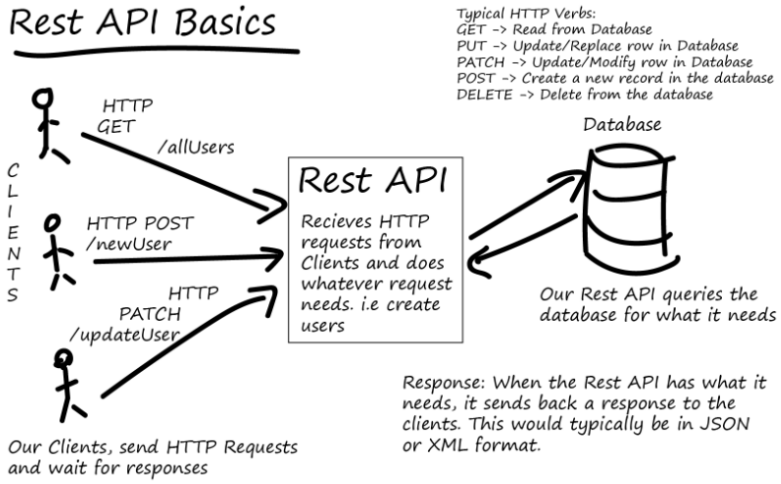
Laboratorul 03: Rest API in ExpressJS

Conceptele fundamentale ale unui Rest API

Un API de tip Rest [https://en.wikipedia.org/wiki/Representational_state_transfer] poate fi privit ca un **blackbox** ce are intrari si iesiri bine definite. Acesta se bazeaza pe 6 concepte fundamentale:

- Arhitectura de tip client-server
- Lipsa starii salvate in contextul cererilor
- Potentialul de caching
- Impartirea pe nivele logice
- Interfata de comunicare uniforma



Rest API in practica

In practica, orice serviciu web care comunica peste **cereri HTTP** si este **stateless** poate fi considerat un Rest API.

Pentru a realiza un astfel de serviciu web, este nevoie sa scrieti un program care se leaga la retea, deci poate fi accesat printr-un ip si un port, expune cel putin o ruta HTTP valida si interactioneaza cu alte servicii sau clienti prin apelarea acelor rute HTTP definite.

De exemplu, script-ul de cateva linii din laboratorul trecut este un Rest API, cu o singura ruta, care returneaza "Hello world!" atunci cand este accesat URL-ul http://ip:port/ [http://ip:port/]

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

const app = express();
app.get('/', (req, res) => {
    res.send("Hello world!");
});

app.listen(3000);
```

Nu exista un limbaj de programare predefinit pentru a scrie Rest API-uri. Daca doriti, puteti scrie unul in C [https://github.com/babelouest/ulfius]. Totusi, exista limbaje de programare si framework-uri care usureaza foarte mult scrierea programelor de genul acesta, precum Django [https://www.djangoproject.com], Go [https://golang.org], ExpressJS [https://expressjs.com], etc….

Avantajul utilizarii unei astfel de tehnologii este ca, datorita simplitatii pe care o propune in scrierea API-urilor, veti putea investi mai mult timp in design-ul aplicatiei, decat in implementarea ei efectiva.

Rest API in ExpressJS

Pe parcursul laboratoarelor 3-7 vom construi impreuna un Rest API de la 0. Ca tehnologie de implementare am ales **ExpressJS**. De ce? Pentru ca este puternic, usor de folosit si este unul dintre cele mai populare [https://www.slant.co/topics/362/~best-backend-web-frameworks] frameworkuri web la ora actuala.

Obiectul principal din ExpressJS

Nucleul oricarui program scris in Express este obiectul obtinut prin apelarea modulului **express**.

```
const express = require('express');
const app = express(); // inima oricarui program ExpressJS
```

Acest obiect, pe care il vom denumi **app** pentru usurinta, contine toate metodele si proprietatile necesare pentru a rula un Rest API.

De exemplu, pentru a lega programul la retea si a-l expune pe un port, este nevoie de apelarea metodei **listen** a obiectului **app**.

