AJAX (Cereri HTTP trimise din front-end JavaScript)

Nu vom reproduce tot codul sursă din exemplele oferite în arhivă (cu câteva excepții), acest document e doar un sumar cu principalele idei care trebuie să reiasă din fiecare exemplu.

Exemplul A: PHP clasic

Descriere: Formular PHP creat în stil "clasic", trimis cu butonul SUBMIT:

- se completează formularul complet cu codul poștal 100, pagina de răspuns va afișa o adresă din Cluj; orice alt cod va fi considerat incorect (nu s-a mai implementat căutarea codului într-o bază de date, presupunem că a fost găsită acea adresă)
- *nu există o diferențiere clară între front-end și back-end* (e un singur fișier, formularul se trimite lui însuși, având o parte PHP care verifică dacă pagina e la prima afișare sau dacă e la afișarea de după primirea formularului)

Neajunsurile butonului SUBMIT: oferă un mecanism default de cerere HTTP cu următoarele limitări:

- **trimite tot** ce găsește în câmpurile formularului (uneori dorim să trimitem și ceva plus, alteori dorim să trimitem doar anumite părți din formular, uneori dorim să trimitem anumite câmpuri fără să așteptăm completarea integrală a formularului, uneori dorim să trimitem date care n-au fost culese prin formular)
- consecința apăsării butonului SUBMIT e **încărcarea unei pagini integrale**; în cazul nostru dorim să se actualizeze doar caseta roșie cu adresa, căci restul paginii e deja afișată
- trimite datele în **formatul default** Query String/Form Data; uneori dorim să optăm pentru alte formate – JSON, XML, CSV etc.
- poate trimite datele prin **cereri de tip GET sau POST**; uneori dorim să le trimitem prin alte metode (PUT, PATCH, DELETE etc.), sunt servere care pot impune asta
- declanșeaza trimiterea prin click; uneori dorim ca alt eveniment să declanșeze trimiterea

Pentru a depăși limitările butonului SUBMIT trebuie să știm programa noi o cerere HTTP în toate detaliile sale – deocamdată în JavaScript (apoi acel pattern va fi recunoscut și în alte limbaje).

Exemplul B: AJAX cu XMLHttpRequest

Descriere:

- codul poștal e trimis acum imediat după completarea sa (la trecerea în următoarea casetă, evenimentul **onblur**), fără a aștepta completarea întregului formular
- imediat apare adresa/mesajul în caseta roșie, fără a se reîncărca integral pagina
- avem o separare clară între front-end (fișierul HTML) și back-end (fișierul PHP); fișierul PHP nu mai are responsabilitatea de a construi o pagină Web, ci doar de a returna datele ce i s-au cerut (se comportă ca o funcție care așteaptă un input și returnează un output)

Acesta e principalul exemplu care detaliază toți pașii mecanismului HTTP într-un exemplu minimum viabil. Exemplele următoare se folosesc de unele simplificări ale codului (ex. librăria JQuery), însă e importantă studierea mecanismului complet, pentru a-l putea recunoaște și în alte limbaje (unde nu există librăria JQuery – deși diverse simplificări sunt oferite și de diverse alte librării).

Mai jos urmăriți comentariile și săgețile pentru a identifica pașii mecanismului HTTP:

1. Front-end: definirea evenimentului care să declanșeze cererea (ONBLUR)
2. Front-end: construirea și trimiterea cererii, cu date atașate în format QueryString
3. Back-end: recepționarea cererii în back-end
4. Back-end: returnarea unui răspuns de către back-end (format default)
5. Front-end: o funcție așteaptă răspunsul și îi verifică succesul
6. Front-end: aceeași funcție (sau alta) se folosește de răspuns (aici o simplă afișare)

