Sisteme Distribuite - Laborator 4
Servicii RESTful

Servicii RESTful - descriere generală

Serviciile web RESTful sunt servicii slab cuplate, potrivite pentru crearea de API-uri destinate clienților din Internet. Acronimul **REST** provine de la „**RE**presentational **S**tate **T**ransfer”, un stil arhitectural de creare a aplicațiilor client-server orientate pe transferul reprezentărilor de resurse prin cereri şi răspunsuri.

Aceste servicii se bazează pe comunicațiile prin protocolul fără stare **HTTP** (***Hypertext Transfer Protocol*** - **`RFC 7231`**). Deoarece stilul arhitectural REST este **orientat pe resursă**, atunci comunicațiile reprezintă practic operațiile ce se pot efectua asupra resurselor puse la dispoziție.

O resursă este reprezentată de **date** sau **funcționalitate**, depinzând de context. Un exemplu de resursă ar fi starea meteo pentru un anume oraş. Resursele sunt accesate folosind **URI**-uri (***Uniform Resource Identifiers***), iar operațiile care se pot face pe o anumită resursă sunt bine definite de API-ul REST pus la dispoziție în funcție de aplicație.

Structura unui URI

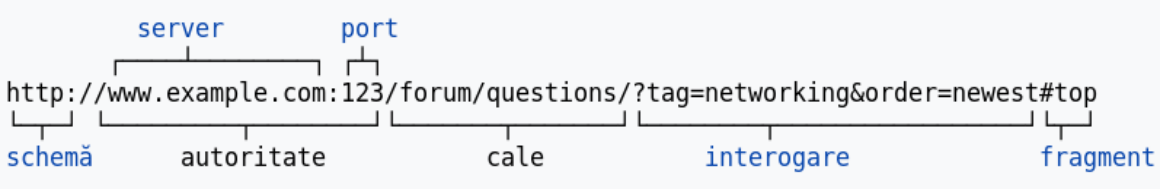


Figura 1 - Structura unui URI

Structura unei cereri HTTP

O cerere HTTP are forma:

```
<VERB_HTTP> <CALE_RESURSĂ> <VERSIUNE_HTTP>
<ANTET_1>: <VALOARE_ANTET_1>
<ANTET_2>: <VALOARE_ANTET_2>
...
<ANTET_N>: <VALOARE_ANTET_N>
<RÂND_GOL>
<CORP_CERERE>
```

Exemplu de cerere GET care preia resursa web identificată prin calea **`/html/rfc5789`** de pe server-ul **`tools.ietf.org`**:

```
GET /html/rfc5789 HTTP/1.1
Host: tools.ietf.org
```

Exemplu de cerere POST trimisă ca rezultat al completării unui formular web cu 2 câmpuri (**`nume`** şi **`prenume`**) către server-ul **`www.ac.tuiasi.ro`**:

```
POST /procesare-student HTTP/1.1
Host: www.ac.tuiasi.ro
Content-Type: application/x-www-form-urlencoded
Content-Length: 27
```

```
nume=Luke&prenume=Skywalker
```

Structura unui răspuns HTTP

```
<VERSIUNE_HTTP> <COD_RĂSPUNS> <EXPLICAȚIE_COD_RĂSPUNS>
<ANTET_1>: <VALOARE_ANTET_1>
<ANTET_2>: <VALOARE_ANTET_2>
...
<ANTET_N>: <VALOARE_ANTET_N>
<RÂND_GOL>
<CORP_CERERE>
```

Exemplu de răspuns HTTP:

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 138
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Connection: close

