TEHNICI WEB

NODE.JS

Claudia Chiriță . 2023/2024

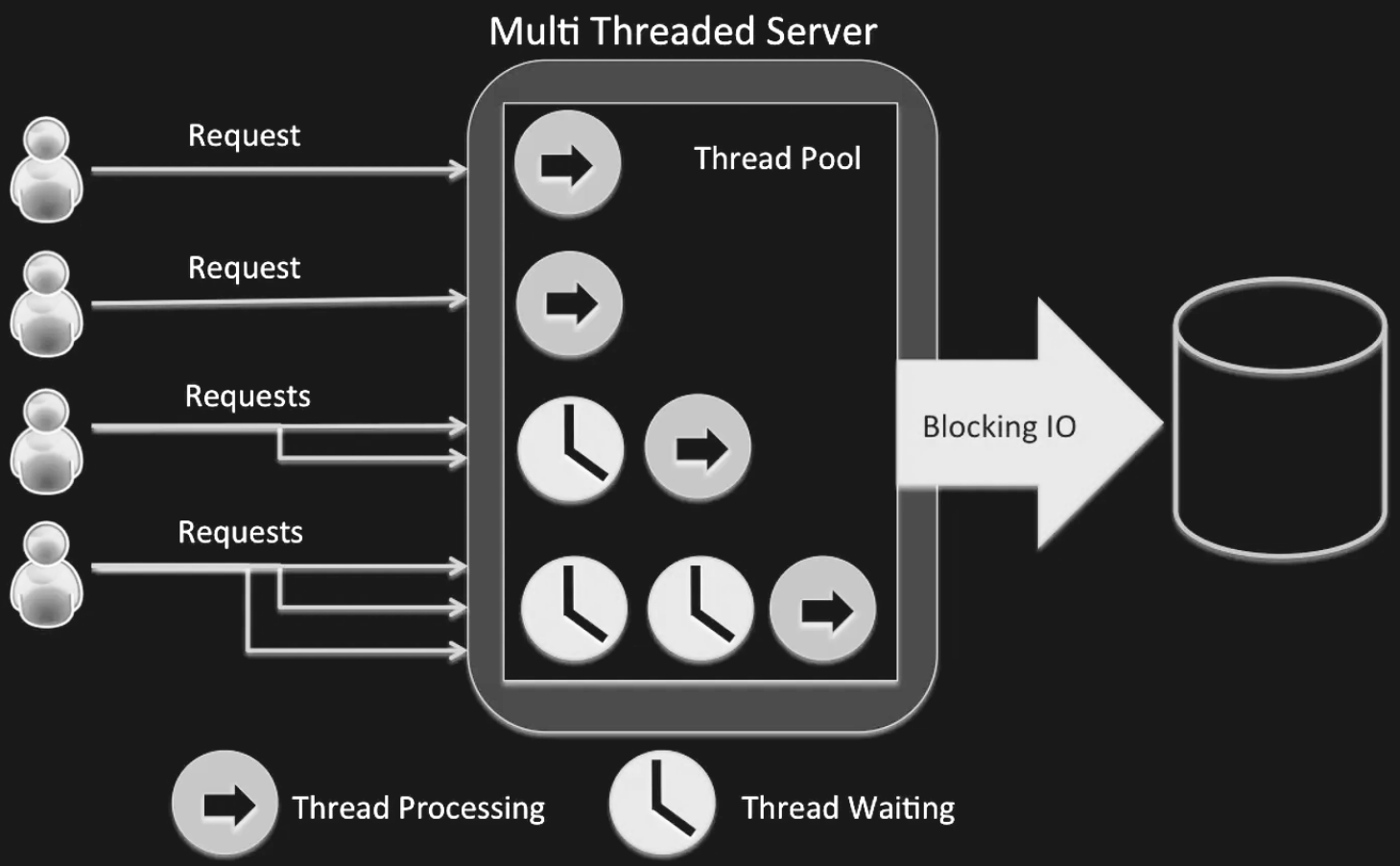
NODE.JS

- runtime environment pentru JavaScript
- permite dezvoltarea de scripturi, servere, aplicații web (la nivel de server), tool-uri în linie de comandă
- gratis, open source, pentru mai multe platforme
- inițial dezvoltat de Ryan Dahl în 2009