FRONT-END

```
<html>
<head>
<title>Formular</title>
<script type="text/javascript">

function trimite(codpostal)
   {
   cerere=new XMLHttpRequest()
//crearea cererii
   cerere.onreadystatechange=procesareRaspuns
//rezervarea funcției ce se va ocupa de răspuns
   cerere.open("GET","2adresa.php?cod="+codpostal)
//configurarea cererii (tip, destinatie) + serializarea datelor în format default (QueryString)
   cerere.send(null)
//trimiterea cererii (null pt că datele sunt deja lipite la adresă)
   }


function procesareRaspuns()
   {
   if ((cerere.readyState==4)&&(cerere.status==200))
//raspunsul e OK si a sosit complet
      {
      raspuns=cerere.responseText
//extragerea raspunsului (nu chiar deserializare caci raspunsul e tratat ca text brut)
      document.getElementById("RezervatAfisare").innerHTML=raspuns
//folosirea raspunsului
      }
   }

</script>
</head>

<body>
   <h1>Introduceti datele</h1>
   <form action="prelucrareform.php" method="post">
      CodPostal <input type="text" id="codpostal" onblur="trimite(this.value)"><br/>
      Nume <input type="text" id="nume"><br/>
      Prenume <input type="text" id="prenume"><br/>
      <input type="submit" value="Trimite formular">
   </form>

   <div id="RezervatAfisare" style="border:1px solid red; height: 300px; width:400px"></div>

   </body>
</html>
```

BACK-END

```
<?php
// QueryString e deserializat automat in $_GET
if ($_GET["cod"]=="100")
   print "Cluj Napoca,Cluj,Aleea Azuga";
else
   print "cod incorect";
// raspuns returnat in format default
?>
```

Observații: exemplul e redus la minimul necesar pentru a evidenția pașii mecanismului HTTP. În practică sunt și alte aspecte de avut în vedere:

- *vizibilitatea variabilei **cerere**: acum a fost lăsat ca variabilă globală ca să fie văzută de funcția **procesareRaspuns** (nerecomandat, risc de conflict cu alte cereri din aceeași pagină, cu librării importate[1]); variabila se poate declara ca locală, dacă funcția procesareRaspuns devine anonimă (atașându-i corpul direct în onreadystatechange):*

```
cerere.onreadystatechange = function() {..procesarea raspunsului..}
```

sau în sintaxa simplificată de tip "arrow function" (se pune o săgeată între argumente și corpul funcției, fără a mai scrie function):

```
cerere.onreadystatechange = ()=>{..procesarea raspunsului..}
```

o funcție atașată în acest mod unui eveniment va avea acces la variabilele locale de pe același nivel cu evenimentul.

- *tratarea **eșecului cererii** (un cod de eșec ar trebui returnat pe ramura else din back-end, o ramură else în if-ul din front-end ar trebui să preia mesajul de eșec)*

- *monitorizarea **progresului cererii** (cererea are o serie de evenimente intermediare ce se pot trata, de exemplu la upload/transfer de fișiere mari, când dorim să afișăm un progres – vezi evenimentele onprogress, onloadstart etc. pentru a semnala în front-end diverse stări intermediare ale transferului[2])*

Exemplul C: Același comportament ca în B, dar realizat cu JQuery

Se observă simplificarea sintaxei de către JQuery:

- **$.get** realizează toți pașii necesari trimiterii cererii
- **$()** în loc de getElemenyById (chiar mai puternică, permițând orice selectori CSS)
- **html()** în loc de innerHTML
- funcția procesareRaspuns primește **raspuns** ca argument default (ea nu poate fi apelată cu argumente! de ce?[3])
- toate aceste facilități necesită importarea librăriei JQuery cu SCRIPT SRC

[1] ... sigur, riscul nu există dacă numele sunt în română; în practică le lăsăm internaționale, în tutoriale le scriem în română pentru a vedea clar unde avem libertatea de a denumi și unde folosim obiecte built-in
[2] Documentația clasei: https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest
[3] Astfel de funcții se numesc funcții-**callback** (caracteristici: nu se pot apela cu argumente date de programator ci primesc argumente predefinite, nu returnează nimic, momentul execuției nu e decis de linia pe care apare în codul sursă, ci de funcția care le rezervă => asta înseamnă **programare asincronă**!)

Exemplul D: Același comportament ca în B/C, dar realizat cu fetch

Tehnica fetch este un upgrade recent al limbajului JavaScript (care încearcă treptat să elimine folosirea de librării precum JQuery).