```
const port = 3000; //poate fi orice valoare valida pentru un port
app.listen(port, () => console.log(`Salut, rulez pe portul ${port}!`));
```

Rutare in ExpressJS

Rutarea este esentiala intr-un Rest API-ul. In express, obiectul **app** are o serie de metode predefinite pentru a crea rute (url-uri) HTTP. Numele metodelor sunt exact verbele HTTP:

```
app.get('/hello', (req, res) => { // -> creaza o ruta GET cu url-ul '/hello'
  res.send('Hello!');
});

app.post('/marco', (req, res) => { // -> creaza o ruta POST cu url-ul '/marco'
  if (req.body.replyBack === true) {
    res.send('Polo!');
  } else {
    res.send('...');
 }
});

app.delete('/order', (req, res) => { // -> creaza o ruta DELETE cu url-ul '/order'
  res.send('Deleted order');
});

app.put('/user', (req, res) => { // -> creaza o ruta PUT cu url-ul '/user'
  res.send('Changed user');
});
```

Puteti observa ca toate rutele au in comun 2 lucruri:

- **req** - obiectul request care are metode si proprietati ce permit interactiunea cu cererea
- **res** - obiectul response care are metode si proprietati ce permit interactiunea cu raspunsul

Obiectul Request

Obiectul **req [https://expressjs.com/en/4x/api.html#req]** are proprietati si functii pentru interactiunea cu cererea primita. Cele mai folosite 3 tipuri de interactiuni sunt:

- extragerea parametrilor din url (**req.params**)

```
app.get('/books/:id', (req, res) => { // exemplu de url: '/books/3'
  const paramId = req.params.id;
  res.send(`You sent a request with param id ${paramId}`);
});
```

Aveti grija cu tipul parametrilor de cale. Acestia sunt, implicit, siruri de caractere. Daca doriti sa ii folositi intr-un context in care, de exemplu, ei sunt numere (e.g.: **id**), trebuie **convertiti** la tipul respectiv (e.g. **parseInt(id)**)

- extragerea parametrilor din cerere (**req.query**)

```
app.get('/books', (req, res) => { // exemplu de url: '/books?id=3'
  const queryId = req.query.id;
  res.send(`You sent a request with query id ${queryId}`);
});
```

- extragerea continutului dintr-o cerere POST/PUT (**req.body**)

```
app.post('/books', (req, res) => {
  const book = req.body.book;
  res.send(`Your body is ${JSON.stringify(book)}`);
}
```

Continutul primit in corpul unei cereri poate fi de mai multe tipuri, printre care cele mai des intalnite: **form-data**, **x-www-form-urlencoded**, **json**, **xml**. Express ofera implicit suport pentru **x-www-form-urlencoded** si **json**. Pentru celelalte, exista pachete pe NPM care permit interactiunea cu alte tipuri de payload.

Obiectul Response

Obiectul **res [https://expressjs.com/en/4x/api.html#res]** are proprietati si functii pentru interactiunea cu raspunsul ce va fi returnat. Cel mai des, res se foloseste pentru:

- a raspunde unei cererei cu text

```
res.send('Hello!');
```

- a seta statusul HTTP intors in raspuns

```
res.status(404).send('Not found!');
```

- a seta headers HTTP intoarse in raspuns

```
res.set('Content-Type', 'application/json');
```

- a intoarce un json ca raspuns (seteaza implicit header-ul application/json)

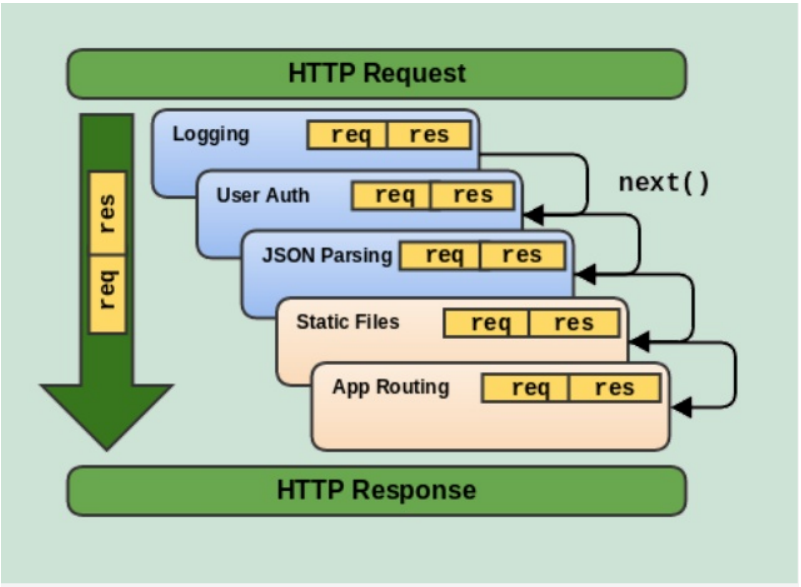
```
res.json(myJsonObject);
```

- a inchide comunicatia REST fara a trimite un raspuns efectiv