<html>
  <head>
    <title>The resource you requested</title>
  </head>
  <body>
    <p>The content of the web resource</p>
  </body>
</html>
```

Coduri de răspuns HTTP

- informative: **1XX**
- de succes: **2XX**
- de redirecționare: **3XX**
- eroare la client: **4XX**
- eroare la server: **5XX**

Pentru detalii despre fiecare cod de eroare în parte, consultați secțiunea 6.1 din RFC 7231: https://tools.ietf.org/html/rfc7231#section-6.1

Maparea operațiilor REST peste metodele HTTP

Un API REST bine construit este alcătuit din operații bine definite asupra resurselor țintă, operații care se mapează corespunzător peste metodele HTTP disponibile conform protocolului. Se recomandă respectarea semnificației metodelor HTTP şi a codurilor de răspuns transmise de server către client, în funcție de caz.

Operațiile de tip **CRUD** care se pot face asupra unei resurse sunt următoarele:

- (**C**reate) crearea unei resurse - **POST** sau **PUT**
  - POST se foloseşte pentru crearea unei resurse al cărui identificator unic (ID) nu este cunoscut de client şi este generat de server
  - PUT se foloseşte atunci când clientul decide cum este identificată resursa respectivă

- (**R**etrieve) preluarea unei resurse - **GET**
- (**U**pdate) actualizarea unei resurse - **PUT** sau **PATCH** (**RFC 5789**)
  - PUT înlocuieşte complet resursa cu reprezentarea acesteia din corpul cererii
  - cererile de tip PATCH conţin doar diferenţele dintre reprezentările resurselor referenţiate, şi deci server-ul va modifica resursa existentă aplicând ***patch***-ul trimis de client
- (**D**elete) ştergerea unei resurse - **DELETE**

Aplicație demonstrativă - agendă telefonică

Pentru a ilustra conceptele REST prezentate anterior, veți crea o aplicație simplă care va gestiona o agendă telefonică ce conține următoarele date:

- identificator unic
- nume
- prenume
- număr de telefon

Schema RESTful

Se începe cu proiectarea schemei RESTful, care expune resursele puse la dispoziție, operațiile care pot fi efectuate asupra lor, precum şi tipurile de răspuns în funcție de cererile făcute de client pe resursele respective.

CALE	VERB HTTP	COD RĂSPUNS	SEMNIFICAȚIE
`/person/{id}`	**GET**	**200 OK**	Clientul a cerut datele unei persoane din agendă şi a primit rezultatul
		404 NOT FOUND	Clientul a cerut datele unei persoane care nu există în agendă
	PUT	**202 ACCEPTED**	Clientul a cerut actualizarea persoanei cu ID-ul **id**, iar server-ul a actualizat intrarea în agendă
		404 NOT FOUND	Clientul a cerut actualizarea unei persoane care nu există în agendă
	DELETE	**200 OK**	Clientul a cerut ştergerea persoanei cu ID-ul **id** din agendă, iar ştergerea s-a efectuat cu succes
		404 NOT FOUND	Clientul a cerut ştergerea unei persoane care nu există în agendă
`/person`	**POST**	**201 CREATED**	Clientul a cerut crearea unei intrări în agendă pe server, ca şi subordonat al resursei **person**. Adăugarea persoanei în agendă a fost făcută cu succes.
		400 BAD REQUEST	Clientul a cerut crearea unei intrări în agendă pe server, dar datele pe care le-a trimis în corpul cererii nu sunt corecte

/agenda **cu parametrii URL:** • **firstName** • **lastName** • **telephone**	**GET**	**200 OK**	Clientul a cerut o listă de persoane din agendă (conform cu eventualele filtre din URL) şi a primit o listă nevidă, validă
		204 NO CONTENT	Clientul a cerut o listă de persoane din agendă, dar nu există niciun element în listă care corespunde filtrelor cerute

Diagrama serviciului AgendaService

Implementarea claselor

Pentru implementare, creați o aplicație **Spring Boot**, conform modelului explicat în laboratorul 3. Utilitarul de gestiune a proiectului este la alegere (Maven / Gradle).

Creați pachetele corespunzătoare diagramei de clase prezentată anterior. Structura proiectului este următoarea:

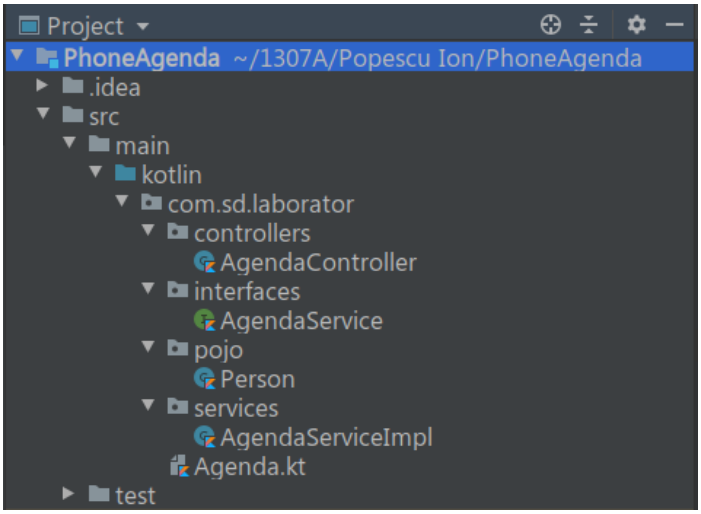


Figura 2 - Structura proiectului

• clasa **Person**

```
package com.sd.laborator.pojo

data class Person(
    var id: Int = 0,
    var lastName: String = "",
    var firstName: String = "",
```

```
    var telephoneNumber: String = ""
)
```

Aceasta încapsulează datele care „circulă” prin componentele aplicației, şi este serializată în momentul în care datele trebuie trimise clientului, respectiv deserializată când clientul trimite date în cererile către server.

De aceea, pentru implementare s-a folosit o clasă de tip **`data class`** din Kotlin.

• interfața **`AgendaService`**

```
package com.sd.laborator.interfaces

import com.sd.laborator.pojo.Person

interface AgendaService {
    fun getPerson(id: Int) : Person?
    fun createPerson(person: Person)
    fun deletePerson(id: Int)
    fun updatePerson(id: Int, person: Person)
    fun searchAgenda(lastNameFilter: String, firstNameFilter: String,
telephoneNumberFilter: String): List<Person>
}
```

Aceasta este interfața ce expune sub formă de serviciu un set de operațiuni care sunt folosite în continuare de *controller*. Implementările efective pentru metodele expuse se află în *bean*-ul **`AgendaServiceImpl`**, explicat în cele ce urmează.

• serviciul **`AgendaServiceImpl`**

```
package com.sd.laborator.services;

import com.sd.laborator.interfaces.AgendaService
import com.sd.laborator.pojo.Person
import org.springframework.stereotype.Service
import java.util.concurrent.ConcurrentHashMap

@Service
class AgendaServiceImpl : AgendaService {
    companion object {
        val initialAgenda = arrayOf(
            Person(1, "Hello", "Kotlin", "1234"),
            Person(2, "Hello", "Spring", "5678"),
            Person(3, "Hello", "Microservice", "9101112")
        )
    }

    private val agenda = ConcurrentHashMap<Int, Person>(
        initialAgenda.associateBy { person: Person -> person.id }
    )

    override fun getPerson(id: Int): Person? {
        return agenda[id]
    }

    override fun createPerson(person: Person) {
        agenda[person.id] = person
```

