

Interogarea serviciilor de date JSON

- în tehnica REST, accesarea datelor se realizează prin **interogări "deghizate" în cereri HTTP** – fiecare tip de cerere e interpretată ca un tip de interogare (GET=SELECT, POST=INSERT etc.)
- în tehnica GraphQL, accesarea datelor se realizează tot **prin cereri HTTP (preferabil de tip POST), cărora li se atașează textul unei interogări** scrisă în sintaxa GraphQL¹.

Tehnica REST

Există o serie de servicii REST² disponibile on-line, cu diverse limitări (de preț, de număr de cereri etc.) sau publice (vezi <https://github.com/toddmotto/public-apis>). Orice serviciu REST oferă o **documentație** prin care ne indică modul în care trebuie să arate cererile HTTP trimise (adresele/rutele, datele pe care le așteaptă, datele pe care le returnează, antetele pe care le acceptă, tipurile de cerere acceptate etc.)

Pentru a fi cât mai flexibili, vom instala propriul server REST gratuit, care va putea fi "interogat" prin cereri HTTP trimise prin mecanismele studiate până acum. E un server cu suport CORS și JSONP, permițând astfel să trimitem cererile din orice pagină creată de noi.

Serverul pentru care optăm pentru exemplificare e **JSON Server**³ și trebuie instalat în **Node.js** (un mediu de execuție ce **permite programarea JavaScript în afara browserului** – pentru back-end, pentru aplicații desktop, pentru componente front-end programate în afara browserului etc.). Pașii instalării și pornirii sunt:

Pas1. Instalați NodeJS de la adresa de mai jos, asigurându-vă că în timpul instalării sunt bifate toate componentele, inclusiv componenta NPM (folosiți versiunea *for Most Users* și nu *Latest Features*, care mai poate avea buguri).

<https://nodejs.org/en/>

Pas2. Așa cum în PHP putem instala librării externe cu Composer (sau în Python cu Pip), în NodeJS ele se instalează cu NPM (inclusiv în instalarea de bază). JSON-Server poate fi folosit atât ca server cât și ca librărie ("package"). Executați în linia de comandă Windows comanda de mai jos, ce se ocupă de descărcarea și instalarea JSON Server în mod global (-g), deci nu va librărie limitată la un anumit proiect:

`npm install -g json-server`

Cei care fac acest seminar în martie 2024 vor instala ultima versiune stabilă cu `npm install -g json-server@0.17.4` (comanda simplă va instala cea mai recentă versiune care în martie 2024 e una de tip alpha, aflată încă în teste, cu bug-uri etc.)

Pas3. Creați un fișier JSON care să conțină datele inițiale ce vor forma baza de date JSON. Salvați acest fișier cu numele `dateinitiale.json` (oriunde, să spunem în `C:\bdjson`) și cu conținutul următor:

¹ Unele servere permit și GraphQL prin GET, însă majoritatea se limitează la POST

² Impropiu numite "API-uri" căci acest termen are o semnificație mult mai largă – un *API* e orice *interfață de programare* (inclusiv clase Java sau mecanisme predefinite într-un limbaj precum fetch, XMLHttpRequest etc.)

³ Detalii complete despre JSON Server puteți găsi la <https://github.com/typicode/json-server>

```
{
  "students": [
    {
      "id": 1,
      "name": "Pop Ion",
    },
    {
      "id": 2,
      "name": "Pop Maria"
    }
  ],
  "courses": [
    {
      "id": 1,
      "title": "Web development",
      "teacher": {
        "name": "Popescu Ion",
        "office": 404
      },
    },
    {
      "id": 2,
      "title": "Java",
      "teacher": {
        "name": "Ionescu Maria",
        "office": 403
      },
    },
    {
      "id": 3,
      "title": "Databases",
      "teacher": {
        "name": "Marian Vasile",
        "office": 401
      }
    }
  ],
  "grades": [
    {
      "courseId": 1,
      "studentId": 1,
      "grade": 7
    },
    {
      "courseId": 1,
      "studentId": 2,
      "grade": 5
    },
    {
      "courseId": 2,
      "studentId": 1,
      "grade": 5
    },
    {
      "courseId": 2,
      "studentId": 2,
      "grade": 5
    },
    {
      "courseId": 3,
      "studentId": 2,
      "grade": 10
    }
  ],
  "faculty": {
    "name": "FSEGA",
    "address": "str. T Mihali 58-60 Cluj Napoca"
  }
}
```

Pas4. Navigați cu linia de comandă Windows până în folderul unde ați salvat fișierul:

```
cd c:\bdjson
```

Pas5. Fiind poziționați în folderul cu fișierul, porniți JSON server precizând să preia datele din fișierul JSON:

```
json-server --watch dateinitiale.json --port 4000
```

Opțiunea *--port* stabilește portul localhost (dacă lipsește portul va fi 3000).

Opțiunea *--watch* face ca orice modificare în fișierul cu date inițiale să se transmită automat la fișierul cu date (dacă îl țineți deschis în VS Code veți vedea actualizările de date pe loc).

Accesați în browser localhost:4000 și veți vedea pagina de întâmpinare JSON Server.

Analizați puțin fișierul JSON, făcând o paralelă cu bazele de date MySQL. Ceea ce avem în fișier este practic echivalentul a trei tabele:

- **students** (cu câmpurile id și name),
- **courses** (cu câmpurile id, title și teacher, ultimul având valori complexe de tip obiect)
- **grades** (cu câmpurile studentId - care funcționează ca o "cheie străină", indicând ID-ul uneia din înregistrările din students; courseId – la fel, cheie străină ce stabilește o relație cu cursurile;

grade – nota pe care studentul referit a luat-o la cursul referit); practic prin "tabelul" grades am realizat o relație many-to-many între studenți și cursuri;

- la acestea se mai adaugă vectorul asociativ **faculty**, care nu e considerat neapărat tabel, cât mai degrabă ca un obiect izolat pentru a vedea că putem interoga nu doar "tabele" ci și alte obiecte pe care le includem pe lângă ele.

Rețineți:

- putem considera fiecare array de pe primul nivel ierarhic al fișierului JSON ca fiind un "tabel" în care cheia primară **obligatoriu trebuie să poarte numele "id"** (cu valori unice!) iar cheile străine poartă un nume format din **singularul numelui tabelului concatenat cu Id**: dacă dorim o relație cu "tabelul" students numele cheii străine trebuie să fie studentId (din acest motiv numele tabelelor trebuie să fie în engleză⁴)
- pe lângă tabele, fișierul JSON poate conține **pe primul nivel ierarhic și obiecte singulare** ce nu pot fi interpretate ca "tabele" (vezi faculty).

Toate entitățile descoperite pe primul nivel ierarhic al fișierul JSON (adică cele 3 tabele și obiectul faculty) sunt considerate "resurse" și vor putea fi **accesate direct prin cereri HTTP**. Tipul cererii HTTP va indica tipul operației, funcționând astfel ca niște interogări conform următoarei echivalări:

- **GET are efectul unui SELECT**
- **POST are efectul unui INSERT**
- **PUT și PATCH au efectul unui UPDATE**
- **DELETE are efectul unui DELETE**

Începem cu câteva cereri GET tastate direct în browser, care practic funcționează ca niște interogări SELECT.

