NodeJS

środowisko i technologia ServerSide

PAWEŁ ŁUKASZUK

Rok akademicki: 2023/2024, Semestr: 2, Wykład: 2

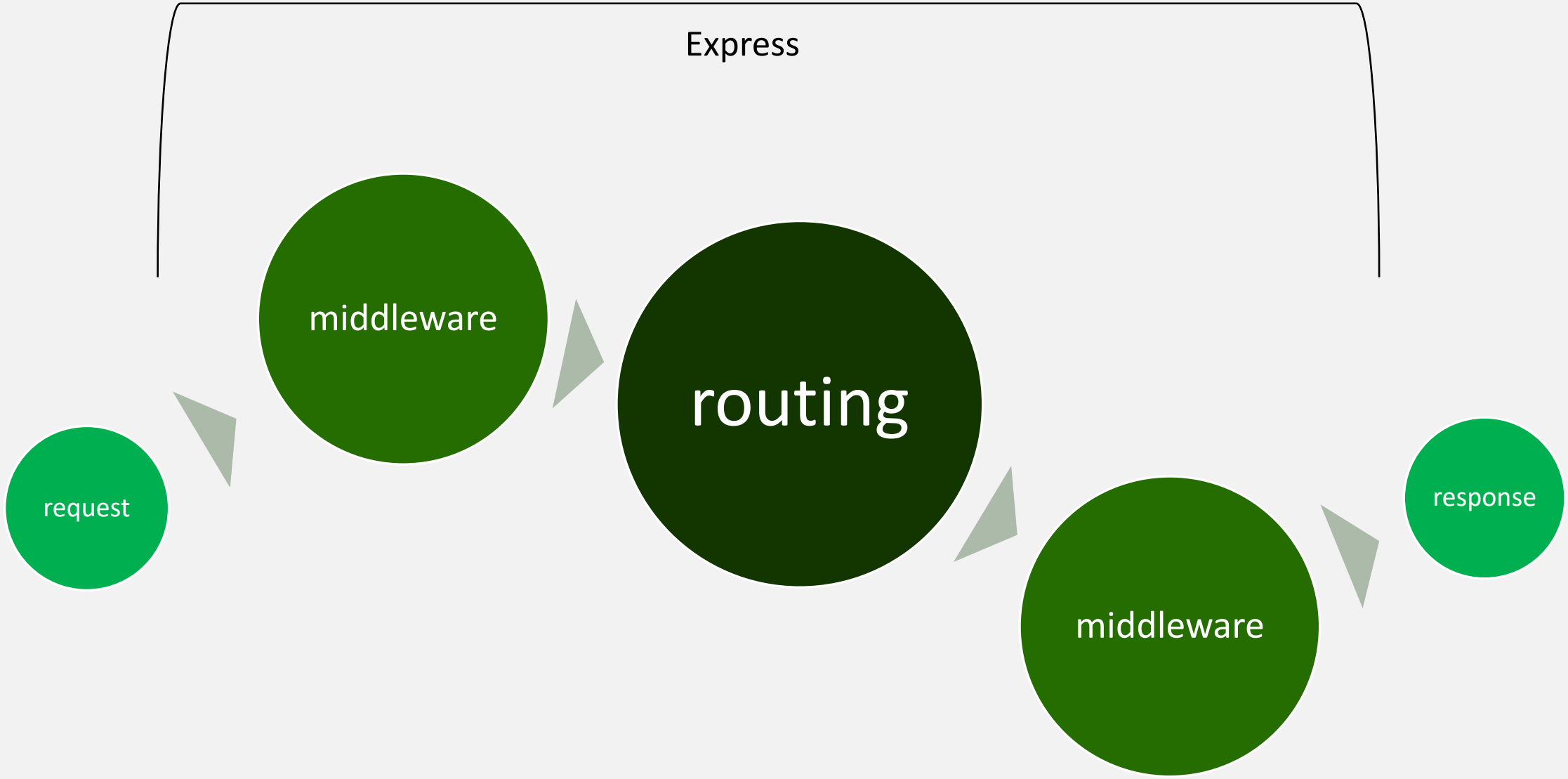
Middleware

Middleware is a type of software or function that enables communication between different applications, services or systems.

Express.js - middleware

Middleware in Express.js plays an intermediate role before executing the appropriate instruction for the path.

In the middleware function we have access to the HTTP request and response and the callback next function, which is responsible for calling the next middleware or executing to the appropriate instruction for the path.


Express
request
middleware
routing
middleware
response

Express.js - middleware

Middleware functions can perform the following tasks

- execute arbitrary code
- make changes to request and response objects
- terminate the request
- invoke the next middleware in the stack

Middleware basics

```
const express = require('express’);
const app = express();
const customMiddleware = (req, res, next) => {
     // some logic ...
     next();
};
app.use(customMiddleware);
app.get('*', (req, res) => {
     // some logic ...
     res.send('response from middleware');
});
app.listen(4700, console.log('server started'));
```

Middleware basics

```
const express = require('express’);
const app = express();
const customMiddleware = (req, res) => {
     // some logic ...
     res.send('response from middleware’);
};
app.get('*', (req, res, next) => {
     // some logic ...
     next();
});
app.use(customMiddleware);
app.listen(4700, console.log('server started'));
```

!Important

If the middleware does not complete the request/response, the next() function must be called, to pass control to the next middleware function.