Se observă că funcția de procesare a răspunsului e aici desfăcută în două funcții ce se apelează înlănțuit:

```
fetch(...).then(extragereRaspuns).then(folosireRaspuns)
```

- prima funcție se ocupă de verificarea confirmării din partea serverului (codul OK) și extragerea/deserializarea răspunsului (aici ca text simplu)
- a doua funcție folosește efectiv a răspunsului
- înlănțuirea e asigurată de **then()** care transferă automat outputul funcției precedente în inputul funcției ce o urmează[4]
- tutorialele on-line prezintă adesea tehnica fetch() într-o sintaxă mai comprimată, aplicând unele simplificări sintactice introduse recent în JavaScript:

- cu funcții anonime (tot corpul funcției inclus ca argument):

```
fetch(...).then(function(raspuns) {...}).then(function(text) {...})
```

- arrow functions, evitând și scrierea cuvântului function:

```
fetch(...).then(raspuns=>{ ...}).then(text=>{...})
```

Exemplul C2: Versiune mai realistă la C, tot bazat pe JQuery

Modificări pe back-end, față de versiunea C:

Gestiunea eșecului:

- back-endul diferențiază cazul de succes (codul poștal 100) de cazul de eșec generând în al doilea caz un cod de eșec HTTP (404 Not Found[5]); atenție, eșecul HTTP nu înseamnă că există erori în back-end sau că răspunsul nu se poate returna, ci că se returnează un răspuns ce nu va fi considerat satisfăcător de client (nu a găsit codul poștal)
- în front-end, mesajul de eșec e afișat o funcție nouă, procesareEsec() ce se rezervă cu funcția fail() – astfel se separă clar tratarea succesului și eșecului cererii

Folosirea antetelor HTTP:

- front-endul anunță că preferă să primească JSON (cu al 4-lea argument din $.get se setează de fapt antetul Accept al cererii);
- back-endul anunță (în Content-Type) că în caz de succes returnează JSON iar în caz de eșec Plain Text

Serializarea/deserializarea:

[4] E vorba de mecanismul Promise, implementat recent în JavaScript pentru a înlănțui funcții cu succes incert (tot un pattern al **programării asincrone** ca și funcțiile-callback)
[5] codurile HTTP se pot consulta la https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

- în ce format trimite front-endul codul postal? deși arată ca JSON, următoarea linie e de fapt un obiect JavaScript:

```
dateDeTrimis={cod:codpostal}
```

- ar fi JSON dacă ar avea următoarele forme (formatul JSON e textual, un string care mimează o structură obiectuală!):

```
jsonDeTrimis='{"cod":100}'
jsonDeTrimis='{"cod":'+codpostal+'}'
```

- ar mai fi JSON și dacă s-ar obține prin serializarea obiectului JavaScript:

```
jsonDeTrimis=JSON.stringify(datedeTrimis)
```

- o sursă frecventă de erori la începători este când nu le e clar dacă lucrează cu un string JSON sau cu un obiect JavaScript; așadar, să nu cădem în capcana de a crede că aici front-endul trimite JSON, e un obiect JavaScript pe care JQuery îl va serializa în formatul default QueryString

- back-endul returnează adresa în format JSON (serializat dintr-un array PHP), nu o mai returnează ca text simplu; asta permite ca front-endul să poată diferenția între oraș, stradă, județ (în cazurile anterioare adresa primită ca Plain Text era afișată integral, fără posibilitate de diferențiere între câmpuri[6])

Observații:

- *Orice limbaj oferă funcții de serializare JSON (conversie din obiect/array în acel string JSON):*

***JSON.stringify()** în JavaScript, **json.encode()** în PHP, **json.dumps()** în Python etc.*

- *Orice limbaj oferă și funcția inversă, de deserializare (din textul JSON primit în obiect/array manipulabil):*

***JSON.parse()** în JavaScript, **json.decode()** în PHP, **json.loads()** în Python etc.*

Exemplul C3: Aproape ca C2, tot cu JQuery

Singura diferență față de C2 e că aici adresa primită de la server nu e afișată în caseta roșie, ci într-un pop-up cu următorul comportament:

- apare automat la sosirea răspunsului
- conține un buton care îl închidee

Pop-ul există de la bun început (DIV-ul rezervat pentru afișate), însă apariția și dispariția sa sunt controlate prin manipularea proprietății CSS **visibility**:

- în funcția de afișare a răspunsului (sau eșecului) e făcut să apară

[6] nici aici nu le tratăm diferit, tot împreună sunt afișate, dar măcar le avem în variabile diferite, e asigurată distincția lor

- în funcția eliminaPopup e făcut să dispară

În general pentru a programa acest gen de comportament front-end apelăm la librării specializate pe efecte de front-end (JQuery UI, Bootstrap). Nu face obiectul acestui curs, aici punem accent pe cum obținem datele.