`http://localhost:4000/db`

(afișează tot conținutul bazei de date)

`http://localhost:4000/students`

(afișează datele tuturor studenților)

`http://localhost:4000/students?_sort=name&_order=DESC`

(afișează datele tuturor studenților sortate descrescători după nume)

`http://localhost:4000/faculty`

(afișează datele obiectului faculty)

`http://localhost:4000/students/1`

(afișează datele studentului cu id=1)

`http://localhost:4000/students?name=Pop Ion` (spațiul se va substitui automat)

(afișează datele studentului cu name="Pop Ion"; se pot adăuga mai multe filtre similare separate cu &)

`http://localhost:4000/students?id_ne=2`

⁴ Se folosește o librărie de mapare între singulare și plurale pentru limba engleză:
<https://github.com/geetarista/underscore.inflections>

(afișează datele studentului cu id diferit de 2; codul **_ne** se interpretează ca **not equal**)

`http://localhost:4000/students?q=Maria`

(afișează datele studentului în ale cărui date apare stringul "Maria")

`http://localhost:4000/grades?grade_gte=6&grade_lte=9`

(afișează notele cu `grade` ≥ 6 și ≤ 9 ; observați codurile folosite: **_gte** și **_lte**, căci nu putem folosi caracterele `>` sau `<` în adrese URL)

Observație: JSON Server creează "tabele" doar din entitățile pe care le găsește pe primul nivel ierarhic. Entitățile de pe nivele ierarhice mai adânci (cum este `teacher`) NU vor putea fi accesate independent – putem obține datele lor laolaltă cu restul entității din care fac parte (`courses`), dar nu putem formula o cerere directă care să ne dea doar profesorii⁵. În schimb putem implica datele profesorilor în condiții:

`http://localhost:4000/courses?teacher.office=403`

(afișează datele cursului ale cărui profesor are biroul 403; observați cum informațiile despre profesori pot fi accesate cu ajutorul punctului, însă numai în scopul testării de condiții)

`http://localhost:4000/grades?_expand=student&_expand=course`

(afișează notele, expandând informația despre studenți și cursuri din tabelele relaționate - similar unui JOIN).

`http://localhost:4000/students/1?_embed=grades`

(afișează datele studentului cu ID=1 incluzând notele acestora, detectate tot pe baza corespondenței `students-studentId`; la fel putem obține și cursuri cu notele incluse - tot un JOIN dar în sens invers)

`http://localhost:4000/students/1/grades`

(afișează notele studentului 1, beneficiind de relația `grades-students`)

`http://localhost:4000/students/1/grades?courseId=2`

(afișează nota studentului 1, la cursul 2)

*Observație: Toate aceste "interogări" returnează vectori sau obiecte COMPLETE de pe primul nivel ierarhic al bazei de date JSON. Serverul nu poate returna o valoare simplă (de genul "titlul cursului 1") – aceasta va trebui extrasă de client din rezultatele primite. Aceasta e o situație tipică de **overfetching** (când primim mai multe date decât avem nevoie) - **pentru a depăși limitarea s-a introdus limbajul GraphQL care permite o extragere mai rafinată a datelor.***

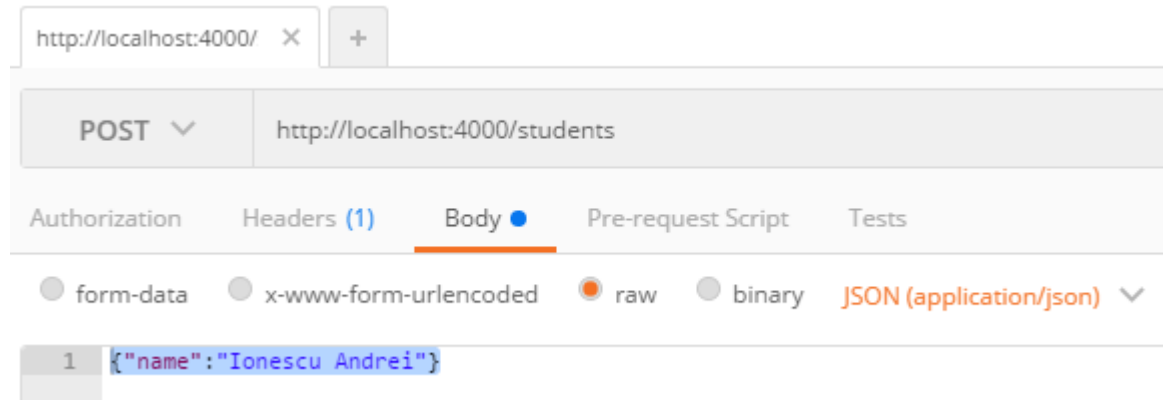
Toate exemplele de până acum au fost cereri GET tastate direct în browser. Pentru alte tipuri de cereri folosiți programul Postman. Operațiile de tip insert/update presupun, evident, atașarea de date în format JSON (și prin urmare declararea Content-Type). Operațiile de ștergere nu au de trimis date, deci nu au nevoie nici de anunțarea Content-Type.

Construiți în Postman o cerere POST cu elementele:

- Adresa `http://localhost:4000/students`
- Metoda POST
- Body: raw JSON cu valoarea `{"name":"Ionescu Andrei"}`.

⁵ Am putea muta profesorii pe primul nivel al fișierului și să construim relații cu `teacherId`, așa cum am făcut și cu `studentId` și `courseId`. Sintaxa JSON ne permite înglobarea multiplă de obiecte unele în altele, însă JSON Server acceptă interogări directe doar pentru entitățile de pe primul nivel ierarhic.

- Headers: Dacă aveți o versiune recentă de Postman, în momentul în care aici setați formatul JSON se setează automat în Headers antetul Content-Type (verificați)



ID-ul nu trebuie precizat căci cheia primară e incrementată automat de JSON-Server.

Construiți și o cerere DELETE pentru a șterge studentul adăugat:

- Adresa `http://localhost:4000/students/3` (remarcați ID-ul studentului, acesta a fost generat de POST-ul anterior!)
- Metoda DELETE
- Body rămâne gol căci nu se trimit date
- Headers: Content-Type se ignoră căci nu se trimit date

Construiți și o cerere PATCH pentru a modifica datele profesorului de la cursul cu ID=2:

- Adresa `http://localhost:4000/courses/2`
- Metoda PATCH
- Body cu datele noi ale profesorului: `{"teacher": {"name": "Moldovan Ioan", "office": 409}}`
- Headers: Content-Type=application/json

Construiți și o cerere PUT pentru a înlocui toate detaliile facultății:

- Adresa `http://localhost:4000/faculty`
- Metoda PUT
- Headers: Content-Type=application/json
- Body cu datele noi ale facultății: `{"country": "Romania", "domain": "Economics"}`

Construiți și o cerere PUT pentru a înlocui toate datele primului student, cu excepția ID-ului (acesta nu poate fi modificat nici cu PUT, nici cu PATCH, așa că nu trebuie inclus în datele trimise):

- Adresa `http://localhost:4000/students/1`
- Metoda PUT
- Headers: Content-Type=application/json
- Body cu datele noi `{"nume de familie": "Popovici", "prenume": "Andrei"}`