```
res.end();
```

Middlewares in Express

Una dintre particularitatile Express este reprezentata de conceptul de **middlewares [https://en.wikipedia.org/wiki/Middleware]**. Orice functie atasata obiectului **app** este un middleware. Functiile care se apeleaza pentru rutele definite sunt, de exemplu, tot middlewares.



Middlewares reprezinta o compozitie *monadica*. In alte cuvinte, puteti sa asociati middlewares cu piese de lego, unde fiecare piesa reprezinta cate o functie. Fiecare middleware vine peste un alt middleware, si tot asa, formandu-se un lant de functii care se executa secvential.

Un middleware este o functie ce contine **obligatoriu** parametrii discutati mai sus, **req** si **res**, precum si un parametru special, **next**.

Parametrul **next** este o functie care, atunci cand este apelata, opreste executia middleware-ului curent si incepe executia middleware-ului urmator din lantul de middleware-uri.

Pentru a adauga un middleware, se pot utiliza doua modalitati:

- utilizarea functiei **use** a obiectului **app** pentru a adauga o functie

```
const myAwesomeMiddleware = (req, res, next) => {
  console.log('Hello from middleware!');
  next();
}

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

// atunci cand este apelata ruta, se va afisa la consola "Hello from middleware", deoarece functia myAwesomeMiddleware se executa prima in lantul de functii
```

- **in-place middleware** sub forma de *functie anonima* atasata unei rute

```
app.get('/hello', (req, res, next) => {
  console.log('Hello from middleware');
  next();
}, (req, res) => {
  res.send('Hello!');
});

// atunci cand este apelata ruta, se va afisa la consola "Hello from middleware", deoarece functia anonima se executa prima in lantul de functii
```

Daca nu executati **next()** in oricare dintre middlewares definite, fluxul de executie nu va avansa si serverul se va bloca

Ordinea in care introduceti middlewares conteaza. Middlewares se executa in ordinea in care sunt introduse

Chiar si functia care incheie lantul de middlewares, adica cea care apeleaza o functie din obiectul **res**, este tot un middleware. Totusi, aceasta nu mai are nevoie sa apeleze **next()**, deoarece este ultima

Middlewares predefinite intalnite des

Cele mai des intalnite middlewares in express sunt cele de securitate, logare sau de manipulare a cererilor:

- Helmet [https://helmetjs.github.io/] este un middleware de securitate, ce adauga o serie de headere importante
- Morgan [https://github.com/expressjs/morgan] este un middleware de logare, ce afiseaza informatii utile despre cereri la consola
- **express.json([options])** este un middleware nativ ce permite extragerea corpurilor **JSON** din cereri HTTP
- **express.urlencoded([options])** este un middleware nativ ce permite extragerea corpurilor **application/x-www-form-urlencoded** din cereri HTTP
- Cors [https://expressjs.com/en/resources/middleware/cors.html] este un middleware ce adauga headerele necesare pentru comunicatii in afara aceluiasi domeniu

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

app.use(express.json());

app.post('/my-awesome-post-route', (req, res) => {
  const body = req.body;
  console.log('Body-ul meu este de tip JSON si a putut fi parsat pentru ca am utilizat middleware-ul express.json(). Body-ul este ${JSON.stringify(body)}');
  res.json(body);
});
```

Middlewares predefinite apeleaza functia **next()** implicit.

Modularizarea rutelor in Express

Chiar daca rutele se pot defini utilizand obiectul **app**, aceasta abordare limiteaza scrierea rutelor in acelasi fisier (in care este definit si app), sau in fisiere separate in care este importat **app**, codul devenind incalcit.

