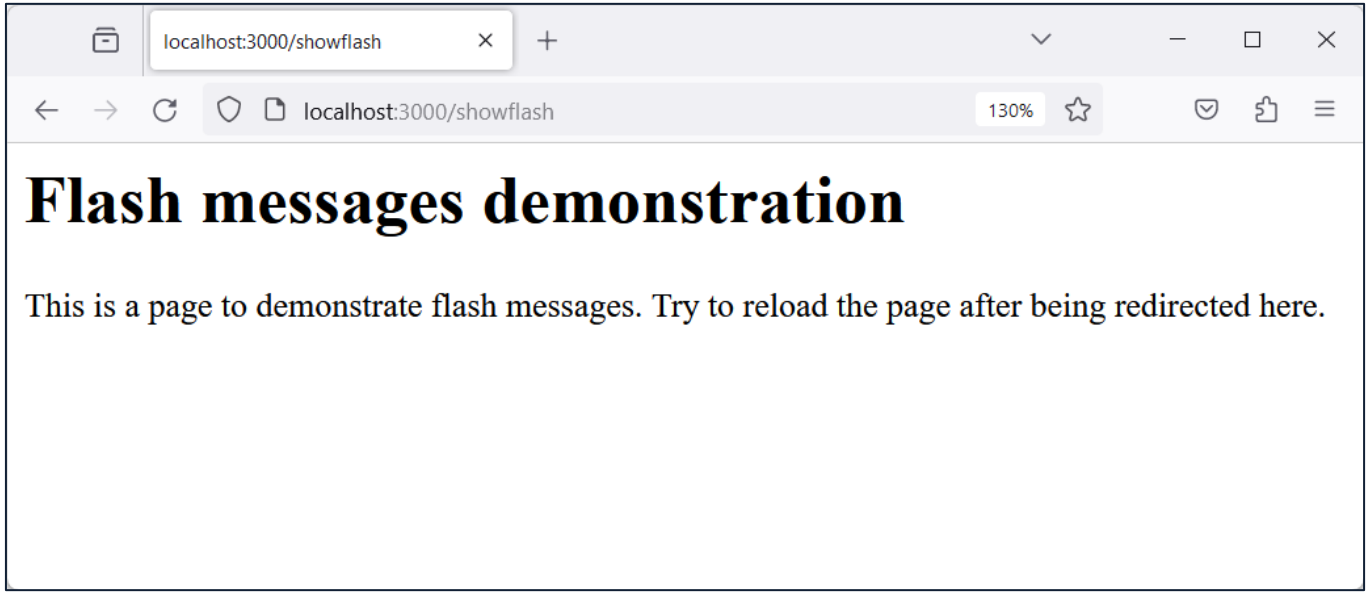
```
ul
  each info in infos
    li(style="color: blue") #{info}
  each success in successes
    li(style="color: green") #{success}
  each error in errors
    li(style="color: red") #{error}
```





FLASH MESSAGES: EXAMPLE

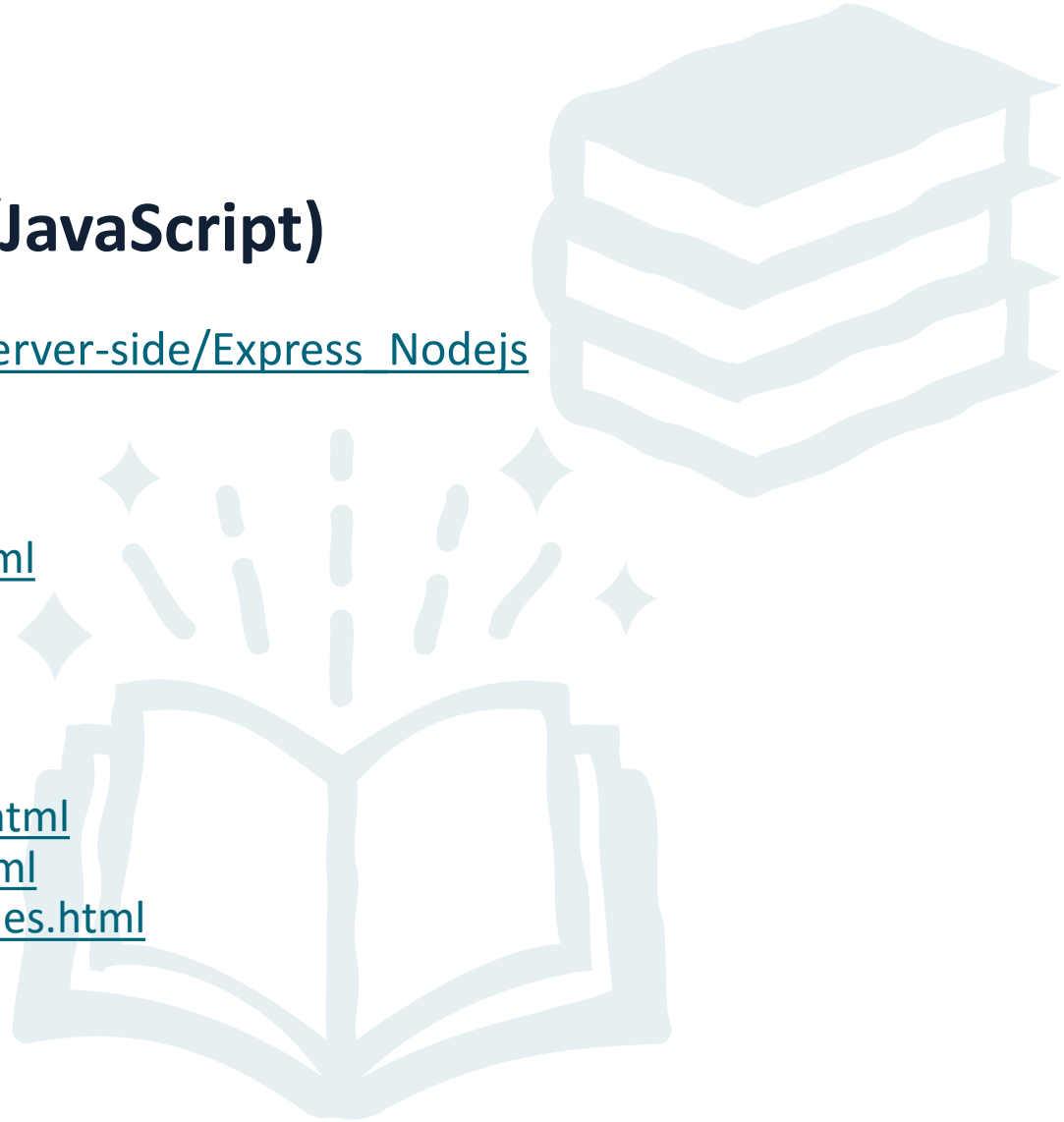
- If we reload the `/showflash` page (or visit it directly without being redirected by the `/redirect` page, no flash message will be stored in the current session, and we get:



SOURCE CODE AVAILABILITY

- The code in these slides is available in the Course Materials on Teams
  - You should check the code out, run (and debug) the web app

REFERENCES



- **Express web framework (Node.js/JavaScript)**
  MDN web docs
  https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs
- **Inversion of Control**
  By Martin Fowler
  https://martinfowler.com/bliki/InversionOfControl.html
- **Express Guide**
  Official Express website
  https://expressjs.com/en/guide/routing.html
  https://expressjs.com/en/guide/writing-middleware.html
  https://expressjs.com/en/guide/using-middleware.html
  https://expressjs.com/en/guide/using-template-engines.html
  https://expressjs.com/en/guide/error-handling.html