Modificați numele de familie al studentului introdus cu ajutorul unei cereri PATCH:

- Adresa `http://localhost:4000/students/1`

- Metoda PATCH
- Headers:Content-Type=application/json
- Body cu datele noi {"nume de familie":"Popescu"}

În final realizăm o operație PUT dintr-o pagină client, prin JQuery, folosind funcția care ne permite să edităm toate configurările cererii, inclusiv Content-Type:

```
<html>
<head>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/2.2.4/jquery.min.js"></script>
<script>
function trimiteCerere()
{
    proprietati={"first name":"Maria","last name":"Pop"}
    dateSerialize=JSON.stringify(propietati)
    configurari={url:"http://localhost:4000/students/2",
        type:"PUT",
        data:dateSerialize,
        contentType:"application/json",
        success:procesareRaspuns}

    $.ajax(configurari)
}
function procesareRaspuns(raspuns)
{
    $("#tinta").html("s-a inserat cu succes studentul "+raspuns["first name"]+" "+raspuns["last name"])
}
</script>
</head>
<body>
<input type="button" onclick="trimiteCerere()" value="Declanseaza cerere"/>
<div id="tinta"></div>
</body>
</html>
```

JSON Server este configurat și pentru tehnica JSON-P. Putem demonstra asta trimițând o cerere GET cu parametrul callback, care să ne listeze toate cursurile din baza de date și profesorii lor:

```
<html>
<head>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/2.2.4/jquery.min.js"></script>
<script>
function solicitare()
{
    adresa="http://localhost:4000/courses?callback=?"
    $.getJSON(adresa,function(raspuns)
    {
        $.each(raspuns,function(indice,curs)
        {
            continutDeAfisat="<div><i>" +curs.title+"</i> predat de "+curs.teacher.name+"</div>"
            $(continutDeAfisat).appendTo(document.body)
        }
        )
    }
    )
}
</script>
</head>
<body>
<input type="button" onclick="solicitare()" value="Declanseaza cerere"/>
<p>Cursurile din baza de date sunt:</p>
</body>
</html>
```

Tehnica GraphQL

Interogările GraphQL ne încurajează să privim datele JSON ca pe "entități conectate într-o rețea de relații" (=graf) și să traversăm acea rețea în orice direcție pentru a obține date conectate. Tehnologia GraphQL este o soluție de compromis între:

(a) servere JSON "clasice" care duc lipsă de un limbaj de interogare propriu și în consecință apelează la tehnica REST, cu limitări legate de situațiile overfetch și underfetch

(b) bazele de grafuri reale care dispun de limbaje de interogare standardizate (vezi SPARQL în serverul RDF4J)⁶.

Presupunând că avem instalat NodeJS (de la exercițiile cu JSON Server), ne folosim și aici de NPM pentru a instala un server pe care putem exersa GraphQL cu un set propriu de date. Instalăm **JSON GraphQL Server** în mod global prin comanda (în linia de comandă Windows):

```
npm install -g json-graphql-server
```

Vom începe prin a crea datele de interogare. Există câteva mici diferențe față de modul de lucru cu JSON Server:

- În JSON Server puteam amesteca "tabele" cu obiecte (acel obiect "faculty"). Aici nu putem avea decât "tabele" (ca liste de obiecte) fără nimic intercalat între ele;
- Convenția pentru chei străine e puțin diferită - cheile străine vor primi nume de forma **course_id**, **student_id**. Desigur, această convenție pentru a exprima "relații" poate să difere de la un server la altul sau poate fi customizată (documentația trebuie să indice astfel de convenții, vezi <https://github.com/marmelab/json-graphql-server>);
- Câmpurile prezente în setul de date inițial vor determina ce câmpuri avem voie să inserăm la operații de inserare! Spre deosebire de JSON-Server care nu restricționa inserarea în niciun fel, **GraphQL impune o schemă pe care o deduce din datele inițiale**, folosită în scopul validării datelor similar cu bazele de date relaționale.

Ținând cont de toate astea, datele trebuie rescrise în felul următor (în fișierul *dateinitialegraphql.json*):

```
{
  "students": [
    {"id": 1,
     "name": "Pop Ion"},
    {"id": 2,
     "name": "Pop Maria"}
  ]
}
```

⁶ Unele servere de baze de grafuri oferă pe lângă limbajul propriu de interogare și o interfață de interogare GraphQL – vezi Neo4J (<https://neo4j.com/developer/graphql/>) sau adaptorul HyperGraphQL (<https://www.hypergraphql.org/>), însă în general GraphQL este mai limitat limbajele native ale acelor servere (SPARQL, CypherQL). De fapt serviciile GraphQL nu sunt propriu-zis servere BD, ci interfețe de interogare prin care servere BD de diverse tipuri (SQL, XML, JSON, grafuri etc.) pot să-și ofere datele în format JSON. În majoritatea implementărilor, interogările GraphQL sunt executate "virtual", adică sunt traduse în limbajul de interogare real al serverului BD ce conține datele (**datele sunt oferite ca "grafuri virtuale"**, ascunzând clientului forma reală a datelor și tehnologia de stocare)

```

    },
    "courses": [
      {
        "id": 1,
        "title": "Web development",
        "teacher": {
          "name": "Popescu Ion",
          "office": 404
        }
      },
      {
        "id": 2,
        "title": "Java",
        "teacher": {
          "name": "Ionescu Maria",
          "office": 403
        }
      },
      {
        "id": 3,
        "title": "Databases",
        "teacher": {
          "name": "Marian Vasile",
          "office": 401
        }
      }
    ],
    "grades": [
      {
        "course_id": 1,
        "student_id": 1,
        "grade": 7
      },
      {
        "course_id": 1,
        "student_id": 2,
        "grade": 5
      },
      {
        "course_id": 2,
        "student_id": 1,
        "grade": 5
      },
      {
        "course_id": 2,
        "student_id": 2,
        "grade": 5
      },
      {
        "course_id": 3,
        "student_id": 2,
        "grade": 10
      }
    ]
  }
}

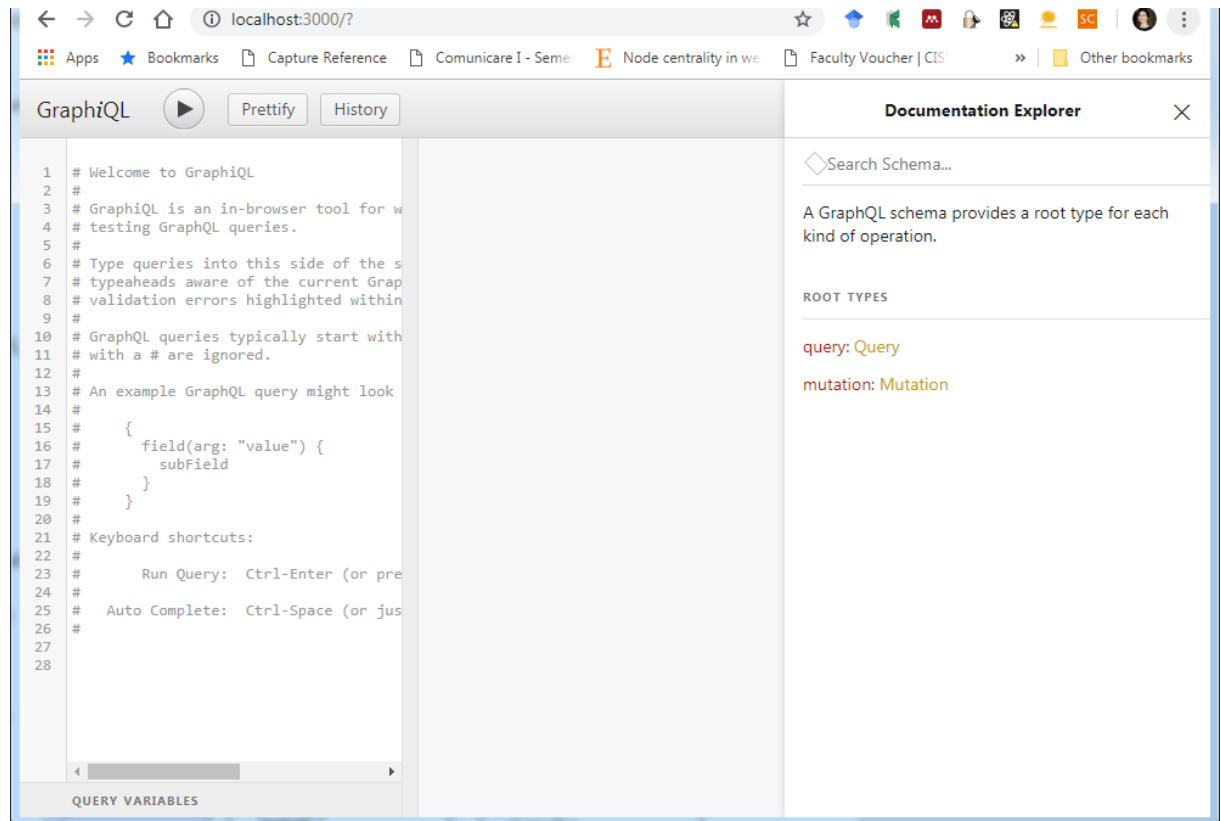
```