Exemplul E: Cerere POST trimisă prin XMLHttpRequest (formular integral, format default)

Diferențe de funcționare față de versiunea GET (exemplul B):

- formularul e trimis acum integral, nu doar codul poștal
- formularul duce lipsă de buton SUBMIT, trimiterea fiind programată de noi printr-o cerere POST; trimiterea e declanșată de mouseover pe a doua casetă roșie din pagină
- back-endul răspunde cu un text de confirmare a primirii datelor, inserat în prima casetă roșie de sub formular

Diferențe față de Exemplul B:

- nu avem propriu-zis un formular, nu există tagul FORM! sunt dor casete INPUT
- în lipsa butonului SUBMIT, avem de programat noi serializarea datelor, un proces anevoios în JavaScript "clasic": un ciclu FOR pentru a le culege din formular, apoi concatenarea lor în format QueryString, cu codificare URI a caracterelor nepermise de formatul QueryString)
- cu POST, datele se trimit în send() (în locul acelui null folosit la cereri GET)
- cu POST avem de declarat formatul de serializare folosit (www-form-urlencoded e de fapt formatul QueryString)

Am putea fi tentați să spunem că nu merită acest efort în comparație cu folosirea butonului SUBMIT, însă următoarele exemple aduc o serie de simplificări la efortul de preparare a datelor.

Exemplul F: Cerere POST trimisă prin JQuery (formular integral, format default)

Aceeași funcționare ca exemplul E, însă cu simplificări aduse de JQuery la culegerea datelor din formular:

- tot efortul de culergere și serializare (pentru care în E am scris propria funcție) e realizat acum de o funcție oferită de JQuery (pentru asta a fost nevoie să reintroducem tagul FORM, căci funcția serializează un formular întreg!):

```
$("#formular").serialize()
```

- formatul aplicat de această funcție e cel default (QueryString/FormData), de care se folosește JQuery când trimite cereri cu $.get și $.post; fiind POST, datele nu se mai lipesc la adresă ci apar ca argument separat:

```
$.post(adresa, dateDeTrimis, procesareRaspuns)
```

Exemplul G: Cerere POST trimisă prin fetch (formular integral, format default)

Aceeași funcționare ca exemplul E/F, cu simplificări aduse de upgradeuri JavaScript recente (reamintim tendința de eliminare a nevoii pentru JQuery):

- ceea ce JQuery realiza cu serialize() acum se face în JavaScript dacă unei cereri fetch() i se atașează datele sub forma unui obiect FormData

```
obiectFormular=new FormData(document.getElementById("formular"))
fetch(adresa,{method:"post",body:obiectFormular}).then(...).then(...)
```

- atenție, un obiect FormData în sine nu asigură serializare (nu putem să-l afișăm direct), e un obiect ce conține datele formularului (se pot parcurge cu FOR); fetch() îi va aplica automat serializarea default QueryString/FormData

În exemplul G am inclus și varianta fetch cu sintaxa comprimată.

Exemplul H: Similar cu E (POST cu XMLHttpRequest) dar formularul e serializat ca JSON

Modificare majoră la back-end:

- PHP nu poate prelua JSON din $_POST/$_GET (acelea sunt pentru date primite ca QueryString/FormData); în consecință PHP va citi datele din interfața standard de intrare a PHP-ului:

```
$JSONsosit=file_get_contents("php://input");
```

- Asta înseamnă că nici deserializarea nu e aplicată automat, o facem noi cu json_decode, iar apoi accesăm câmpurile pe cale obiectuală

Modificări front-end:

- Content-Type va indica faptul că trimitem JSON
- Avem iar de preparat datele în vederea trimiterii. Am reintrodus tagul FORM să nu le culegem câmp cu câmp, deci ne folosim de simplificările aduse de obiectul FormData (văzut deja la exemplul cu fetch). Problema e că obiectul FormData nu poate fi serializat direct în JSON – el e optimizat pentru a fi serializat ca QueryString, ceea ce se face cu ajutorul constructorului URLSearchParams (fetch face automat asta):

```
obiectFormular=new FormData(document.getElementById("formular"))
QueryString=new URLSearchParams(obiectFormular)
```