Otherwise, the request will be suspended.

Example

```
// ...
const timeMiddleware = (req, res, next) => {
    req.requestTime = new Date();
    next();
};

app.use(timeMiddleware);

app.get('*', (req, res) => {
    res.send('request time: ' + req.requestTime);
});

// ...
```

Middleware types

In Express we can use such types of middleware such as:

- Application-level middleware
  https://expressjs.com/en/guide/using-middleware.html#middleware.application
- Router-level middleware
  https://expressjs.com/en/guide/using-middleware.html#middleware.router
- Error-handling middleware
  https://expressjs.com/en/guide/using-middleware.html#middleware.error-handling

- Built-in middleware
  https://expressjs.com/en/guide/using-middleware.html#middleware.built-in
- Third-party middleware
  https://expressjs.com/en/guide/using-middleware.html#middleware.third-party

APPLICATION-LEVEL MIDDLEWARE

Application-level middleware binding.

This type of middleware will execute for the entire application or for a discrete group of paths.

Example

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

app.use((req, res, next) => {
    req.requestTime = new Date();
    next();
});

app.get('*', (req, res) => {
    res.send('request time: ' + req.requestTime);
});

app.listen(4700, console.log('server started'));
```

Example with path

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

app.use('/user', (req, res, next) => {
    req.userTime = new Date();
    next();
});

app.get('*', (req, res) => {
    res.send('request time: ' + req.userTime);
});

app.listen(4700, () => console.log('server started'));
```

ROUTER-LEVEL MIDDLEWARE

Router-level middleware works similarly to application-level middleware.

Only difference is that this type of middleware runs within an express.Router() instance

Example

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

router.use((req, res, next) => {
    console.log('current time', new Date());
    next();
});

// ...
```

ERROR-HANDLING MIDDLEWARE

Error-handling middleware handles execution errors.

The definition of this type of middleware looks similar to the previous types, but this type of middleware takes four arguments.

Example

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

app.use((error, req, res, next) => {
    console.error(error.message);
    res.send(500, error.message);
});

// ...
```

BUILT-IN MIDDLEWARE

Built-in middleware that is embedded in the Express webframework

- express.static - middleware responsible for providing static resources
- express.json - parsing the content of the incoming JSON request
- express.urlencoded - parsing the content of the incoming form request

BUILT-IN MIDDLEWARE

Since version 4.x, Express provides only the three previously mentioned middleware functions.

The rest of the middleware is developed in separate modules.

List of available modules developed by Express:

https://github.com/senchalabs/connect#middleware

THIRD-PARTY MIDDLEWARE

Third-party middleware are functions created by developers outside the Express framework development team.

List of recommended modules:

https://expressjs.com/en/resources/middleware.html

Example

```
const express = require('express');
const bodyParser = require('body-parser'):

const app = express();
app.use(bodyParser.json());

// ...
```

Error handling

Error handling refers to the way the Express framework intercepts and processes errors that occur synchronously as well as asynchronously.

Express itself includes a default error handler, so if you don't need one then you don’t have to write one yourself.

Catching errors - sync

It is important that Express should capture any errors that occur while running either the routing and middleware programs.

Errors occurring in synchronous code inside routing procedures and middleware do not require errors that occur in synchronous code inside the routing and middleware routing routines do not require additional work.

If the synchronous code reports an error, Express intercepts it and processes it.

Example sync

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

app.get('/', function (req, res) {
    throw new Error('some dumb error');
});

// ...
```

Catching errors - async

In the case of errors returned by asynchronous functions called by handlers and middleware, pass them to the next() function, where Express will catch them and process them.

Example async – no error handling

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

app.get('/', function (req, res) {
    setTimeout(() => {
        throw new Error('some dumb error'); //this will break app
    }, 5000);
});

// ...
```

Example async – local error handling

```
const express = require('express');
const app = express();
app.get('/', function (req, res) {
    setTimeout(function () {
        try {
          throw new Error("some dumb error");
        } catch (err) {
          // handling error in function
        }
      }, 5000);
});
// ...
```

Example async – middleware error handling

```
const express = require('express');
const app = express();
app.get('/', function (req, res, next) { // added next function
    setTimeout(function () {
        try {
          throw new Error("some dumb error");
        } catch (err) {
          next(err); // passing to error middleware
        }
      }, 5000);
});
// ...
```

Default error handler

Express provides built-in error handling that takes care of any errors that may occur in your application.

This default software support function for handling errors is added at the END of the stack of of the middleware.

If we pass an error to the next() function and do not handle it in the custom error handling routine, it will be handled by the built-in error handling mechanism errors.

Error handlers

The middleware construct for error handling takes four arguments, not like the usual intermediate functions:

(err, req, res, next)

Error handling middleware must be used after adding the other middleware functions as well as path handlers.

Example

```
// imports

app.use(bodyParser.json());
app.use(cookieParser);
app.get('/', ...);

// routing

app.use((error, req, res, next) => {
     // error handling
});
```