Lansăm o fereastră de comandă Windows, navigăm până în folderul unde am salvat fișierul și pornim serverul prin comanda:

```
json-graphql-server dateinitialeqql.json
```

Implicit portul folosit este 3000, astfel că îl vom lansa în browser la adresa *localhost:3000*. Ceea ce apare este interfața **Graphiql**, un instrument asemănător cu Postman pentru exersarea de interogări înainte să le programăm propriu-zis.

Am putea folosi și Postman, dar Graphiql ne oferă și o documentație care ne ajută în construirea interogărilor (panoul din dreapta).



În panoul din stânga putem scrie interogări, în mijloc vor fi afișate rezultatele, iar panoul din dreapta (vizibil după un click pe Docs) oferă **documentația de introspecție**: cine are dificultăți în a-și imagina "graful" datelor interconectate, poate vedea acolo ce interogări se pot construi. Serverele GraphQL în general trebuie să ofere o astfel de documentație clienților care nu sunt deja familiarizați cu datele. Serverele GraphQL de obicei permit două categorii de interogări:

- **Query** sunt operații de citire (similare cu SELECT)
- **Mutation** sunt operații de scriere (inserare, modificare, ștergere).

Operații de citire

În secțiunea Documentation Explorer cu Query putem vedea care sunt interogările permise, sub forma unor șabloane de interogări:

- *șabloane pentru acces individual la o "înregistrare" pe bază de ID (Student, Course, Grade),*
- *șabloane care extrag toate entitățile de un anumit tip, cu posibilitate de filtrare (allStudents, allCourses, allGrades),*
- *șabloane ce realizează calcule de agregare (_allStudentsMeta, _allGradesMeta, _allCoursesMeta oferă fiecare doar o operație count – cine stăpânește NodeJS poate extinde serverul și cu alte operații).*

Următoarea interogare va returna numele studentului cu ID=1:

```
query {
  Student (id:1) {name}
}
```

După cum se vede, o interogare GraphQL e formată din:

- operația dorită (*query* sau *mutation*) cuvântul query pentru operații de citire poate lipsi)
- între acolade, entitatea accesată (*Student*), conform șabloanelor permise din panoul Documentation Explorer
- între paranteze rotunde se pun condiții de filtrare (documentația pune un semn de exclamare în dreptul celor obligatorii, în cazul acesta *ID!*)
- între acoladele interne se indică ce câmpuri să se returneze (aici *name*).

Cuvântul query poate lipsi (nu și la operații mutation):

```
{Student (id:1) {name}}
```

Observați rezultatul JSON, al cărui structură trebuie cunoscută în momentul în care vom programa aceste interogări în scripturi – tot rezultatul e împachetat într-un obiect cu câmpul *data*:

```
{
  "data": {
    "Student": {
      "name": "Pop Ion"
    }
  }
}
```

Un avantaj major față de JSON Server este că rezultatul e mai precis – aici am extras **doar numele**. În JSON Server returnam studentul integral, cu toate atributele sale – limitare numită **overfetching** (când o cerere HTTP obține mai multe câmpuri decât dorește clientul, neavând posibilitatea de a fi selectivă la nivel de câmp).

Aminteam că panoul de introspecție ne ajută să construim gradual o interogare - despre șablonul *Student(id: ID!)* ni se indică faptul că putem solicita și numele, precum și un JOIN cu entitatea *Grades* – care are propriul șablon, propriile câmpuri returnabile, propriile JOIN-uri permise.

id: ID!

name: String!

Grades: [Grade]

Pentru a accesa mai multe câmpuri le enumerăm separate de spațiu:

```
{Student (id:1) {id name}}
```

Pentru a declanșa un JOIN dăm click pe Grade în documentație să aflăm ce anume putem solicita din "tabelul" notelor:

course_id: ID!

student_id: ID!

grade: Int!

Course: Course

Student: Student

Incluzând un JOIN cu șablonul Grades (prin simpla includere a entității), observăm că putem obține nota cu atributul grade (într-o nouă acoladă internă):

```
{
  Student(id: 1) {
    id
    name
    Grades {grade}
  }
}
```

Rezultatul va fi:

```
Response:
{ "data": {
  "Student": {
    "id": "1",
    "name": "Pop Ion",
    "Grades": [
      {
        "grade": 7
      },
      {
        "grade": 5
      }
    ]
  }
}
}
```

Tot din panoul din dreapta aflăm că am putea urmări mai departe alte JOINuri cu Course și Student!