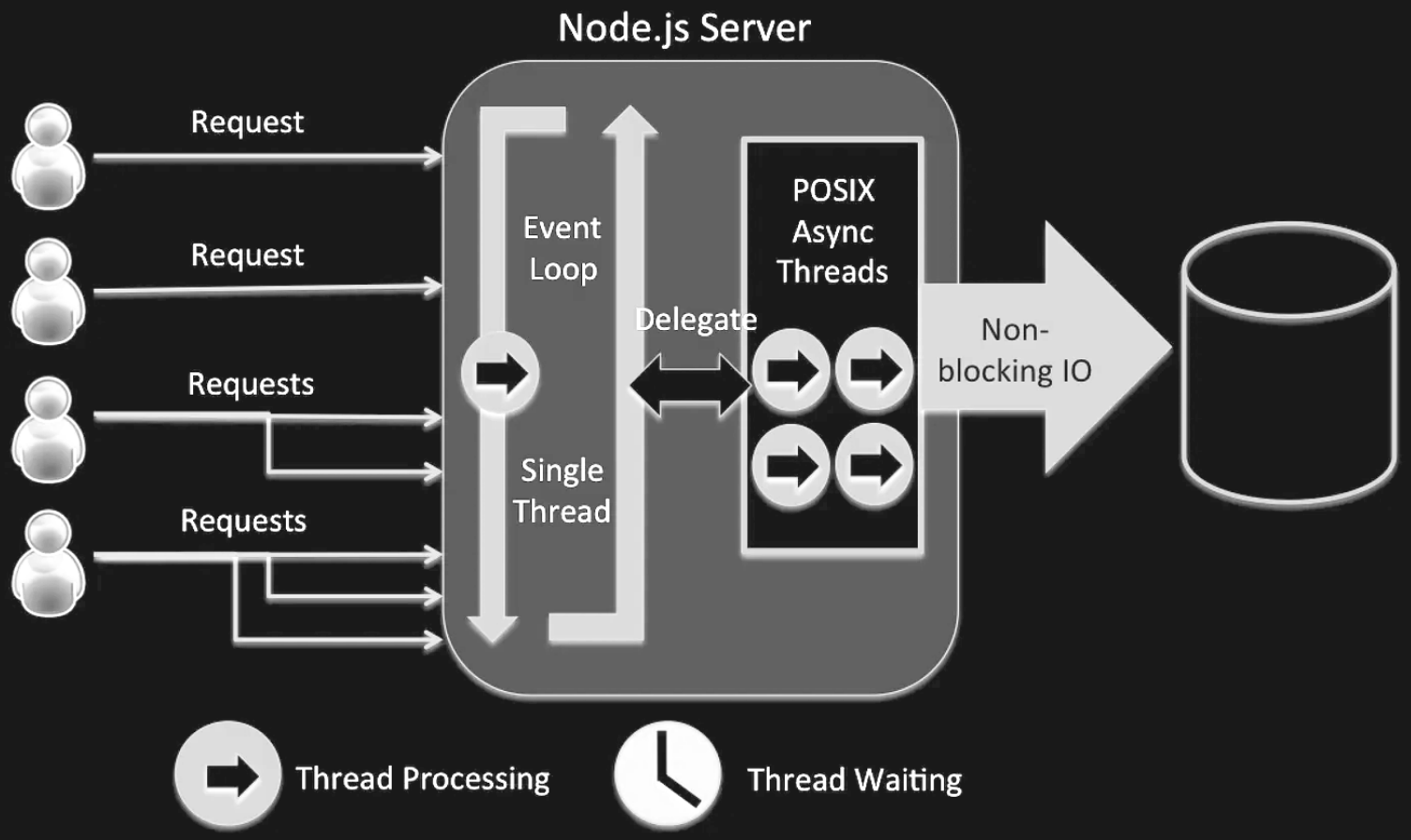
NODE.JS

- runtime environment pentru JavaScript: asincron, bazat pe evenimente
- codul JavaScript rulat pe partea de server așteaptă și tratează cereri provenite de la clienți
- o aplicație rulează într-un singur thread
- diferență majoră față de serverele web tradiționale ce folosesc servere multi-thread

NODE.JS


Multi Threaded Server
Request
Request
Requests
Requests
Thread Pool
Blocking IO
Thread Processing
Thread Waiting

NODE.JS


Node.js Server
Request
Request
Requests
Requests
Event Loop
Single Thread
Delegate
POSIX Async Threads
Non-blocking IO
Thread Processing
Thread Waiting

NODE.JS

- generare de conținut dinamic pe pagină
- crearea, deschiderea, citirea, scrierea, ștergerea și închiderea de fișiere pe server
- colectarea datelor dintr-un formular
- adăugarea, ștergerea, modificarea datelor într-o bază de date
- creare de sesiuni
- criptare/decriptare

REPL

- REPL (Read-Eval-Print-Loop)
- lansare folosind *node* în linia de comandă

```
home:~$ node
```

CUM RULĂM?

- pentru a rula o aplicație în Node.js:

```
home:~$ node app.js
```

MODULE

- aplicațiile Node.js folosesc module
- modul: fișier JavaScript
- pentru folosirea unui modul într-o aplicație Node.js folosim funcția *require* ce întoarce un obiect asociat modulului:

```
var module = require('module_name');
```

MODULE

- module predefinite care se instalează odată cu Node (http, url, fs, querystring, crypto etc.)
- funcționalități suplimentare sunt oferite în module administrate cu *npm* (nodemailer, express, formidable, cookie-parser, ejs, express-session etc.)

NPM (NODE PACKAGE MANAGER)

- utilitar pentru administrarea pachetelor: instalare, update, dezinstalare, publicarea modulelor
- se instalează odată cu Node.js
- comenzi specifice pentru operații asupra modulelor

NPM (NODE PACKAGE MANAGER)

- modulele care nu sunt predefinite trebuie instalate (adăugate ca dependențe ale proiectului):

```
npm install module_name --save
```

PACKAGE.JSON

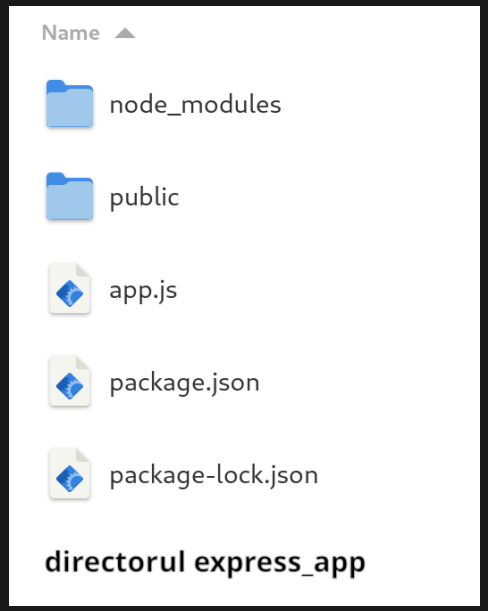
- pentru fiecare aplicație se creează un director (numit *rădăcină*) în care se vor găsi toate fișierele
- se rulează apoi în linia de comandă

```
npm init
```

pentru crearea fișierului 'package.json' în rădăcină

- conține metadate (nume modul, versiune, autor, ...) + informații privind dependențele de alte module adăugate cu *npm install*