Express ofera obiectul **Router [https://expressjs.com/en/api.html#router]**. Acesta este o extensie a obiectului **app** si este folosit doar pentru a defini rute intr-o maniera modulara.

route.js

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

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

module.exports = router;
```

start.js

```
const express = require('express');
const myAwesomeRoute = require('./route.js');

const app = express();

app.use('/hello', myAwesomeRoute);

app.listen(3000, () => {console.log('App listening on port 3000');});
```

Pentru a folosi o ruta definita cu un obiect **router**, este nevoie ca obiectul sa fie **exportat** din fisierul in care este definit si **importat** in fisierul principal, unde se afla **app**, pentru a fi inclus in lantul de middlewares. Includerea se face folosind **app.use**, **o cale** si **obiectul router importat**.

Router are aceleasi proprietati si functii legate de rutare pe care le are si app

Schema de denumire CRUD

Chiar daca rutele pot avea orice nume, este indicat sa denumiti rutele cat mai simplu posibil, dar totusi sugestiv pentru a se retine contextul in interiorul caruia se opereaza:

De exemplu, sa presupunem ca avem un API ce ofera acces asupra mai multor meniuri de mancare. Putem avea urmatoarele rute:

- **GET /menus** → afiseaza toate meniurile
- **GET /menus/:id** → afiseaza meniul cu id-ul dat ca parametru de cale
- **GET /menus?something=something_else** → afiseaza meniurile care respecta conditia data ca parametru de cerere
- **POST /menus** → introduce un meniu nou
- **PUT /menus/:id** → actualizeaza meniul cu id-ul dat ca parametru de cale
- **PUT /menus?something=something_else** → actualizeaza meniurile care respecta conditiile date ca parametru de cerere
- **DELETE /menus** → sterge toate meniurile
- **DELETE /menus/:id** → sterge meniul cu id-ul date ca parametru de cale
- **DELETE /menus?something=something_else** → sterge meniurile care respecta conditiile date ca parametru de cerere

Urmand aceasta schema, am legat logic toate rutele de mai sus de contextul "meniu", facand codul mai clar si usor de inteles.

Observati ca toate rutele au in comun prefixul **/menus**. Pentru a nu scrie /menus de fiecare data, puteti defini caile din interioul **Ruterului** fara acest prefix si apoi, cand importati meniul, sa puneti calea /menus in apelul functiei **use**.

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

router.get('/', (req, res) => { ... });
router.post('/', (req, res) => {...});
// etc...

module.exports = router;
```

```
const express = require('express');
const menuRouter = require('./menuRouter.js');

const app = express();

app.use('/menus', menuRouter); // -> observati cum /menus este scris o singura data, in calea data functiei use

app.listen(3000, () => console.log('App is listening on port 3000'));
```

Exercitii

Deoarece inca nu am lucrat cu baze de date, pentru acest laborator vom folosi un modul ce va simula o baza de date.

database.js

```
let myArray = [];
let index = 0;

const insertIntoDb = (obj) => {
    const book = { id: index, ...obj };
    index++;
    myArray.push(book);
    return book;
}
const getAllFromDb = () => { return myArray; }
const getFromDbById = (id) => {
    const obj = myArray.find(el => el.id === parseInt(id));
    if (obj) {
        return obj;
    }
    throw new Error(`The object with the id = ${id} does not exists!`);
}
const getFromDbByAuthor = (author) => {
    const books = myArray.filter(el => el.author === author);
    return books;
}
const updateById = (id, payload) => {

    id = parseInt(id);

    const elemIndex = myArray.findIndex(el => el.id === id);
    if (elemIndex > -1) {
        myArray[elemIndex] = {
            id,
            ...payload
        }

        return myArray[elemIndex];
    } else {
        throw new Error(`The object with the id = ${id} does not exists!`);
    }
```

```
}
const removeFromDbById = (id) => {
    const newArray = myArray.filter(el => el.id !== parseInt(id));
    myArray = newArray;
}
const removeFromDbByAuthor = (author) => {
    const newArray = myArray.filter(el => el.author !== author);
    myArray = newArray;
}
const purgeDb = () => {
    myArray = [];
}

module.exports = {
    insertIntoDb,
    getAllFromDb,
    getFromDbById,
    getFromDbByAuthor,
    updateById,
    removeFromDbById,
    removeFromDbByAuthor,
    purgeDb
};
```