Observație: Ne uităm în acel panou de introspecție când nu avem în minte structura de ansamblu a datelor și preferăm să mergem din aproape în aproape, adăugând treptat noi câmpuri sau JOIN-uri cu noi entități. Aceasta e o situație uzuală când nu cunoaștem dinainte structura informației oferite de serverul GraphQL – astfel de servere trebuie să pună la dispoziție panoul de introspecție pentru a permite clienților să construiască o interogare din aproape în aproape! Un exemplu de server public pe care se vede acest tip de documentație este

<https://countries.trevorblades.com/>

Extinzând mai departe interogarea vom accesa din note și cursurile, de la care preluăm titlul:

```
{
  Student(id: 1) {
    id
    name
    Grades {
      grade
      Course {
        title
      }
    }
  }
}
```

```
}  
}  
}
```

Sau, profesorul:

```
{  
  Student(id: 1) {  
    id  
    name  
    Grades {  
      grade  
      Course {  
        teacher  
      }  
    }  
  }  
}
```

Observați însă că pentru profesor documentația nu ne mai indică o entitate (doar faptul că acolo mai sunt ceva date JSON) așa că nu putem continua lanțul de relații pentru a extrage câmpuri separate din teacher. Ca și în JSON Server, s-au generat entități doar din "tabelele" (arrayul) de pe primul strat ierarhic din datele inițiale. Asta înseamnă că datele profesorului pot fi obținute doar integral (ca în JSON Server) și nu putem fi selectivi cu atributele sale. **Deci avem totuși *overfetching* și în GraphQL, dar la un nivel mai redus și mai ușor de customizat decât în serviciile REST** (nu intrăm aici în detalii privind implementarea serverelor GraphQL, ne interesează doar mecanismele de interogare).

Relațiile dintre "tabele" pot fi navigate în ambele direcții. Putem obține la fel de ușor și studenții pornind de la un curs, traversând relațiile în sens invers curs->note->studenți (pentru cursul cu ID=1 obținem toate notele, apoi studenții care au luat acele note):

```
{  
  Course(id: 1) {  
    title  
    Grades {  
      grade  
      Student {  
        name  
      }  
    }  
  }  
}
```

În JSON Server, pentru acest tip de navigare de relații aveam nevoie de cuvintele cheie *embed* sau *expand* incluse în adresa cererii HTTP - **cu limitarea importantă că nu puteam înlănțui mai multe relații consecutive, ceea ce ne obligă să trimitem mai multe cereri HTTP pentru a traversa un lanț lung de relații. Acel fenomen se numește *underfetching* (când o singură cerere HTTP nu e suficientă pentru a obține toate datele dorite) iar GraphQL oferă o înlănțuire flexibilă de relații indiferent de numărul lor și de direcția de parcurgere⁷.**

Putem chiar să parcurgem relațiile în buclă (după ce am găsit toți studenții care au luat notă la acel curs, să revenim la Grades pentru a obține toate notele acestor studenți). Urmăriți succesiunea termenilor boldați:

```
{
```

⁷ De aceea spunem că în GraphQL datele sunt tratate ca un graf virtual

```

Course(id: 1) {
  title
  Grades {
    grade
    Student {
      name
      Grades {grade}
    }
  }
}

```

Dacă nu vrem overfetching, ci doar să obținem numele studenților de la capătul unui lanț de relații (pornind de la primul curs), din interogare eliminăm câmpurile care nu ne interesează păstrând doar JOIN-urile până la informația de interes:

```

{
  Course(id: 1) {
    Grades {
      Student {
        name
      }
    }
  }
}

```

Similar putem folosi interogările care returnează date de la toate "înregistrările" (și șablonul acestora poate fi consultat în **documentația de introspecție**). Returnăm numele tuturor studenților cu alt șablon decât cel folosit până acum, cel care nu ne impune căutare după ID:

```

{
  allStudents {name}
}

```

Traversăm relația pentru a obține toți studenții și notele lor, aplicând totodată o sortare crescătoare după nume:

```

{
  allStudents (sortField: "name", sortOrder: "asc"){
    name
    Grades {
      grade
    }
  }
}

```

Șabloanele care încep cu **all** oferă clauze suplimentare precum aceste sortField și sortOrder. Și acestea pot fi consultate în panoul de introspecție:

page: Int

perPage: Int

sortField: String

sortOrder: String

filter: StudentFilter

Nu vom insista pe clauzele *page/perPage*, acestea fiind folosite pentru cantități mari de date care dorim să fie returnate paginat (cu un număr limitat de înregistrări "per pagină").

În schimb observați clauza *filter*, ce ne permite să punem condiții asupra entității de pe primul nivel al interogării (din păcate nu și asupra celor cu care se fac JOIN-uri). Un click pe StudentFilter ne va oferi tipurile de condiții permise pentru studenți:

- **q** – căutare după un string,
- **ids** – filtrare la o listă de ID-uri,
- **name** sau alte câmpuri – filtrare după valorile câmpurilor disponibile pentru entitatea curentă,
- **id** – limitare la un ID, asta se putea și cu șablonul Student(id:...).

Următorul exemplu extrage doar studenții în ale căror date apare stringul "Maria" indiferent de câmpul în care apare:

```
{
  allStudents(filter:{q:"Maria"}) {
    name
    Grades {
      grade
    }
  }
}
```

Diferite entități pot avea diferite condiții de filtrare, în funcție de tipul de date. De exemplu allGrades, deoarece conține câmpul numeric *grade* va oferi condiții bazate pe comparații numerice (similare celor din JSON Server: **_gte** reprezintă "greater than or equal"). Următorul exemplu returnează toate notele mai mari decât 8, iar pentru fiecare preia titlul cursului (mergând pe relația cu Course) și numele studentului (mergând pe relația cu Student).

```
{
  allGrades(filter:{grade_gt: 8}) {
    grade
    Course {title}
    Student {name}
  }
}
```

Mai multe filtre pot fi separate cu virgulă. Următorul exemplu ne dă nota studentului 1 la cursul 2:

```
{
  allGrades(filter: {course_id: 2, student_id: 1}) {
    grade
  }
}
```

Tot din panoul de introspecție aflăm că ni se oferă și o serie de interogări care conțin termenul Meta. Acestea acoperă de obicei diverse operații de agregare (count, sum etc.) dar pentru acest server singura operație care ni se oferă este count (se poate verifica printr-un click pe ListMetadata din dreptul acestor categorii de interogări). Următoarea interogare va număra toate cursurile (observați că și aceste interogări permit aplicare de filtrare, sortare, paginare):

```
{
  _allCoursesMeta {
    count
  }
}
```

Uneori este util să redenumim câmpurile din rezultatul JSON returnat, de exemplu atunci când:

- obținem date din mai multe surse și există riscul apariției unor conflicte de nume,
- dorim să redenumim câmpuri conform unei scheme folosite deja în codul sursă al clientului.

În următorul exemplu obținem lista cursurilor, însă câmpurile din rezultat se vor traduce în română:

```
{
  allCourses {
    titlu: title
    profesor: teacher
  }
}
```

Operații de scriere

Pentru operații de modificare asupra datelor se folosesc interogările de tip **mutation**, care și ele sunt documentate în panoul de introspecție. Pentru fiecare tip de entități (*Student*, *Grade*, *Course*) serverul oferă câte șabloane **create**, **createMany**, **update** și **remove**.

Spre deosebire de JSON Server:

- operațiile de scriere se realizează doar în server și nu se salvează în fișierul .json cu datele inițiale; în consecință succesul fiecărei operații trebuie verificat printr-o interogare de citire
- serverele GraphQL impun o schemă la adăugarea de date; deci nu se pot insera alte câmpuri decât cele care s-au detectat în datele inițiale
- versiunile recente ale JSON GraphQL Server oferă și operații de inserare bulk (mai multe înregistrări inserate simultan), scutindu-ne de efectuarea mai multor cereri HTTP pentru inserări multiple.