PACKAGE.JSON



package.json

```
{
  "name": "express_app",
  "version": "1.0.0",
  "description": "my node express app",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "first",
    "node",
    "app"
  ],
  "author": "claudia",
  "license": "ISC",
  "dependencies": {
    "express": "^4.19.2",
    "formidable": "^3.5.1",
    "nodemailer": "^6.9.13"
  }
}
```

MODULUL HTTP

- include funcționalități HTTP de bază
- permite primirea și transferarea datelor prin HTTP
- clase:
  - http.Agent
  - http.ClientRequest
  - http.Server
  - http.ServerResponse
  - http.IncomingMessage
  - http.OutgoingMessage

CREAREA UNUI SERVER WEB

```
var http = require("http");
var server = http.createServer(handler);
// întoarce un obiect din clasa http.Server
```

pornirea serverului pe un port dat

```
server.listen(port, host, backlog, callback)
```

- port: numeric
- host: string
- nr. de cereri acceptate în paralel (implicit 511)
- funcție care se execută la pornirea serverului

CREAREA UNUI SERVER WEB

```
http.createServer(function(request,response){...});
// funcție care se execută atunci când clientul face o cerere
```

- obiectul *request*:
  - conține datele cererii primite de la client;
  - obiect din clasa *IncomingMessage*
- obiectul *response*: răspunsul HTTP emis de server
  - conține metode pentru setarea câmpurilor de antet, a status codului întors de server, scrierea datelor în răspuns, finalizarea raspunsului
  - obiect din clasa *ServerResponse*

CERERI EMISE DE CLIENT

metode uzuale:

```
write()   end()   setTimeout()
```

evenimente ce pot fi tratate:

```
response   connect   continue ...
```

CERERI EMISE DE CLIENT

```
request.write(chunk[, encoding][, callback])
// trimite o parte din datele din corpul cererii
```

- chunk: string
- encoding: string (implicit "utf8")
- callback: funcție care se va executa după ce datele au fost trimise

```
request.end([data[, encoding]][, callback])
// încheie trimiterea cererii
```

- data: string
- encoding: string (implicit "utf8")
- callback: funcție care se va executa după trimiterea cererii

RĂSPUNSURI EMISE DE SERVER

metode uzuale:

```
writeHead()  getHeader()  removeHeader()  write()  end()
```

evenimente ce pot fi tratate:

```
close  finish
```

proprietăți utile:

```
statusCode  headersSent
```

RĂSPUNSURI EMISE DE SERVER

```
response.write(chunk[, encoding][, callback])
// trimite o parte din date către client
```

- chunk: string
- encoding: string (implicit "utf8")
- callback: funcție care se va executa după ce datele au fost trimise

```
response.writeHead(statusCode[, statusMessage][, headers])
// trimite un antet de răspuns
```

- statusCode: numeric
- statusMessage: string
- headers: obiect

RĂSPUNSURI EMISE DE SERVER

```
response.end([data[, encoding]][, callback])
// răspunsul e complet
```

- data: string
- encoding: string (implicit "utf8")
- callback: funcție care se va executa după ce răspunsul de la server este finalizat

SALUT.JS

```
// un program JavaScript care răspunde cu un mesaj
// de salut la toate cererile adresate de clienți Web

var http = require('http')
// folosim modulul 'http' predefinit

var server = http.createServer(  // creăm un server Web
  // funcție anonimă ce tratează o cerere
  // și trimite un răspuns
  function(request,response){
    console.log("Am primit o cerere..");
    // stabilim valori pentru diverse câmpuri-antet HTTP
    response.writeHead(200, {"Content-Type" : "text/html"});

    // emitem răspunsul propriu-zis conform tipului MIME (cod
```

CLASA URL

procesarea adreselor Web

```
const myURL = new URL(input[, base])
// creează un obiect cu URL-ul parsat

myURL = new URL("http://example.com/foo/?p1=a&p2=b");

myURL = new URL("/foo/?p1=a&p2=b","http://example.com/");
```

CLASA URLSEARCHPARAMS

oferă acces la partea de query a unei adrese URL

```
const myURL = new URL('https://example.org/?abc=123');

// myURL: obiect din clasa URL
// myURL.searchParams : obiect din clasa URLSearchParams

console.log(myURL.searchParams.get('abc')); // 123
```

URL.JS

```
// program ce ilustrează procesarea URL-urilor
var url = require ('url');

var adresa = url.parse (
  'http://TehniciWeb:8080/anulI/grupa141/?nume_student=Chirita
  true // generează un obiect 'query' ce include câmpurile din
);
console.log (adresa);

if (adresa['query'].nota_student >= 5) {
  console.log (adresa['query'].nume_student + ', ai promovat e
} else {
  console.log (adresa['query'].nume_student + ', ne vedem la r
}
```

MODULUL FS

operații cu fișiere/directoare pe server:
citire, creare, adăugare date, ștergere etc.

```
readFile()  writeFile()  appendFile()
// variantele asincrone

readFileSync()  writeFileSync()  appendFileSync()
// variantele sincrone
```

MODULUL FS

```
// fs.readFile(fileName [,options], callback)
var fs = require('fs');
fs.readFile('file.txt','utf8', function (err, data) {
                                  if (err) throw err;
                                  console.log(data);
                                });
console.log('citire asincrona');
```

```
// fs.readFileSync(fileName [,options])
var fs = require('fs');
var data = fs.readFileSync('file.txt', 'utf8');
console.log(data);
console.log('citire sincrona');
```

FILE.JS

```
var fs = require('fs');

fs.readFile('test.json',function (err, data) {
  if (err) throw err;
  var json = JSON.parse(data);
        fs.writeFileSync('test.html','');
  for(var i = 0; i < json.length; i++)
  fs.appendFileSync('test.html','<img src="+json[i].picture.so
        fs.appendFileSync('test.html','');

        console.log('Operatie completa.');
});
```

EXEMPLU: FORM.HTML

trimiterea datelor dintr-un formular cu metoda GET și salvarea lor într-un fișier text