Cu alte cuvinte, în acest obiect vom găsi un array de perechi [[cheie,valoare],...] și nu un obiect de forma {cheie:valoare,...} gata de transformat în string JSON. O soluție ar fi să-l parcurgem cu un ciclu foreach(), construind treptat atribute obiectuale din perechi [cheie, valoare], însă JavaScript oferă deja o funcție pentru acest tip de restructurare – Object.fromEntries:

```
objectFormular=new FormData(document.getElementById("formular"))
obiectRestructurat=Object.fromEntries(obiectFormular)
JSONDeTrimis=JSON.stringify(obiectRestructurat)
```

Exemplul I: Similar cu H dar prin JQuery (POST cu formular serializat ca JSON)

Componenta back-end rămâne cu modificările din exemplul precedent, deci PHP așteaptă formularul ca JSON și nu prin $_POST.

Pe partea front-end apar două modificări majore:

- serializarea formularului în format JSON ridică o provocare similară și în JQuery (precum și cu fetch, căci ambele aplică implicit serializare QueryString/FormData); reamintim că în JQuery am cules datele din formular cu serialize(), care ne oferea un QueryString. Din păcate nu ni se oferă ceva similar pentru a converti un formular în JSON, însă nu ne împiedică nimic să îmbinăm JQuery cu facilitățile JavaScript native. În consecință serializarea JSON arată astfel[7]:

```
objectFormular=new FormData($("#formular")[0])
obiectRestructurat=Object.fromEntries(objectFormular)
JSONdeTrimis=JSON.stringify(obiectRestructurat)
```

Indicele [0] e necesar pentru că funcția $() returnează de fapt un array cu tagurile găsite, chiar dacă am făcut selecția după ID!

- a doua complicație e că trimiterea JSON necesită și aici setarea antetului Content-Type, pe care $.post nu o face în JQuery; dacă dorim acces și la sertarea de antete, precum și la numeroase alte configurări ale cererilor HTTP, vom avea libertate mai mare folosind funcția generală $.ajax, ce primește ca argument un obiect cu toate configurările – printre acestea se poate face și setarea Content-Type (de asemenea, tot asta vom folosi pentru orice alte tipuri de cereri – PUT, PATCH etc.):

```
    configurari={url:adresa,
                 type:"POST",
                 data:JSONdeTrimis,
                 contentType:"application/json",
                 success:procesareRaspuns}
    $.ajax(configurari)
```

Notă: În general acest efort de a serializa un formular în JSON nu prea se practică (când avem de trimis un formular complet e mai facilă trimiterea în format default, dacă destinatarul nu e un API care ne impune JSON). Ne-am folosit totuși de acest exemplu pentru a semnala aspectele noi care intervin la trimitere de date JSON, indiferent că ele provind din formular sau din altă sursă: obligativitatea setării Content-Type, sosirea datelor în PHP prin php://input, serializarea la trimitere, deserializarea la

[7] Există totuși în JQuery serializeArray() care produce un array din câmpurile formularului, iar acela ar mai putea fi transformat puțin pentru a ajunge în format JSON, dar versiunea cu FormData e și mai simplă, și mai performantă.

destinație. Ne-am folosit de formular pentru că era o sursă de date la îndemână păstrând exemplele minimale.

Instrumente de testare a cererilor HTTP

Atunci când programăm cereri HTTP și avem bug-uri sau ceva nu funcționează, nu e foarte evident dacă problema e în back-end, în front-end, înainte de trimiterea cererii, după primirea răspunsului etc. De asemenea componentele back-end nu pot fi accesate individual: încercați să deschideți fișierul php din ultimul exemplu direct în browser. Vom vedea o serie de erori care nu înseamnă că e ceva greșit în componenta back-end, ci doar că aceasta nu a primit inputul așteptat de la front-end.

Notice: Trying to get property of non-object in **C:\xamp AJAX\Varianta I - Cerere POST trimisa prin Jquery JSON)\5prelucrare.php** on line **9**

Notice: Trying to get property of non-object in **C:\xamp AJAX\Varianta I - Cerere POST trimisa prin Jquery JSON)\5prelucrare.php** on line **10**