Adăugați un curs nou. Ca și în JSON Server, id-ul nu se va preciza căci se autoincrementează:

```
mutation {
  createCourse(title: "Business Process Modelling", teacher: "Moldovan Ioan") {id}
}
```

Putem returna mai mult decât ID-ul incluzând între acolade lista de câmpuri de returnat pentru confirmarea succesului inserării (practic avem un INSERT simultan cu un SELECT asupra înregistrării inserate):

```
mutation {
  createCourse(title: "Economics", teacher: "Moldovan Ioan") {id title}
}
```

Afișați toate cursurile pentru a vedea ce s-a adăugat:

```
{
  allCourses {
    id title teacher
  }
}
```

Putem verifica și faptul că nu se pot insera câmpuri neprevăzute în datele inițiale.

Pentru inserări bulk se va folosi șablonul **createMany** care acceptă un array de înregistrări într-un câmp denumit **data**. Șablonul createMany va auto-incrementa ID-uri la fel ca și șablonul create simplu:

```
mutation {
  createManyCourse(data:
```

```

    [{title: "Curs Python", teacher:"Bologa Cristian"},
    {title: "Curs Retele", teacher:"Jecan Sergiu"}]
  } {id title teacher}
}

```

Verificați inserarea (pentru repetarea interogării de verificare, folosiți butonul History din interfața GraphQL):

```

{
  allCourses {
    id title teacher
  }
}

```

Modificați datele cursului cu ID=1

```

mutation {
  updateCourse(id: 1, title: "Semantic Business Process Modelling", teacher: "Moldovan Ioana") {
    id
    title
  }
}

```

Verificați modificarea (ar trebui să aveți interogarea în panoul History):

```

{
  allCourses {
    id title teacher
  }
}

```

Ștergeți cursul cu ID=1:

```

mutation {
  removeCourse(id:1) {id}
}

```

Verificați modificarea (în History):

```

{
  allCourses {
    id title teacher
  }
}

```

Atașarea interogărilor GraphQL la cereri HTTP

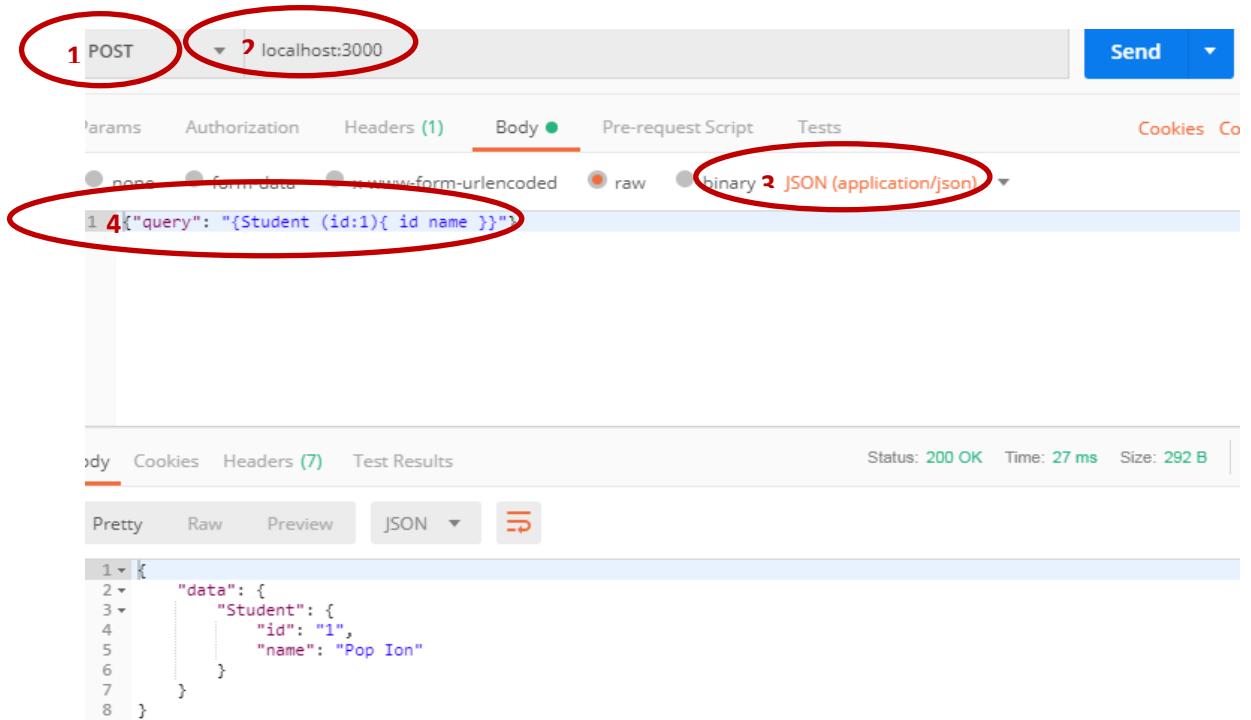
Restartați serverul pentru a reîncărca datele originale (pentru a nu fi afectați de ștergerile realizate). Deși interogările se pot trimite și prin GET, preferința majoritară a serverelor GraphQL este

- să fie trimise prin POST
- în cadrul unui obiect JSON (deci cu Content-Type setat pe "application/json")
- în câmpul query (indiferent că se face citire sau scriere)
- la o singură adresă (nu mai există rute diferențiate pentru fiecare "tabel" separat)

Testăm mai întâi elementele cererii HTTP în Postman:

- (1) setați tipul cererii pe POST
- (2) setați serverul destinație localhost:3000
- (3) în Body, unde setați formatul trimis la raw JSON (să fie activat și antetul Content-Type pentru a declara că se trimite ceva în format JSON)
- (4) scrieți textul interogării în format JSON:

```
{"query": "{ Student(id: 1) { id name }}"}
```



Indiferent dacă e o operație de scriere sau citire, în cererea HTTP interogarea trebuie atașată unui parametru cu numele query. Deci o operație de ștergere în JSON se va scrie astfel:

```
{"query": "mutation{removeCourse(id:1)}"}
```

Și nu astfel:

```
{"mutation": "{removeCourse(id:1)}"}
```

În următorul exemplu interogarea e trimisă cu JQuery printr-o cerere POST (indiferent de tipul operației):

```
<html>
<head>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/2.2.4/jquery.min.js"></script>
<script>
function trimite()
{
    obiectInterogare={query:"{Course(id: 1){title Grades{grade Student {name}}}}"}
    textInterogare=JSON.stringify(obiectInterogare)
    //aici intai avem interogarea ca obiect JS, apoi o convertim in string JSON (e posibil sa o scriem direct ca string JSON)
```

```

        configurari={url:"http://localhost:3000",
                      type:"POST",
                      data:textInterogare,
                      contentType:"application/json",
                      success:procesareRaspuns}
        $.ajax(configurari)
    }
}
function procesareRaspuns(raspuns)
{
    textDeInserat=<h1>Notele la cursul "+raspuns.data.Course.title+" </h1>"
    $("#spatiuRezervat").append(textDeInserat)

    // am inserat intai un titlu ce include denumirea cursului
    // apoi dedesubt realizam un for each (in sintaxa JQuery!) ce va apela functia afisareText pentru fiecare nota returnata

    inregistrari=raspuns.data.Course.Grades
    $.each(inregistrari,afisareText)
}
function afisareText(indice,inregistrare)
{
    textDeInserat="Studentul "+inregistrare.Student.name+" a luat nota "+inregistrare.grade+"<br/>"
    $("#spatiuRezervat").append(textDeInserat)
}
}
</script>
</head>
<body>
<input type="button" onclick="trimite()" value="Declanseaza cerere"/>
<div id="spatiuRezervat"></div>
</body>
</html>