```
    }

    override fun deletePerson(id: Int) {
        agenda.remove(id)
    }

    override fun updatePerson(id: Int, person: Person) {
        deletePerson(id)
        createPerson(person)
    }

    override fun searchAgenda(lastNameFilter: String, firstNameFilter:
String, telephoneNumberFilter: String): List<Person> {
        return agenda.filter {
            it.value.lastName.toLowerCase().contains(lastNameFilter,
ignoreCase = true) &&

it.value.firstName.toLowerCase().contains(firstNameFilter, ignoreCase
= true) &&

it.value.telephoneNumber.contains(telephoneNumberFilter)
        }.map {
            it.value
        }.toList()
    }
}
```

Serviciul **`AgendaServiceImpl`** este expus sub formă de *bean* în contextul de execuție Spring (de aici adnotarea **`@Service`**).

Pentru simplitate, nu s-a folosit o bază de date sau o altă modalitate de persistență a datelor, ci agenda telefonică este păstrată în memorie şi este validă atât timp cât aplicația Spring se află în execuție. Agenda este păstrată într-o structură de date de tip **`ConcurrentHashMap`** pentru a asigura proprietatea ***thread-safe***.

Inițial, agenda telefonică este populată cu câteva date de test menținute într-un obiect companion Kotlin (***companion object***). De ce? Pentru că acele date inițiale sunt aceleaşi pentru orice instanță s-ar crea în memorie, şi deci vectorul **`initialAgenda`** trebuie să aparțină clasei, şi nu instanței clasei respective.

Implementările metodelor nu sunt complicate, ci de la sine înțelese, cu excepția (poate) a metodei **`searchAgenda`**: aceasta filtrează elementele din colecția **`agenda`**, pe baza unui predicat logic trimis funcției **`filter`**. Rezultatul funcției **`filter`** este tot o colecție de acelaşi tip, deci un **`ConcurrentHashMap`**, dar care conține doar elementele ce satisfac predicatul respectiv. Colecția rezultată este transformată folosind funcția map, ce preia fiecare pereche **`<Int, Person>`** şi o transformă într-un element de tip **`Person`** (practic, se ignoră identificatorul). De ce se face acest lucru? Pentru că se doreşte a se trimite ca răspuns o listă cât mai simplă, formată doar din obiecte de tip **`Person`**. Lista este construită cu metoda **`toList()`** aplicată pe **`HashMap`**-ul rezultat.

- *controller*-ul **`AgendaController`**

```
package com.sd.laborator.controllers

import com.sd.laborator.interfaces.AgendaService
import com.sd.laborator.pojo.Person
```

```
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.http.HttpStatus
import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.*

@RestController
class AgendaController {
    @Autowired
    private lateinit var agendaService: AgendaService

    @RequestMapping(value = ["/person"], method =
[RequestMethod.POST])
    fun createPerson(@RequestBody person: Person):
ResponseEntity<Unit> {
        agendaService.createPerson(person)
        return ResponseEntity(Unit, HttpStatus.CREATED)
    }

    @RequestMapping(value = ["/person/{id}"], method =
[RequestMethod.GET])
    fun getPerson(@PathVariable id: Int): ResponseEntity<Person?> {
        val person: Person? = agendaService.getPerson(id)
        val status = if (person == null) {
            HttpStatus.NOT_FOUND
        } else {
            HttpStatus.OK
        }
        return ResponseEntity(person, status)
    }

    @RequestMapping(value = ["/person/{id}"], method =
[RequestMethod.PUT])
    fun updatePerson(@PathVariable id: Int, @RequestBody person:
Person): ResponseEntity<Unit> {
        agendaService.getPerson(id)?.let {
            agendaService.updatePerson(it.id, person)
            return ResponseEntity(Unit, HttpStatus.ACCEPTED)
        } ?: return ResponseEntity(Unit, HttpStatus.NOT_FOUND)
    }

    @RequestMapping(value = ["/person/{id}"], method =
[RequestMethod.DELETE])
    fun deletePerson(@PathVariable id: Int): ResponseEntity<Unit> {
        if (agendaService.getPerson(id) != null) {
            agendaService.deletePerson(id)
            return ResponseEntity(Unit, HttpStatus.OK)
        } else {
            return ResponseEntity(Unit, HttpStatus.NOT_FOUND)
        }
    }

    @RequestMapping(value = ["/agenda"], method = [RequestMethod.GET])
    fun search(@RequestParam(required = false, name = "lastName",
defaultValue = "") lastName: String,
                   @RequestParam(required = false, name =
"firstName", defaultValue = "") firstName: String,
```

```
                    @RequestParam(required = false, name =
"telephone", defaultValue = "") telephoneNumber: String):
            ResponseEntity<List<Person>> {
        val personList = agendaService.searchAgenda(lastName,
firstName, telephoneNumber)
        var httpStatus = HttpStatus.OK
        if (personList.isEmpty()) {
            httpStatus = HttpStatus.NO_CONTENT
        }
        return ResponseEntity(personList, httpStatus)
    }
}
```

Clasa *controller* este adnotată cu **`@RestController`** pentru ca Spring să o configureze corespunzător şi să simplifice munca dezvoltatorului: serializările / deserializările se fac automat în şi din obiecte JSON (transparent pentru dezvoltator), iar răspunsurile trimise în metodele mapate pe căile de acces se consideră în mod automat ca fiind corpul răspunsurilor HTTP la cererile clientului (din nou, fără intervenţia dezvoltatorului).

Controller-ul implementează câte o metodă de tratare a fiecărei operaţii posibile din schema REST prezentată anterior.

Serviciul **`AgendaService`** este injectat automat ca dependenţă în faza de scanare de componente a Spring (de aici adnotarea **`@Autowired`** - va fi ales *bean*-ul ce conţine o implementare a interfeţei **`AgendaService`**, adică **`AgendaServiceImpl`**, în acest caz).

Fiecare metodă tratează cazurile de excepţie ce pot apărea din vina clientului. Spre exemplu, considerând operaţia de actualizare a unei persoane în agenda de telefon:

```
    @RequestMapping(value = ["/person/{id}"], method =
[RequestMethod.PUT])
    fun updatePerson(@PathVariable id: Int, @RequestBody person:
Person): ResponseEntity<Unit> {
        agendaService.getPerson(id)?.let {
            agendaService.updatePerson(it.id, person)
            return ResponseEntity(Unit, HttpStatus.ACCEPTED)
        } ?: return ResponseEntity(Unit, HttpStatus.NOT_FOUND)
    }
```

Variabila de cale **`id`** este preluată automat din URL folosind adnotarea **`@PathVariable`**, iar corpul cererii clientului (obiectul în sine care urmează a fi actualizat) este preluat folosind adnotarea **`@RequestBody`**. Aşadar, în interiorul metodei **`updatePerson()`**, dezvoltatorul are acces la corpul cererii clientului, respectiv la parametrii trimişi în URL.

Dacă utilizatorul cere să actualizeze o persoană cu un ID inexistent, metoda trimite ca răspuns un cod de eroare de tip **`404 NOT FOUND`**, semantic echivalent cu situaţia apărută. Pentru a trimite un răspuns personalizat înapoi la client, pe baza logicii implementate de dezvoltator, se încapsulează corpul răspunsului într-un obiect de tip **`ResponseEntity`**. Această clasă şablon (***template***-izată) acceptă ca membri conţinutul răspunsului efectiv şi codul de răspuns HTTP dorit de dezvoltator.

Dacă se doreşte ca metoda să returneze un obiect nul, sau un obiect gol, se foloseşte clasa **`ResponseEntity`** în conjuncţie cu clasa **`Unit`** din Kotlin, ce ţine locul tipului de date **`void`**. În acest caz, pentru metoda **`updatePerson()`**, nu se doreşte a fi trimis niciun corp de răspuns, ci doar codul în sine (**`202 ACCEPTED`**, sau **`404 NOT FOUND`**), şi deci răspunsul va fi de tip **`ResponseEntity<Unit>`** şi clientul primeşte un răspuns fără conţinut.

Considerând operaţia de preluare a unei persoane din agendă:

```
    @RequestMapping(value = ["/person/{id}"], method =
[RequestMethod.GET])
    fun getPerson(@PathVariable id: Int): ResponseEntity<Person?> {
        val person: Person? = agendaService.getPerson(id)
        val status = if (person == null) {
            HttpStatus.NOT_FOUND
        } else {
            HttpStatus.OK
        }
        return ResponseEntity(person, status)
    }
```

Asemănător cu cazul anterior, se dorește ca atunci când poate apărea o problemă din vina clientului, aceasta să fie tratată corespunzător: dacă se cer datele unei persoane cu un ID inexistent, server-ul trebuie să notifice clientul de acest lucru, printr-un răspuns de tip **`404 NOT FOUND`**.

Dacă persoana respectivă există, atunci server-ul o va prelua din colecția internă şi o va servi clientului serializând obiectul ce o încapsulează în corpul răspunsului HTTP (împreună cu un cod **`200 OK`**).

• punctul de intrare al aplicației Spring - **`Agenda.kt`**

```
package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class PhoneAgenda

fun main(args: Array<String>) {
    runApplication<PhoneAgenda>(*args)
}
```

Testarea aplicației cu Postman

Având de a face cu un API de tip REST, testarea aplicației implementate va presupune comunicații HTTP, cereri client de diverse tipuri în funcție de ceea ce se dorește din partea server-ului. De asemenea, corpul cererilor trebuie să conțină date în format JSON, deoarece acesta este modul implicit de lucru cu date în cazul serviciilor REST.

O variantă simplă de a lucra cu cereri HTTP personalizate şi de a vizualiza răspunsurile într-un mod facil este folosirea aplicației **Postman**. Descărcați-l de aici:

https://www.getpostman.com/product/api-client

Pe Linux nu necesită instalare, ci doar dezarhivare şi apoi execuția este simplă, deschizând un terminal în folder-ul **`Postman`** dezarhivat şi folosind comanda:

```
./Postman
```

La pornire, veți fi întrebați dacă doriți să vă creați cont, să vă înregistrați etc. Ignorați toate aceste cereri până ajungeți la interfața principală.

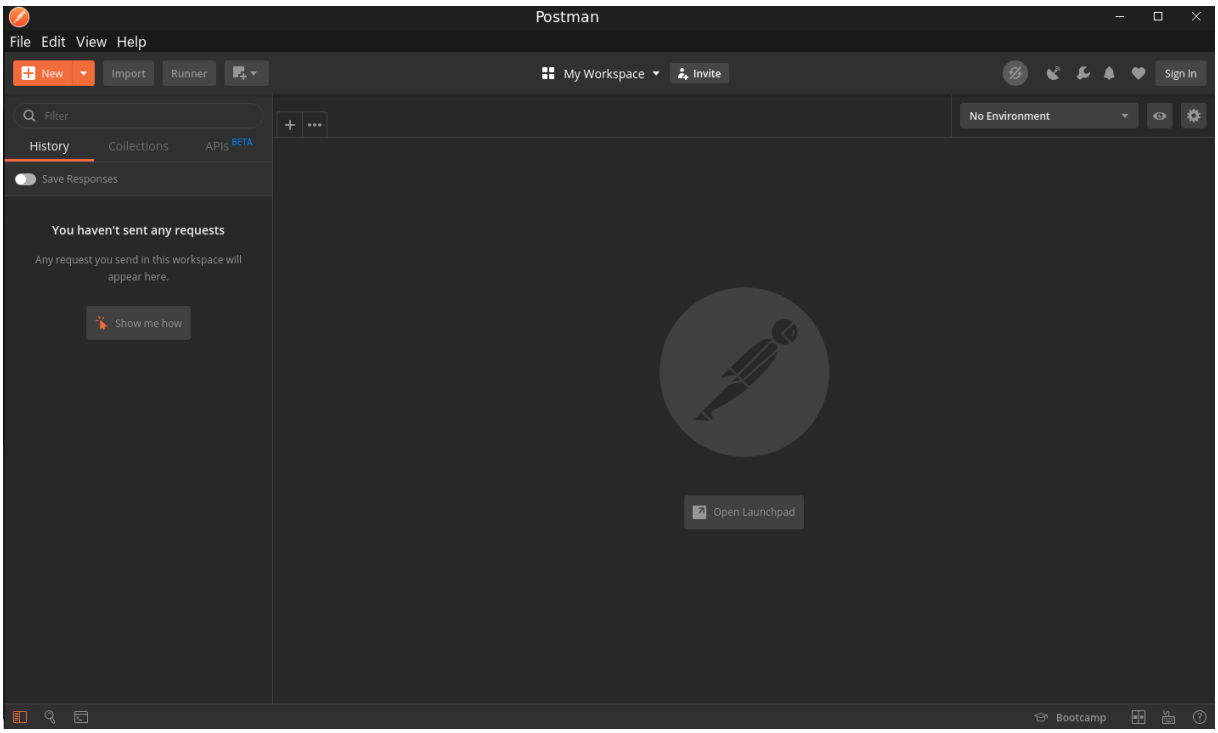


Figura 3 - Interfața Postman

Apăsați pe butonul cu un semn „plus” pentru a crea o cerere HTTP nouă.

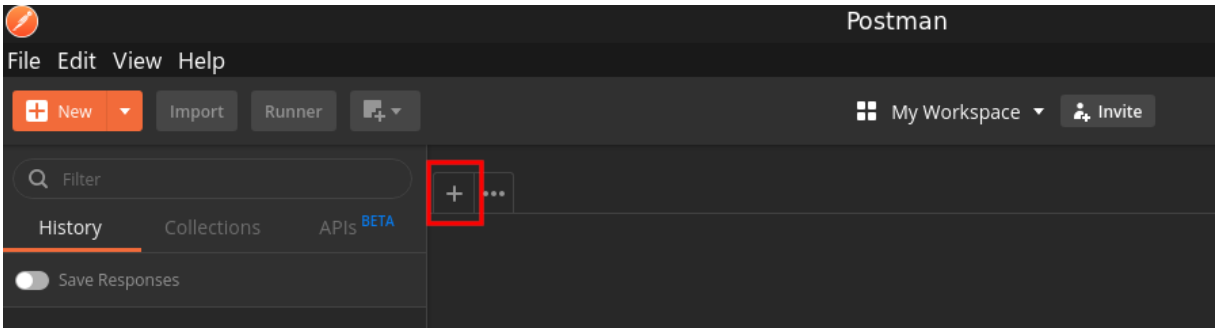


Figura 4 - Introducere cerere nouă în Postman

În secțiunea „**Params**” se pot adăuga parametri de filtrare în URL. Aceştia au rolul, în cazul API-urilor REST, de filtrare a răspunsurilor trimise clientului în funcție de preferințe.

În secțiunea „**Body**” se completează corpul mesajului HTTP trimis către server.

Pentru început, preluați întreaga listă de persoane din agenda telefonică (nu uitați să compilați proiectul şi să porniți aplicația Spring Boot din IntelliJ).

Conform schemei REST a aplicației, pentru a prelua o listă de persoane, trebuie să accesați resursa disponibilă la calea **`/agenda`**, deci, în câmpul „**Enter request URL**”, introduceți URL-ul: http://localhost:8080/agenda. În partea stângă a acestui câmp selectați **`GET`** ca şi tip de cerere. Nu introduceți niciun parametru pentru moment, şi apăsați butonul „**Send**” din partea dreaptă.

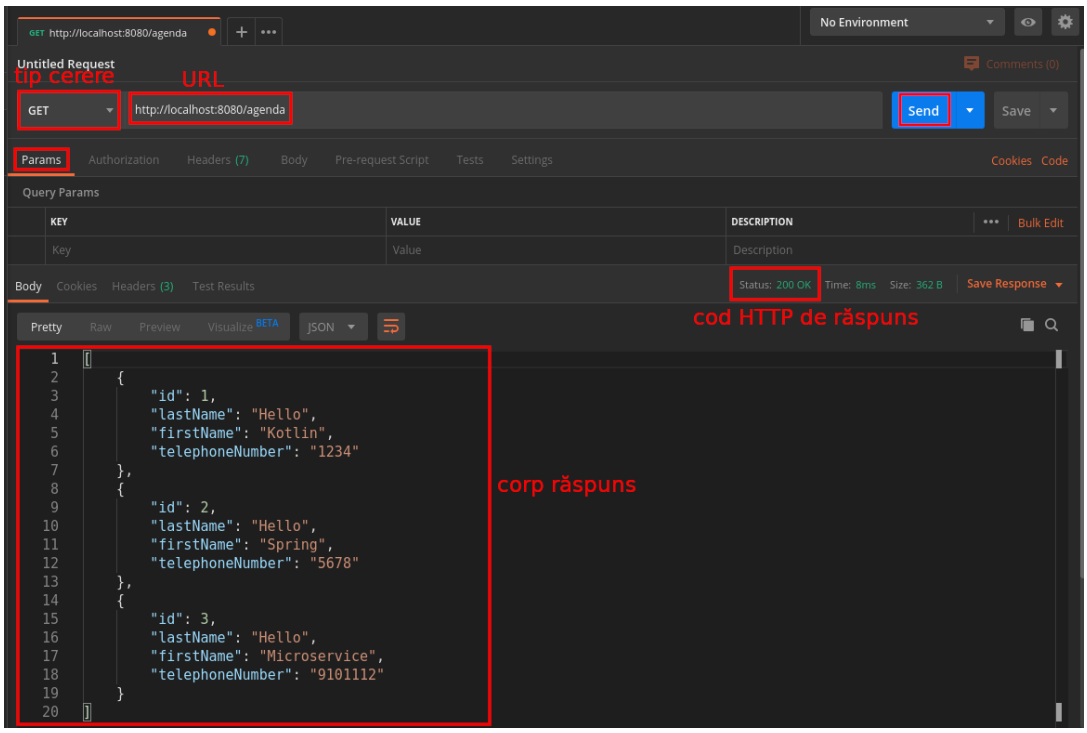


Figura 5 - Exemplu de cerere HTTP de tip GET

Se observă că server-ul a răspuns prin trimiterea întregii agende telefonice sub formă de obiect JSON (deoarece controller-ul REST utilizează acest format pentru obiectele trimise în comunicațiile client-server).