Trebuie sa implementati o mica librarie sub forma unui REST Api care implementeaza operatiile CRUD (Create, Read, Update, Delete) pentru "baza de date" propusa mai sus:

- O ruta pentru inserarea unei carti **(1p)**
- O ruta pentru afisarea unei carti dupa id dat ca parametru de cale **(1p)**
- O ruta pentru afisarea unei carti dupa autor dat ca parametru de cerere **(1p)**
- O ruta pentru afisarea tuturor cartilor **(1p)**
- O ruta pentru actualizarea unei carti dupa id **(1p)**
- O ruta pentru stergerea unei carti dupa id dat ca parametru de cale **(1p)**
- O ruta pentru stergerea mai multor carti dupa autor dat ca parametru de cerere **(1p)**
- O ruta pentru stergerea intregii baze de date **(1p)**
- Rutele trebuie definite intr-un ruter extern **(1p)**
- Rutele de inserare si actualizare vor opera pe **JSON**, deci trebuie sa activati parasarea de JSON **(1p)**

Hint! middleware **express.json()**

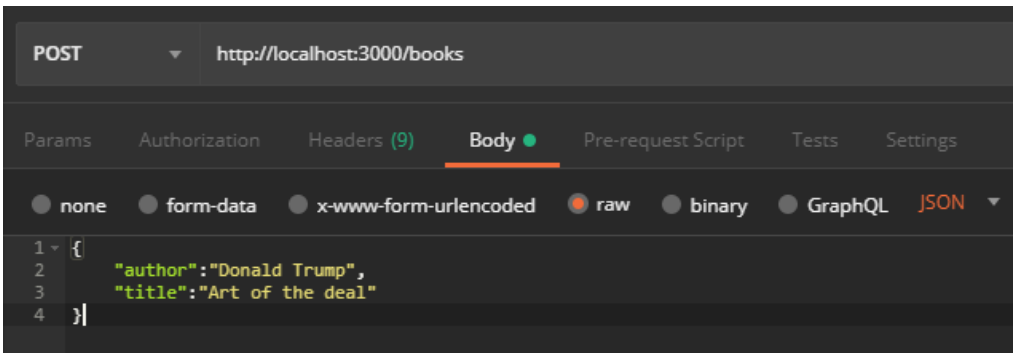
Mentiuni:

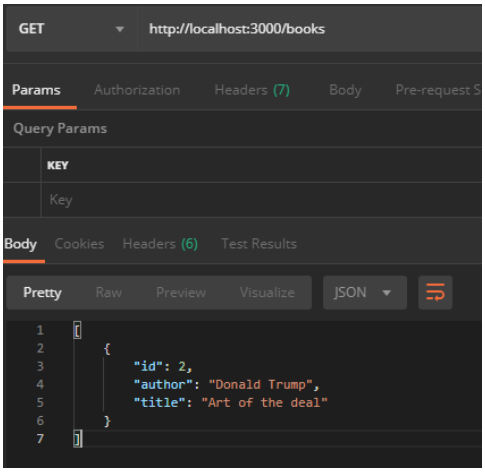
- Este foarte indicat ca rutele voastre sa respecte schema de denumire CRUD
- Este foarte indicat sa tratati erorile utilizand **try...catch** pentru a nu lasa clientul care se conecteaza la API-ul vostru sa vada erorile generate in sistem
- Statusurile HTTP [https://en.wikipedia.org/wiki/List_of_HTTP_status_codes] returnate trebuie sa fie in concordanta cu actiunea petrecuta (201 pt. create, 400 pt. bad request, etc…)

Obiectul cu care veti lucra, va avea formatul:

```
const book = {
  title: String,
  author: String
}
```

Atentie, fiecare carte va avea un **id** generat in "baza de date". Id-ul se genereaza automat, nu trebuie sa il scrieti voi, insa va fi returnat atunci cand se interogheaza "baza de date".





La finalul laboratorului trebuie sa incarcati codul pe Gitlab. Laboratorul se puncteaza dupa ce codul este urcat.

pw/laboratoare/03.txt · Last modified: 2021/03/25 17:17 by alexandru.hogea