```

Atunci când un script realizează inserări/modificări, adesea datele de introdus sunt disponibile în diverse variabile (de exemplu cu valori provenind din front-end). Ele vor deveni parte din textul interogării prin **concatenare** – însă în operația de concatenare trebuie să fim atenți la caractere nepermise. Două tehnici se pot folosi pentru a evita o concatenare complexă/riscantă:

Tehnica1 – câmpul variables. Obiectul ce conține interogarea poate conține, pe lângă câmpul *query* (cu textul interogării) și câmpul *variables* în care se poate pune un obiect serializat JSON cu diverse variabile-locuitor folosite în textul interogării. În acest caz însă întreaga interogare se împachetează într-o operație parametrizată, ce restricționează tipurile variabilelor ce se vor pasa din exterior spre textul interogării:

```

mutation adaugare($titlu:String!, $detaliiprof:JSON!)
{
    createCourse(title:$titlu, teacher:$detaliiprof) {id title teacher}
}

```

Câmpul `variables` va folosi exact parametri așteptați de operația parametrizată – aceștia se pot numi diferit de câmpurile JSON de inserat efectiv, după cum se vede în acest exemplu, ceea ce e de așteptat (variabilele din care provin datele se pot numi diferit de câmpurile din serverul GraphQL):

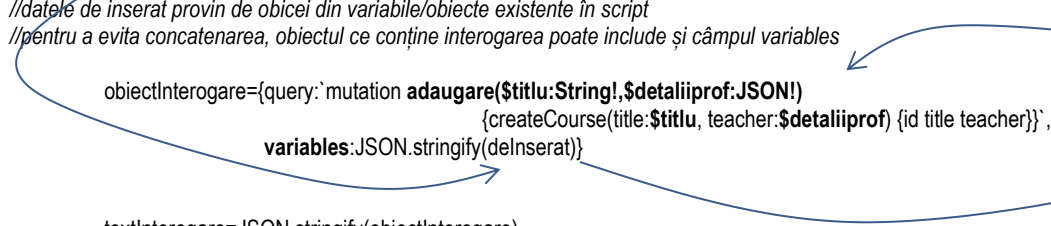
```
<html>
<head>
<script src="http://ajax.googleapis.com/ajax/libs/jquery/2.2.4/jquery.min.js"></script>
<script>
function trimite()
{
  delInserat={ titlu:"C++",detaliiprof:{id:300,name:"Pop Ana",office:300}}
  //datele de inserat provin de obicei din variabile/obiecte existente în script
  //pentru a evita concatenarea, obiectul ce conține interogarea poate include și câmpul variables
  obiectInterogare={query:`mutation adaugare($titlu:String!,$detaliiprof:JSON!)
    {createCourse(title:$titlu, teacher:$detaliiprof) {id title teacher}}`,
    variables:JSON.stringify(delInserat)}

  textInterogare=JSON.stringify(obiectInterogare)
  //pentru a atașa interogarea la cerere am serializat-o cu tot cu variabile

  configurari={url:"http://localhost:3000",
    type:"POST",
    data:textInterogare,
    contentType:"application/json",
    success:procesareRaspuns}
  $.ajax(configurari)
}

function procesareRaspuns(raspuns)
{
  rezultat=raspuns.data.createCourse
  //operațiile de scriere conțin în ultima acoladă câmpuri ce dorim să fie returnate pe post de confirmare!

  textDelInserat="S-a inserat cursul "+rezultat.title+" al profesorului "+rezultat.teacher.name
  $("#spatiuRezervat").append(textDelInserat)
}
</script>
</head>
<body>
<input type="button" onclick="trimite()" value="Declanseaza cerere"/>
<div id="spatiuRezervat"></div>
</body>
</html>
```



Observați și că am folosit apostroful înclinat pentru a delimita textul interogării – asta ne permite să scriem stringuri multi-line în JavaScript, dar are și un beneficiu mai mare, reflectat în cea de-a doua tehnică:

Tehnica2 – interpolare de stringuri (template strings). Independent de GraphQL, JavaScript oferă posibilitatea de a folosi delimitatorii ``{...}`` în combinație cu apostroful înclinat (```) pentru a injecta valori de variabile/expresii într-un string fără a folosi concatenarea explicită. În cazul nostru putem injecta astfel datele necesare în textul unei interogări:

```
titlu="Python"
detaliiprof='{id:300,name:"Pop Ana",office:300}'
obiectInterogare={query:`mutation {createCourse(title:"${titlu}", teacher:${detaliiprof}) {id title teacher}}`}
textInterogare=JSON.stringify(obiectInterogare)
```

Similar, putem injecta și rezultatele în mesajul de confirmare:

```
rezultate=raspuns.data.createCourse
```

textDelInserat='S-a inserat cursul \${rezultate.title} al profesorului \${rezultate.teacher.name}'

Notă: Atenție, nu confundați această utilizare a caracterului \$ cu funcțiile \$() din JQuery sau cu prefixarea numelor de variabile din PHP – nu au nicio legătură!

Discuție

GraphQL nu este o tehnologie specifică JavaScript sau NodeJS. **Despre cereri HTTP știm că pot fi trimise din orice limbaj de programare, spre orice fel de server HTTP, în consecință același lucru devine valabil pentru interogări GraphQL.** Condiția esențială este ca serverul să fie compatibil GraphQL (deci să ne ofere acea documentație de introspecție din care putem afla cum să ne construim interogările).

Rezultatul va sosi de obicei în format JSON, dar nici asta nu e obligatoriu! Chiar dacă interogarea "vede" datele ca JSON, serverul poate fi programat să le returneze în orice format de serializare (cu preferință pentru JSON deoarece astfel datele vor oglindi structura interogării).

În consecință putem considera GraphQL:

- din punctul de vedere al clientului e **un limbaj universal de interogare ce exploatează relațiile existente între date**
- din punct de vedere al serverului e **un mecanism de construire de grafuri virtuale peste relații existente** în sursele de date reale din spatele serverului

Am arătat că GraphQL rezolvă adesea problemele de overfetching sau underfetching care apar în cazul serviciilor de tip REST. Totuși, și flexibilitatea GraphQL este limitată (prin comparație cu SQL) de exemplu: imposibilitatea de a include filtre logice complexe (cu operatori logici AND, OR etc.), filtre simultane pe mai multe câmpuri ale mai multor entități, subinterogări, grupare, tipuri diferențiate de JOIN etc.