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Formular</title>
<link rel="stylesheet" href="demo.css">
</head>
<body>
<form action="http://localhost:8080/cale"
method="GET">
  <label>Nume:</label>
  <input type="text" name="name">
<br>
 <label> Varsta:</label>
 <input type="text" name="age">
<br>
  <label>Localitate:</label>
 <select name="city">
    <option value="Bucuresti"
selected>Bucuresti</option>
    <option value="Cluj">Cluj</option>
    <option value="Brasov">Brasov</option>
```

›

EXEMPLU: FORM.JS

```
var http = require('http');
var fs = require('fs');
var server = http.createServer(function (req, res)
{
   console.log("O cerere;");
   var url_parts = new URL(req.url,'http://localhost:8080/');
   console.log(url_parts);
   if(url_parts.pathname =='/cale'){
    var query = url_parts.searchParams;
     fs.appendFileSync('date.txt', query.get('name') + ',' + q
     res.writeHead(200, {'Content-Type': 'text/plain'});
     res.end(query.get('name') + ' din ' + query.get('city') +
  }).listen(8080);
console.log ('Serverul creat asteapta cereri la http://localho
```

EXEMPLU: STUDENTI

- la requesturi către "/salveaza" (trimiterea datelor dintr-un formular cu metoda POST) se va afișa un răspuns în format html cu datele trimise și se vor salva datele într-un fișier json
- la requesturi către "/afiseaza" se vor citi date dintr-un fișier json și se va afișa răspunsul sub forma unui tabel html cu datele din fișier

EXEMPLU: STUDENTI.HTML

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Formular</title>
<link rel="stylesheet" href="demo.css">
</head>
<body>
<form action="http://localhost:8030/salveaza"
method="POST">
  <label>Nume student:</label>
  <input type="text" name="nume">
<br><br>
  <label>Grupa:</label>
 <select name="grupa">
    <option value="1" selected>1</option>
    <option value="2">2</option>
    <option value="3">3</option>
 </select>
<br><br>
<label> Nota:</label>
 <input type="range" name="nota" min=1 max=10>
```

EXEMPLU: STUDENTI.JS

```
const http = require('http');
const querystring = require('querystring');
const fs = require('fs');
const server = http.createServer(function(request, response){
   var body = "";
   console.log(request.url);
   var url_parts = new URL(request.url,'http://localhost:8030/

   if(url_parts.pathname =='/salveaza'){
    request.on('data', function(date){
        body += date; console.log(body);
        });
   request.on('end', function(){
        console.log("Am primit o cerere");
        console.log(body);
```

EXEMPLU: STUDENTI.JS

```
if(url_parts.pathname =='/afiseaza'){
        fs.readFile("studenti.json",function(err,date){
        if(err) throw err;
        var studenti = JSON.parse(date);
        response.statusCode = 200;
        response.write('<html><body><table style="border:1px s
  for(s of studenti) {
    response.write('<tr><td style="border:1px solid black">');
    response.write(s.nume);
    response.write('</td><td style="border:1px solid black">')
    response.write(s.grupa);
    response.write('</td><td style="border:1px solid black">')
    response.write(s.nota);
    response.write('</td></tr>');
```

MODULE CUSTOM

- module create de utilizator și incluse apoi în app
- folosind *module.exports*
- mymodule.js

```
module.exports = {
myDate : function () {
  var data = new Date();
  return data; },
myMessage: function() { return 'Node.js';} };
```

MODULE CUSTOM

mymodule.js

```
var http = require('http');
var date = require('./mymodule');

http.createServer(function(req, res) {
  console.log('am primit un request');
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end('<html><body>Astazi <b>' + date.myDate()
         + '</b> am invatat despre <b> '
         +  date.myMessage() + '</b></body></html>');
        }).listen(8050);

console.log ('Serverul creat asteapta cereri la http://localho
```

MODULUL CRYPTO

oferă metode pentru criptarea și decriptarea datelor
(ex. securizarea parolelor înainte de a fi stocate în BD)

```
createCipheriv()   createDecipheriv()   update()   final()
```

MODULUL CRYPTO

```
const crypto = require('crypto');
//includem modulul crypto

function encrypt(text){
var cipher = crypto.createCipheriv('aes128', // creeaza un obi
'passwordpassword', 'vectorvector1234')
var crypted = cipher.update(text, 'utf8', 'hex') // criptarea
crypted += cipher.final('hex') // finalizarea criptarii
return crypted;
}

function decrypt(text){
var decipher = crypto.createDecipheriv('aes128',
'passwordpassword', 'vectorvector1234')
var dec = decipher.update(text, 'hex', 'utf8')
```

MODULUL NODEMAILER

trimiterea de emailuri folosind modulul *nodemailer*

```
var nodemailer = require('nodemailer');

var transporter = nodemailer.createTransport({
service: 'gmail',
auth: {
user: 'my.mail.node@gmail.com',
pass: 'nodemailer'
}}); //face autentificarea

var mailOptions = {
from: 'my.mail.node@gmail.com',
to: 'your.mail@gmail.com',
subject: 'Mesaj din Node.js',
text: 'Hello!'
}; //optiunile mesajului
```

MODULUL NODEMAILER

folosirea modulului nodemailer cu gmail

EXPRESS.JS

- framework minimalist cu ajutorul căruia se pot implementa aplicații web mai ușor (cod mai simplu și mai clar)
- integrează diferite module pentru procesarea de cereri și de răspunsuri HTTP (express-session, cookie-parser, nodemailer etc.)