Încercați acum să aplicați un filtru, spre exemplu să căutați un număr de telefon pe baza unui nume. Adăugați un parametru URL care să filtreze rezultatele pe baza prenumelui „**Kotlin**”:

- key: **`firstName`**
- value: **`Kotlin`**

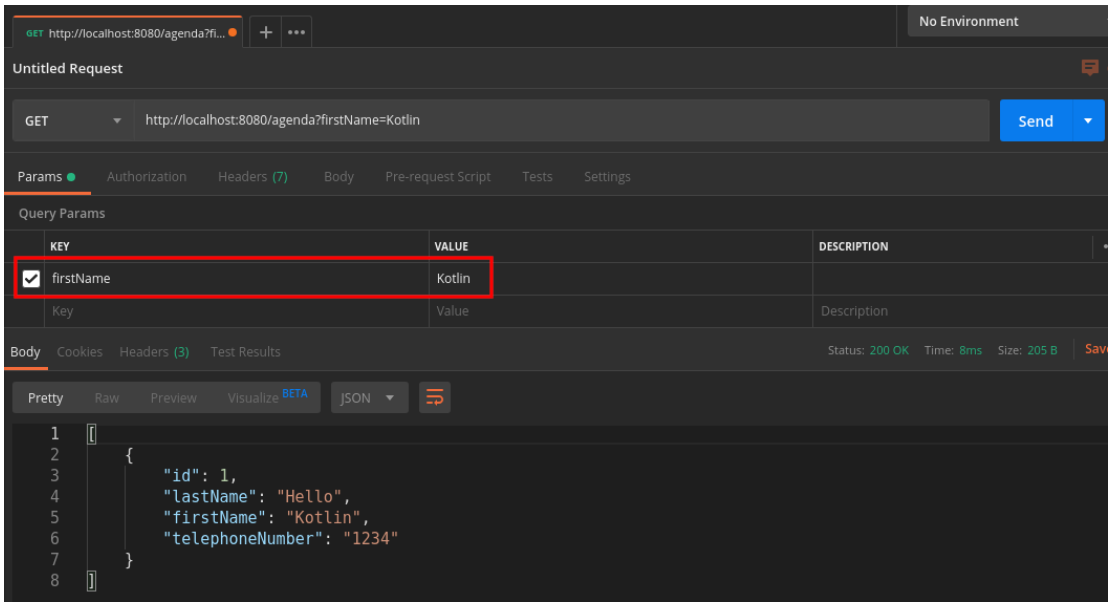


Figura 6 - Exemplu de cerere GET cu filtre

În acest caz, a fost returnat un singur rezultat ce corespundea filtrului.

Acum, încercați să aplicați un filtru care nu generează niciun rezultat, spre exemplu: **`firstName: 123`**.

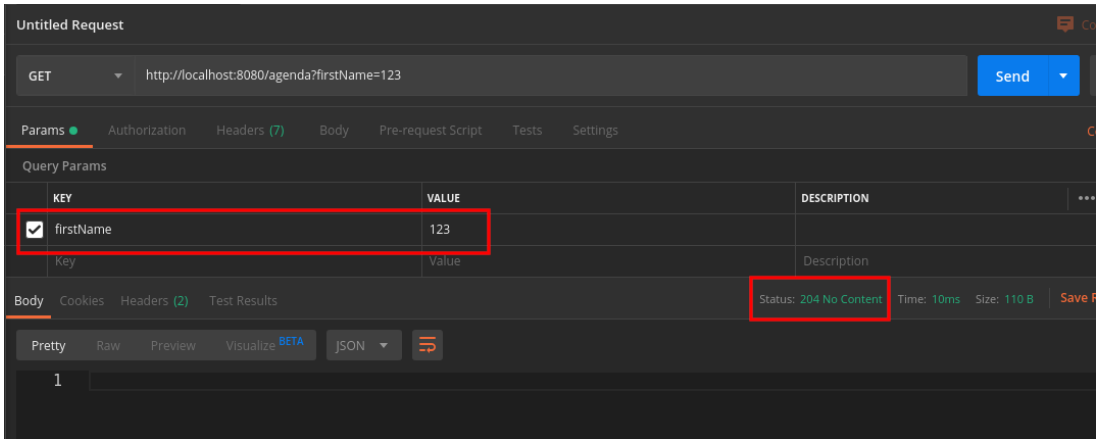
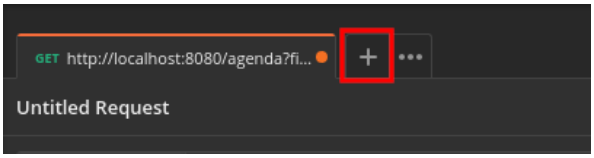


Figura 7 - Răspuns vid la cerere de tip GET

Se observă cum server-ul a răspuns cu **`204 NO CONTENT`**, deoarece nu există niciun element în agenda telefonică ce să corespundă cu filtrul ales.

Acum, încercați să adăugați o nouă intrare în agendă. Conform schemei REST a aplicației, trebuie să se trimită o cerere HTTP de tip **`POST`** către calea **`/person`**, iar în corpul cererii se încapsulează obiectul JSON cu datele despre persoană. Apăsați pe butonul „**plus**” pentru a deschide alt tab cu o cerere nouă.



Selectați tipul de cerere „POST” și completați URL-ul: http://localhost:8080/person. Apoi, selectați tab-ul „**Body**”, bifați „**raw**” și selectați ca tip de conținut „**JSON**”. Completați corpul cererii astfel:

```
{
   "id": 10,
   "firstName": "Luke",
   "lastName": "Skywalker",
   "telephoneNumber": "123456789"
}
```

Apăsați pe butonul „**Send**” și observați cum server-ul răspunde cu **`201 CREATED`**, deci resursa dorită a fost creată cu succes (vedeți figura de mai jos).

Dacă veți schimba acum la tab-ul anterior cu cererea GET și veți șterge filtrele adăugate (**sau dezactiva, folosind bifa din partea stângă**), după trimiterea cererii respective, obțineți lista actualizată cu agenda telefonică, ce conține și noul obiect ce tocmai a fost adăugat.

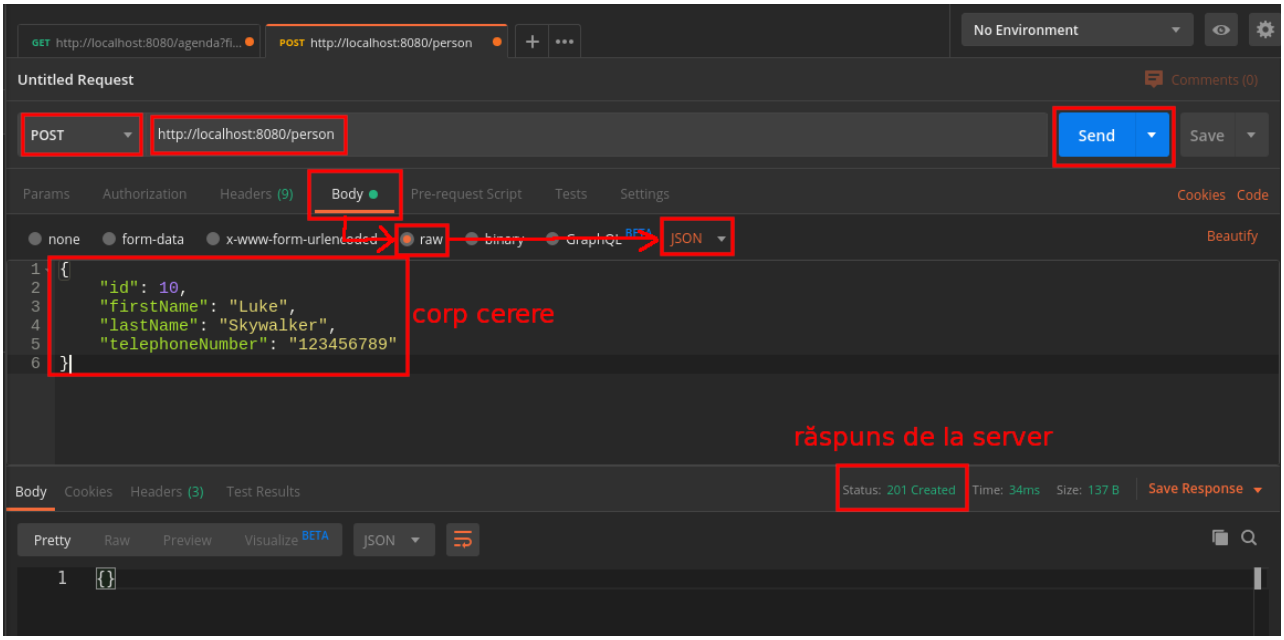


Figura 8 - Exemplu de cerere POST

Pentru a exemplifica şi tratarea cazurilor de excepție, încercați să ştergeți o resursă de pe server. Din nou, conform schemei REST, pentru a şterge o intrare din agendă, trebuie trimisă o cerere HTTP de tip **`DELETE`** către calea **`/person/{id}`**, cu variabila de cale **`id`** având una din valorile existente în colecția de obiecte de tip **`Persoana`**.