Încercați să realizați pe datele noastre o interogare relativ simplă precum "returnează nota lui Pop Ion la cursul de Java" și ajungem imediat într-o situație de over/underfetching. O posibilă abordare este să obținem printr-o cerere *toate notele lui Pop Ion...*

```
{
  allStudents(filter: {name: "Pop Ion"}) {
    Grades {
      grade
      Course {
        title
      }
    }
  }
}
```

...apoi în datele ce sosesc să parcurgem notele până găsim cursul de Java (situație tipică de **overfetching**)

Sau, putem obține printr-o interogare dublă *ID-urile lui Pop Ion și Java...*

```
{
  allStudents(filter: {name: "Pop Ion"}) {id}
  allCourses(filter: {title: "Java"}) {id}
}
```

...iar apoi folosim ID-urile într-o nouă interogare, deci o nouă cerere HTTP (situație tipică de **underfetching**):

```
{
  allGrades(filter: {course_id: 2, student_id: 1}) {
    grade
  }
}
```

Problema e rezolvabilă pe diverse căi, dar nu cu ajutorul limbajului de interogare propriu-zis (ar fi nevoie de subinterogări, filtre mai sofisticate, mecanisme de înlănțuire a interogărilor). Posibile soluții:

(a) Unele librării (vezi Apollo) permit compunerea/înlănțuirea de interogări ori aplicarea de filtre suplimentare în scriptul ce trimite interogarea. Se poate obține un cod mai compact dar adesea tot va trimite mai multe cereri, ascunzând acest detaliu programatorului prin faptul că i se oferă o sintaxă mai comprimată;

(b) Dacă avem control asupra serverului, putem customiza serverul pentru a include opțiuni suplimentare, cu filtre mai complexe decât cele exemplificate – serverul aici folosit, JSON-GraphQL-Server, este de altfel un pachet NodeJS open source pe care îl poate adapta oricine propriilor nevoi adăugându-i noi metode de acces ("resolvere") la datele reale (aici stocate într-un fișier text, dar ar putea fi stocate oriunde). Asta înseamnă și că interogările GraphQL pot varia radical de la un server la altul (totuși tipurile de interogări aici exemplificate sunt destul de uzuale).

Un limbaj de interogare a grafurilor comparabil cu SQL (în ce privește statutul de standard și complexitate), și care nu suferă de problemele over/underfetching pe grafuri este SPARQL, folosit în bazele de grafuri "reale". Reamintim că în GraphQL e foarte probabil ca datele interogate nici măcar să nu fie grafuri - pot fi tabele, XML etc, doar clientul le percepe ca "grafuri virtuale" datorită interfeței asigurate de serverul GraphQL. Într-o bază de grafuri reale, interogările nu depind de modul în care e programat serviciul/serverul – limbajul de interogare e standardizat, nu sunt necesare librării de customizare a serverului, iar serverul e capabil să returneze și subgrafuri.

În exemplele de mai jos exprimăm datele legate de nota la Java a lui Pop Ion în două moduri, indicând în mod grafic cum arată graful, precum și interogarea SPARQL care identifică exact nota dorită și nimic mai mult, evitând situația overfetching/underfetching de care serverul GraphQL încă suferă:

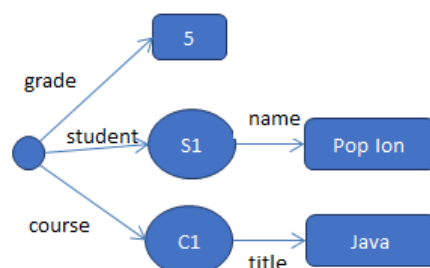
Varianta A:

Cum se scriu datele:

```
[ :grade 5; :student :S1; :course :C1 ].
:S1 :name "Pop Ion".
:C1 :title "Java".
```

Interogarea:

```
SELECT ?x
{
  [ :grade ?x; :student/:name "Pop Ion"; :course/:title "Java" ]
}
```



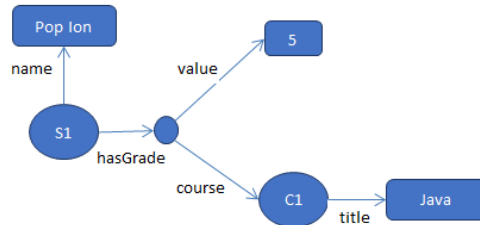
Varianta B:

Cum se scriu datele:

```
:S1 :name "Pop Ion"; :hasGrade [:value 5; :course :C1].  
:C1 :title "Java".
```

Interogarea:

```
SELECT ?x  
{  
  "PopIon" ^:name/:hasGrade [:value ?x; :course/:title "Java"].  
}
```



Putem concluziona că:

- SPARQL aplică același principiu ca și GraphQL de a oglindi structura datelor indicând la nivel de câmp ce se dorește, plus posibilități de filtrare mult mai flexibilă și navigare prin graf în orice direcție (fără a necesita vreo customizare/schemă la nivel de server);
- comparativ cu SQL are același statut de standard (comenzi cu o semnificație precisă, se folosesc la fel pe orice server).

Trimiterea interogărilor SPARQL se realizează similar cu trimiterea interogărilor GraphQL, prin atașarea lor la o cerere HTTP; sau, se pot folosi librării dedicate care ascund operația respectivă (EasyRDF în PHP, SPARQLWrapper în Python etc.).

Următoarele tutoriale se vor axa pe folosirea acestui limbaj demonstrat pe serverul de grafuri RDF4J.

Cereri HTTP paralele trimise din PHP

Mai jos aveți un exemplu de script care solicită date simultan de la cele două servere studiate în acest tutorial. Pentru a putea să funcționeze deodată, presupunem că JSON-Server a fost pornit pe portul 4000, iar JSON-GraphQL-Server pe portul său default 3000.

Am prezentat anterior librăria Guzzle pentru cereri HTTP programate în PHP. Aceasta are avantajul major că permite execuția paralelizată a cererilor. Salvați următorul script în folderul XAMPP unde aveți instalată librăria Guzzle (posibil ca acela să fie folderul site1, unde am folosit anterior librăria pentru scripturi proxy).

```
<?php  
require "vendor/autoload.php";  
$client=new \GuzzleHttpClient();  
  
$interogareSerialized=["json"=>["query"=>"{allCourses{teacher}}"]];  
$antet=["headers"=>["Content-Type"=>"application/json"]];  
  
$cereri=[  
    "cerere1"=>$client->getAsync("http://localhost:4000/students"),  
    "cerere2"=>$client->postAsync("http://localhost:3000",$interogareSerialized,$antet)  
];
```

```
$rezultate=\GuzzleHttp\Promise\unwrap($cereri);  
  
print $rezultate["cerere1"]->getBody();  
print "<br/>";  
print $rezultate["cerere2"]->getBody();  
?>
```

Alternativ, în loc de `unwrap()` se poate folosi `settle()` dacă dorim ca execuția să se realizeze chiar dacă unele din cererile paralele eșuează.

Documentația oficială `GuzzleHttp`, unde găsiți numeroase exemple de cereri `Guzzle`, poate fi consultată la adresa:

- <http://docs.guzzlephp.org/en/latest/quickstart.html>