EXPRESS.JS

- oferă metode pentru *routing* (crearea de rute) pentru stabilirea modului de procesare a unei cereri în funcție de resursa solicitată și de metoda folosită
- permite redarea dinamică a paginilor HTML pe baza unor template-uri (ejs)
- furnizează accesul la informații stocate în diferite surse de date

EXPRESS.JS

instalare:

```
npm install express --save
```

EXPRESS()

creare server cu express

```
var express = require('express');
// importăm modulul express; obținem o funcție pe care
// o apelăm pentru a crea o aplicație express

var app = express(); // obiectul app corespunzător aplicației
// express are metode pentru:
// - definirea rutelor corespunzatoare cererilor HTTP
// - redarea HTML (template-uri folosind EJS)
// - accesul la resurse statice (middleware-ul express.static)

app.listen(5000);
// pornirea serverului la portul specificat
```

ROUTING

- rutele create în Express.js reprezintă modul de procesare a cererii în funcție de tipul ei (GET, POST, PUT, DELETE) și a resursei cerute

ROUTING

```
app.metoda(cale_ruta, callback)
```

- metoda: get, post, put, delete
- cale_ruta: expresie regulată
- callback:

```
function(request,response,next){..}
```

- obiectul request
- obiectul response
- next: funcția middleware următoare
- dacă în funcția callback răspunsul emis este complet, se folosesc doar primii doi parametri

ROUTING

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

// ruta către rădăcina (cereri get) (http://localhost:5000/)
app.get( "/", function(req,res){ res.send('root'); });

// ruta către pagina1 (http://localhost:5000/pagina1)
app.get( "/pagina1", function(req,res){ res.send('cerere către

// ruta către toate paginile care se termina cu .html (http://
app.get("/*.html", function(req,res){ //procesarea cererii });

app.listen(5000);
```

OBIECTUL DE TIP REQUEST

proprietăți utile pentru procesarea cererilor

```
app.get('/cale', function(req, res) {...});

req.query // obiect ce conține parametrii din query
req.body // obiect ce conține body-ul parsat
req.path // partea din url denumită path
```

OBIECTUL DE TIP RESPONSE

metode pentru setarea răspunsului HTTP

```
app.get('/cale', function(req, res) {...});

res.write(content) // scrie în conținutul răspunsului
res.status(code) // seteaza status codul răspunsului
res.end() // încheie răspunsul
res.end(msg) // încheie răspunsul cu un conținut
res.send(content) // write() + end()
res.redirect(url) // redirecționare către alt url
```

FUNCȚII MIDDLEWARE

- utilizate atunci când sunt necesare mai multe procesări pentru a răspunde la o anumită solicitare
- funcții care primesc ca argumente obiectele request, response și următoarea funcție (denumită de obicei next) din ciclul cerere-răspuns al aplicației
- dacă funcția middleware curentă nu încheie ciclul cerere-răspuns, trebuie să apeleze next() pentru a trece controlul la următoarea funcție middleware; altfel, cererea va rămâne suspendată

FUNCȚII MIDDLEWARE

```
app.use(function (req, res, next) {...})
```

- metoda use() este folosită pentru setarea unei funcții middleware
- ordinea de setare a funcțiilor contează: procesările se fac în ordinea în care au fost definite

FUNCȚII MIDDLEWARE

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

app.use('/pagina1', function(req, res, next){
// funcție middleware care se execută la fiecare request
// către '/pagina1', înaintea funcției handler
var data = new Date();
console.log("O cerere catre pagina1 a fost primita in " + data
next();
});

app.get('/pagina1', function(req, res){
// funcție handler care trimite răspunsul
res.send('Pagina 1');
});
```

FISIERE STATICE

- fișierele statice sunt fișiere pe care clienții le descarcă așa cum sunt de pe server
- în mod implicit, Express nu poate servi fișiere statice: folosim middleware-ul *express.static*

```
app.use(express.static('director'))
```

'director' este numele unui director static în folderul rădăcină al aplicației Express

FISIERE STATICE

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

app.use(express.static('html'));
app.use(express.static('poze'));

app.listen(3000);
```

'html' şi 'poze' sunt directoare în folderul rădăcină al aplicaţiei

MIDDLEWARE URLENCODED

parsează body-ul pentru formulare trimise cu post

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

app.use(cale_ruta,express.urlencoded({extended:true/false}))

// cale_ruta - calea unde se vor trimite datele trimise cu pos
// extended:true - permite obiecte încapsulate
```

MIDDLEWARE URLENCODED

exemplu:

```
app.use('/post', express.urlencoded({extended:true}));

app.post('/post',function(req, res) {console.log(req.body);
res.send(req.body.persoane.name + ' din '
         + req.body.city + ' are ' + req.body.age
         + ' (de) ani');})
```

EXEMPLU: STUDENTI

exemplul cu studenți rescris cu Express.js

MODULUL EJS

- EJS (Embedded JavaScript Templates) este un view engine utilizat pentru generarea de marcaj HTML cu ajutorul JavaScript (template-uri)
-
```
npm install ejs --save
```

.EJS

- un document EJS (cu extensia '.ejs') conține cod HTML și JavaScript care referă atributele transmise de la nivelul de logică a aplicației (serverul)
- fișierele de tip template trebuie salvate într-un director numit **views**
- trebuie specificat motorul (*view engine*) care va fi utilizat pentru redarea acestor fișiere

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

DELIMITATORI

delimitatori folosiți în documentele EJS pentru încadrarea codului JavaScript:

- **<%** controlul fluxului de execuție al programului ca delimitator de început; nu produce nimic în HTML
- **<%=** evaluarea rezultatului expresiei conținute și plasarea acestuia în șablonul obținut, înlocuind caracterele speciale HTML prin codul lor (afișează tagurile)
- **<%-** evaluarea rezultatului expresiei conținute și plasarea acestuia în șablonul obținut, fără a înlocui caracterele speciale HTML prin codul lor (interpretează tagurile)
- **<%#** comentarii; codul JavaScript nu este executat; nu produce vreun rezultat

DELIMITATORI

delimitatori folosiți în documentele EJS pentru încadrarea codului JavaScript:

- **<%%** redarea secvenței '<%' la nivelul paginii HTML care este generată pe baza sa
- **%>** controlul fluxului de execuție al programului ca delimitator de sfârșit
- **-%>** eliminarea caracterelor '\n', în cazul în care acestea sunt conținute în codul JS
- **<%_ , _%>** eliminarea spațiilor de dinainte de (respectiv de după) el

TEMPLATES

```
<%- include('cale-fisier') %>
```

- inserarea codului din fișierul specificat ca parametru
- se folosește pentru zonele de cod care apar pe mai multe pagini (ex. header, footer, meniu, meta)

TEMPLATES

- recomandare: crearea unui subdirector în **views** cu fragmente de cod de inserat (fișiere partiale) și un subdirector cu paginile aplicației

```
<header>
<%- include('../partiale/header'); %>
</header>
```

EXEMPLU: MYINDEX.EJS

```
<!-- views/pagini/myindex.ejs -->
<!DOCTYPE html>
<html lang="ro">
<head>
   <%- include('../partiale/head'); %>

   <link rel="stylesheet" href="demo.css">
</head>
<body>

   <header>
      <%- include('../partiale/header'); %>
   </header>

   <main>
     <div>
        <h1>EJS</h1>
        <p>EJS este un view engine utilizat
           pentru template-uri</p>
     </div>
   </main>
```

EXEMPLU: MYINDEX.EJS

```
<!-- views/partiale/head.ejs -->

<meta charset="UTF-8">
<title>Template cu EJS</title>
<style>
  body {padding-top:50px;}
  header{width:80%; border:1px solid red;}
  footer{width:80%; border:1px solid blue;}
</style>
```

EXEMPLU: MYINDEX.EJS

```
<!-- views/partiale/header.ejs -->

<nav>
<a href="">Meniu 1</a>
<a href="">Meniu 2</a>
<a href="">Meniu 3</a>
</nav>
```

```
<!-- views/partiale/footer.ejs -->

<p>Footer: pagină creată cu EJS</p>
```

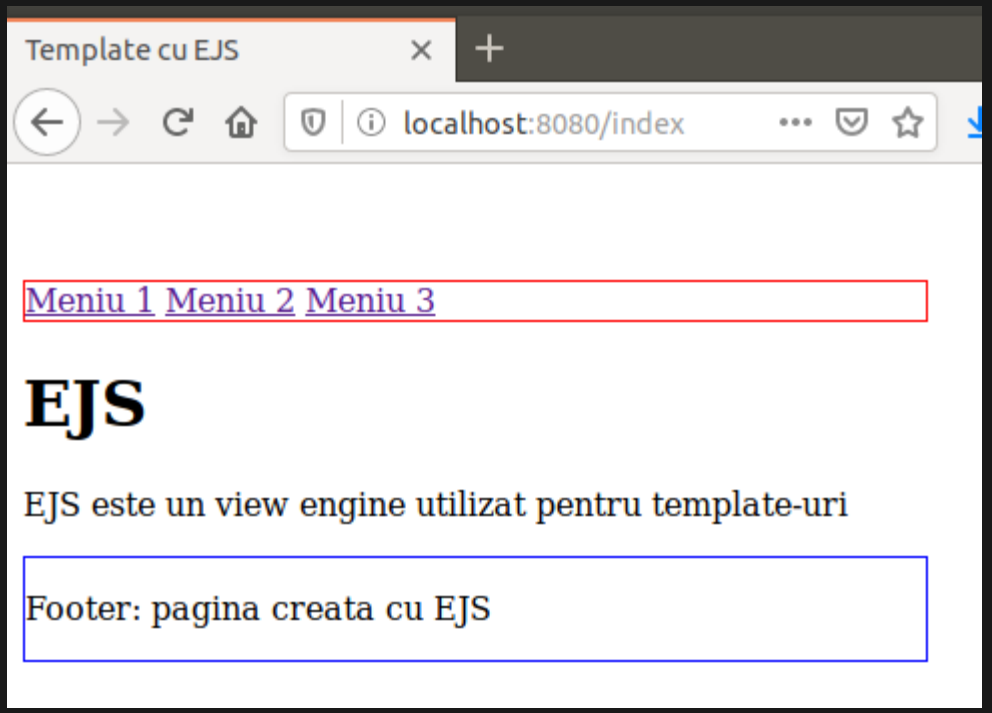
RENDER()

- folosește view engine-ul setat pentru a genera și a afișa pagina

```
response.render(cale-relativa-fisier, date)
// calea relativă la folderul views
```

```
// app.js
...
app.get('/myindex', function(req,res){
res.render('pagini/myindex');});
...
```

EXEMPLU: MYINDEX.EJS

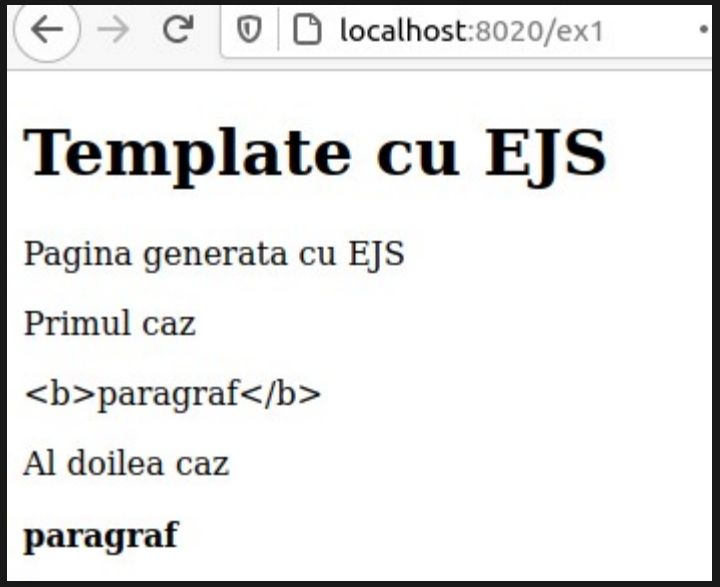


EXEMPLU: RENDER()

```
// ex1.ejs
...
<body>
<h1><%= titlu %></h1>
<p>><%= continut %></p>
<% var x="<b>paragraf</b>"%>
<p>Primul caz</p>
<%= x %>
<p>Al doilea caz</p>
<%- x %>
</body>
...
```