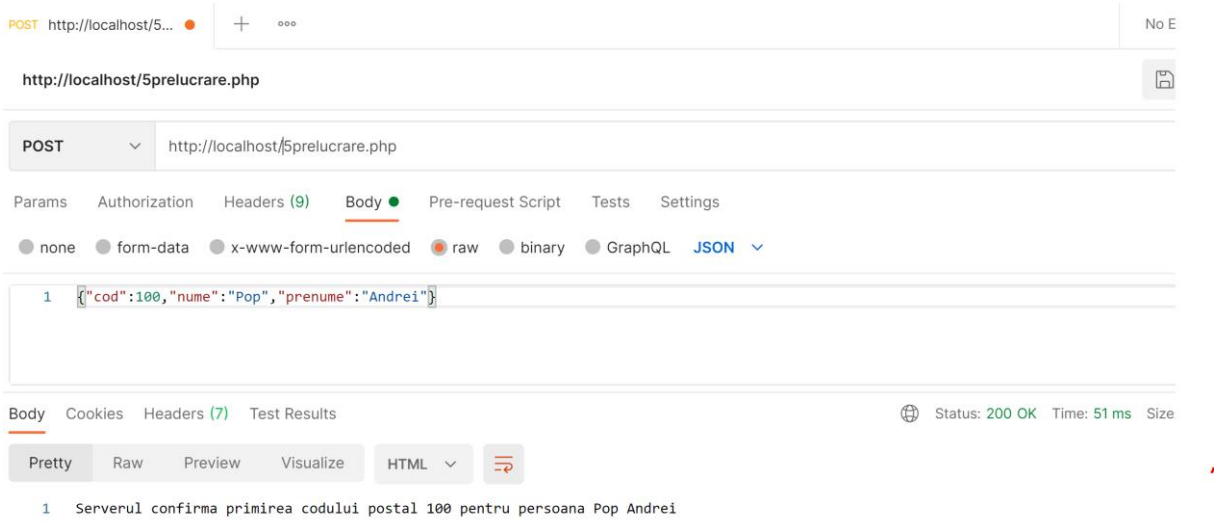
Notice: Trying to get property of non-object in **C:\xamp AJAX\Varianta I - Cerere POST trimisa prin Jquery JSON)\5prelucrare.php** on line **11**

Reamintim, aici fișierele PHP nu mai sunt pagini gata de deschis în browser, ci **funcții care așteaptă un input și returnează un răspuns**. În consecință erorile din back-end adesea nu au un mesaj de eroare vizibil în browser, avem nevoie de instrumente specializate de testare. În general ne folosim de două instrumente în scopul testării:

- **Developer Tools** din browser pentru a monitoriza cereri implementate deja (trimise din browser, nu se poate folosi pentru cereri trimise din back-end cum vom avea în tutorialele viitoare)
- **Postman** (https://www.postman.com/downloads/) pentru a exersa cereri HTTP pe un back-end existent, fără să le programăm (se folosește pentru a putea testa componente back-end înainte să se creeze componentele front-end, ori pentru a ne familiariza cu un API înainte să programăm cereri spre acel API)

Pentru exemplul I, construirea unei cereri în Postman presupune:

- introducerea adresei componentei back-end
- selecția tipului de cerere din lista derulantă de la stânga adresei
- introducerea datelor atașate cererii:
    - dacă se trimit prin GET se folosește rubrica **Params**, în rest se folosește rubrica **Body**
    - în **Body**, tastăm datele JSON iar la dreapta setăm formatul de serializare **JSON** (selecția va seta automat antetul Content-Type; pentru alte antete există și rubrica **Headers**)
    - dacă dorim trimitere de date prin POST în formatele default pentru formulare, tot aici în Body avem putem alege **FormData** și vom putea introduce câmpuri de formular
- apăsarea butonului Send trimite cererea și afișează dedesubt răspunsul, împreună cu antetele primite de la back-end, codul de răspuns (200 OK) și chiar date statistice (timp, număr de octeți) cu care putem construi un raport de evaluare a performanței



Pentru același exemplu, panoul Developer Tools ne oferă în rubrica Network posibilitatea de a monitoriza cererile

- fiecare execuție a unei cereri (la mouseover jos, în exemplul din imagine) face să apară cererea în lista din stânga;
- selectarea ei ne deschide panoul din mijloc unde în rubrica Headers vedem tot ce s-a trimis (jos de tot sunt datele trimise) iar în rubrica Response vedem ce s-a primit; în funcție de asta, putem sesiza dacă s-a produs vreo eroare înainte de trimitere, între trimitere și răspuns, după răspuns, între răspuns și afișarea pe ecran – asta ne permite o depanare mai eficientă, când eroarea nu are o manifestare clar vizibilă în front-end