EXEMPLU: RENDER()

```
// app.js
....
app.get('/ex1', function(req,res){
  res.render('pagini/ex1',{titlu:'Template cu EJS',
  continut:'Pagina generata cu EJS'});});
...
```



EXEMPLU: VECTOR

- afișarea unui vector de stringuri sub forma unei liste ordonate (în HTML)

```
// app.js
app.get('/for_vector', function(req, res) {
animale = ["tapir", "pisică", "wombat", "ornitorinc"]
plante = ["ghiocel", "tei", "mimoză"]
res.render('for_vector', {vector_animale: animale,
                          vector_plante: plante});
```

EXEMPLU: VECTOR

```
// for_vector.ejs
<ol>
<% for (var i = 0; i < vector_animale.length; i++) { %>
<li><%= vector_animale[i] %></li>
<% } %>
</ol>
```

- fiecare secvență de cod JavaScript se scrie între <% ... %>
- <%= afișează valoarea variabilei scrise după el (vector_animale[i])
- acoladele de la for (și alte instrucțiuni repetitive/condiționale) sunt obligatorii

EXEMPLU: IF-ELSE

```
<p>
<% var x = Math.random();
if (x<0.5) {%>
<%= (x+"e mai mic ca 0.5") %>
<% } else {%>
<%= (x+"e mai mare ca 0.5") %>
<% } %>
</p>
```

EXEMPLU: SWITCH

```
<p>
  <% culori = ["red","green","blue"]
  var ind = Math.trunc(Math.random()*culori.length); %>
  Culoarea este:
  <span style='color:<%= culori[ind] %>'>
  <% switch (ind){
      case 0: %> rosie
      <% break;
      case 1:%> verde
     <% break;
     case 2:%> albastra
   <% } %>
   </span>
</p>
```

TEMPLATES

- daca vrem să folosim extensia .html în loc de .ejs scriem în app.js:

```
app.set('view engine', 'html');
ejs = require('ejs')
app.engine('.html', ejs.renderFile);
```

- daca vrem să setăm alt director de views decât cel default:

```
app.set('views', 'templateuri');
```

MODULUL COOKIE-PARSER

- folosit pentru definirea cookie-urilor
- date stocate în browser și trimise de server împreună cu răspunsurile la cererile clienților
- pot fi utilizate pentru întreținerea sesiunilor

```
npm install cookie-parser --save
```

```
var cookieParser = require('cookie-parser');
app.use(cookieParser());
```

MODULUL COOKIE-PARSER

- crearea unui cookie

```
response.cookie(nume-cookie,valoare-cookie)
```

```
var date_user = {'nume':'Claudia','varsta':33}
response.cookie('user',date_user);
```

- ștergerea unui cookie

```
response.clearCookie(nume-cookie)
```

- accesarea cookie-urilor

```
request.cookies
```

EXEMPLU: COOKIES

- la trimiterea unui formular se creează un cookie cu optiunea selectată, iar la reîncărcarea paginii se va păstra selecția făcută

```
<html lang="us">
<body>
<p>limba selectata: <%= selectedLang %></p>
<form action="formpost" method="post">
<select name="limba">
<% langs.forEach(function(lang) {%>
<option <%= lang == selectedLang ? 'selected' : '' %>>
<%= lang %></option>
<% }); %>
</select>
<button type="submit">save</button>
</form>
</body>
</html>
```

EXEMPLU: COOKIES

- la trimiterea unui formular se creează un cookie cu opțiunea selectată, iar la reîncărcarea paginii se va păstra selecția făcută

```
// app.js
var express = require('express');
var app = express();
var cookieParser = require('cookie-parser');
app.use(cookieParser());
app.set('view engine', 'ejs');
app.use('/formpost',express.urlencoded({extended:true}));
app.get('/form', function(req,res){
res.render('pagini/form',{selectedLang:req.cookies.limba,
langs : "['romana', 'franceza', 'greaca', 'spaniola']"});
});
app.post('/formpost', function(req,res){
res.cookie('limba', req.body.limba);
res.send('saved');
});
```

MODULUL EVENTS

- permite utilizarea de evenimente definite de utilizator

```
var events = require('events');
var eventEmitter = new events.EventEmitter();
```

MODULUL EVENTS

- metoda *on()* este utilizată pentru înregistrarea handlerului evenimentului

```
eventEmitter.on('myevent', eventHandler);
```

- pot fi înregistrate mai multe funcții handler pentru un eveniment

```
eventEmitter.on('myevent', anotherEventHandler);
```

MODULUL EVENTS

- metoda *emit()* este utilizată pentru declanșarea evenimentului

```
eventEmitter.emit('myevent',[argumente]);
```

- permite transmiterea unui set de argumente arbitrare funcțiilor handler asociate cu metoda *on()*

```
eventEmitter.on('myevent',function(arg1,arg2){
                              for(let i = 0; i < arg1; i++)
                                console.log(arg2);});
eventEmitter.emit('myevent',10,'a');
```

MODULUL FORMIDABLE

- folosit pentru preluarea datelor dintr-un formular și upload de fișiere
- permite accesarea atât a datelor de tip text, cât și a fșierelor din inputurile de tip file dintr-un formular

```
npm install formidable --save
```

```
var formidable = require('formidable');
```

MODULUL FORMIDABLE

- crearea unui obiect de tip formular folosind clasa *IncomingForm*:

```
var form = new formidable.IncomingForm();
```

- parsarea datelor din cererile de tip post:

```
form.parse(req,function(err,fields,files){...});
```

  - metoda *parse* primește ca argumente obiectul cerere *req* și o funcție callback care va prelucra datele după parsare
  - în obiectul *fields*: câmpurile formularului în afara celor de tip file
  - în obiectul *files*: câmpurile de tip file

MODULUL FORMIDABLE

- pentru upload de fișiere, trebuie să setăm calea la care se va face uploadul
  - fie la crearea obiectului de tip formular

```
var form = new formidable.IncomingForm
    ({uploadDir:'cale_director', keepExtensions:true})
```

  - fie în handlerul evenimentului *fileBegin* care se declanșează la începutul încărcării fișierului

```
form.on('fileBegin', function(name,file){
    file.path = 'cale_director' + file.name;
    // ca să păstrăm numele inițial al fișierului din fil
}
```

MODULUL FORMIDABLE: EXEMPLU

la trimiterea unui formular cu metoda post se salvează datele într-un fișier .json, iar fișierele se vor uploada în directorul 'upload'

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

app.use(express.static('html'));
app.post('/salveaza',function(req,res){
        var ob;
        if (fs.existsSync("pers.json")){
          var date = fs.readFileSync("pers.json");
          ob = JSON.parse(date);
        }
        else ob = [];
        var form = new formidable.IncomingForm
          ({uploadDir:'upload', keepExtensions:true});
```

SESIUNI

- utile atunci când vrem sa păstram date de la un request la altul
- la crearea unei sesiuni, clientul primește un session ID; pentru fiecare request viitor al clientului, vom accesa informații despre sesiune folosind acel ID

```
npm install express-session -save
```

```
const session = require('express-session');
```

SESIUNI

- pentru a crea o sesiune, se setează middleware-ul:

```
app.use(session({
secret: 'abcdefg', // pentru criptarea session ID-ului
resave: true, // să nu șteargă sesiunile idle
saveUninitialized: false
// nu salvează obiectul sesiune dacă nu am setat niciun câmp
}));
```

- după crearea sesiunii, în obiectele de tip request va fi disponibilă o proprietate nouă, numită *session* – obiect în care putem seta proprietăți cu valorile pe care dorim să le salvăm în sesiunea curentă

SESIUNI: EXEMPLU

contorizarea vizitării unei pagini

```
var express = require('express');
var app = express();
const session = require('express-session');

app.use(session({
    secret: 'abcdefg',
    resave: true,
    saveUninitialized: false,
}));

app.get('/pagina1', function(req, res){
    if(req.session.vizitat){
      req.session.vizitat++;
    }
    else {
```

SESIUNI: LOGIN/LOGOUT

formularul de login dintr-un document html/ejs

```
<form method="post" action="login">
  <label>
    Username: <input type="text" name="username"
                     value="Claudia">
  </label>
  <label>
    Parola: <input type="password" name="parola"
                   value="">
  </label>
  <input type="submit" value="Submit">
</form>
```

SESIUNI: LOGIN/LOGOUT

login

```
app.post('/login', function(req, res) {
   var form = new formidable.IncomingForm();
   form.parse(req, function(err, fields, files) {
        user =  verifica(fields.username, fields.parola);
        // verificarea datelor de login

        if(user){
          req.session.username = user;
          // setez userul ca proprietate a sesiunii
          res.redirect('/logat'); }
        else
          req.session.username = false;
   });
});
```

SESIUNI: LOGIN/LOGOUT

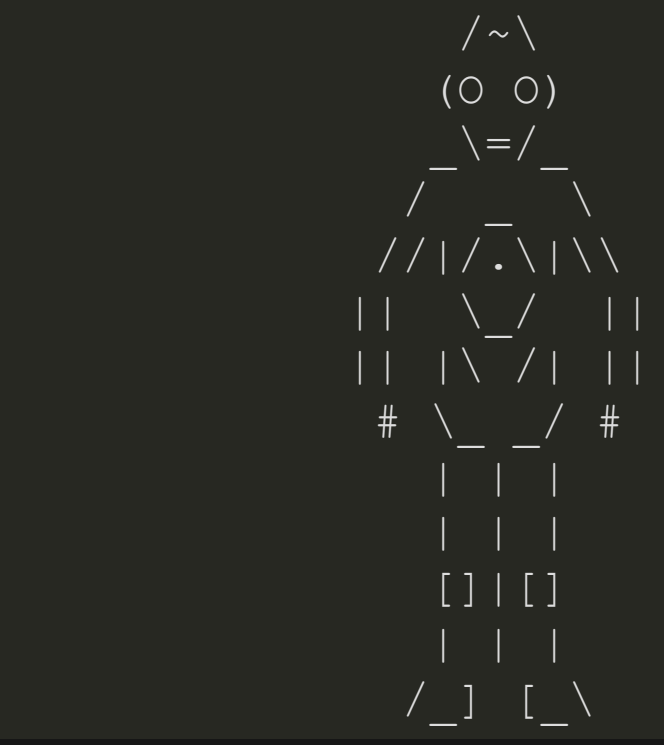
- verificarea datelor de login se face de obicei comparând username-ul și parola cu datele stocate într-un tabel dintr-o bază de date (putem simula verificarea folosind un fișier JSON cu date despre utilizatori)
- în mod normal parola este criptată și se verifică șirul obținut prin criptarea parolei date de utilizator la login cu șirul deja criptat din tabel

SESIUNI: LOGIN/LOGOUT

logout

```
app.get('/logout', function(req, res) {
   req.session.destroy();
   // distrugem sesiunea la intrarea pe pagina de logout
   res.render('pagini/login');
});
```

- presupunem că avem o pagină de logout la care utilizatorul ajunge dând click pe un link/buton
- delogarea presupune distrugerea sesiunii (ștergerea tuturor proprietăților setate în session)



întrebări?