De exemplu, ştergeți persoana cu ID-ul **`1`**: apăsați din nou pe butonul „**plus**”, selectați **`DELETE`** ca şi tip de cerere, completați URL-ul cu http://localhost:8080/person/1, lăsați celelalte câmpuri necompletate şi apăsați „**Send**”.

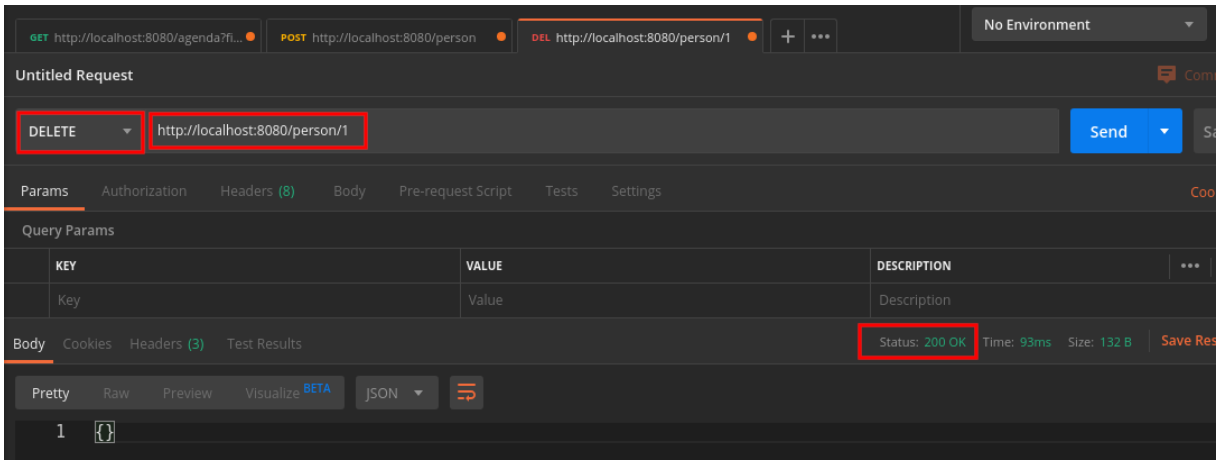


Figura 9 - Exemplu de cerere de tip DELETE

Server-ul a şters resursa cu succes, conform răspunsului primit, de tip 200 OK. Confirmați acest lucru preluând din nou întreaga listă de persoane din agendă, cu cererea HTTP configurată în primul tab:

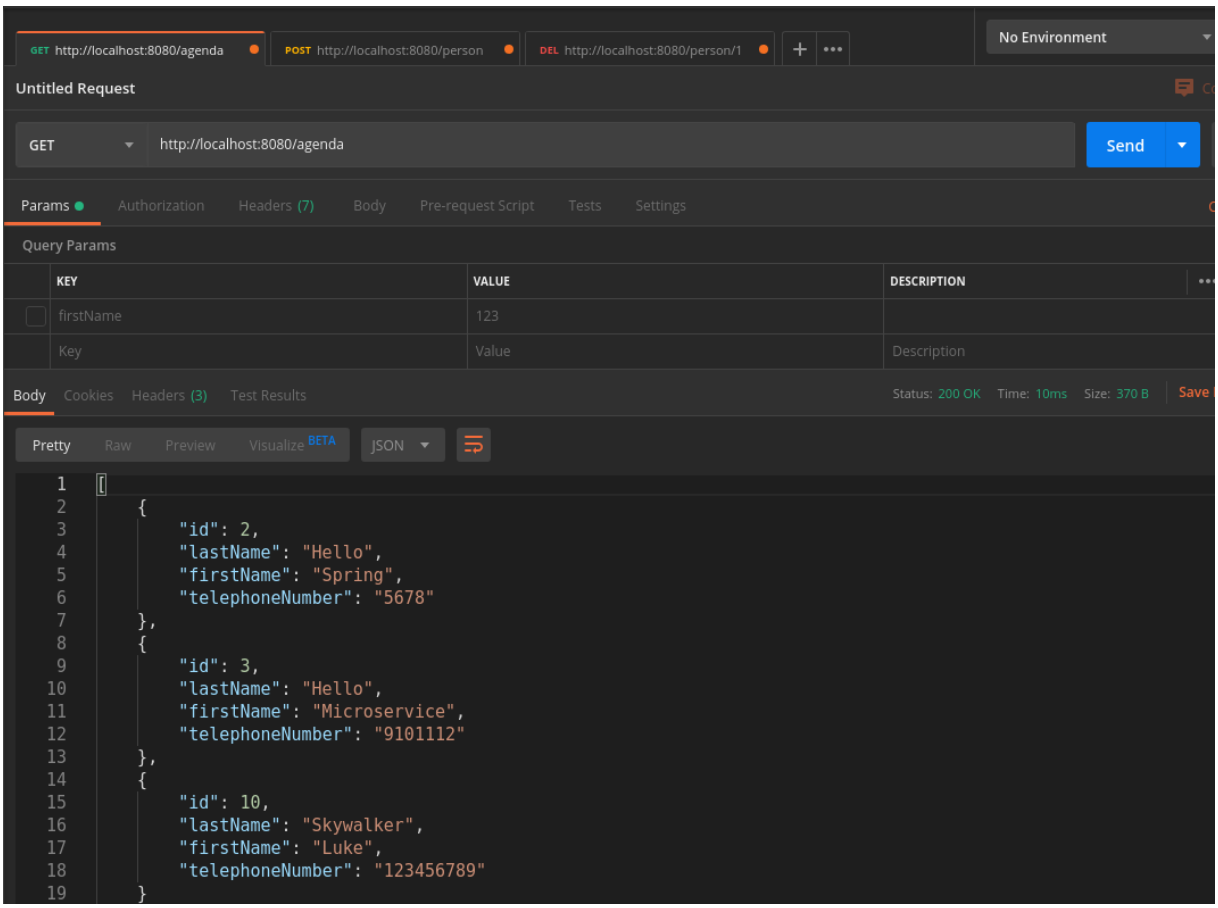


Figura 10 - Lista de persoane după ştergere

Dacă încercați să ştergeți din nou persoana cu ID-ul 1, veți primi o eroare de la server, deoarece aceasta nu mai există (conform cu acele cazuri de excepție tratate în codul de business):

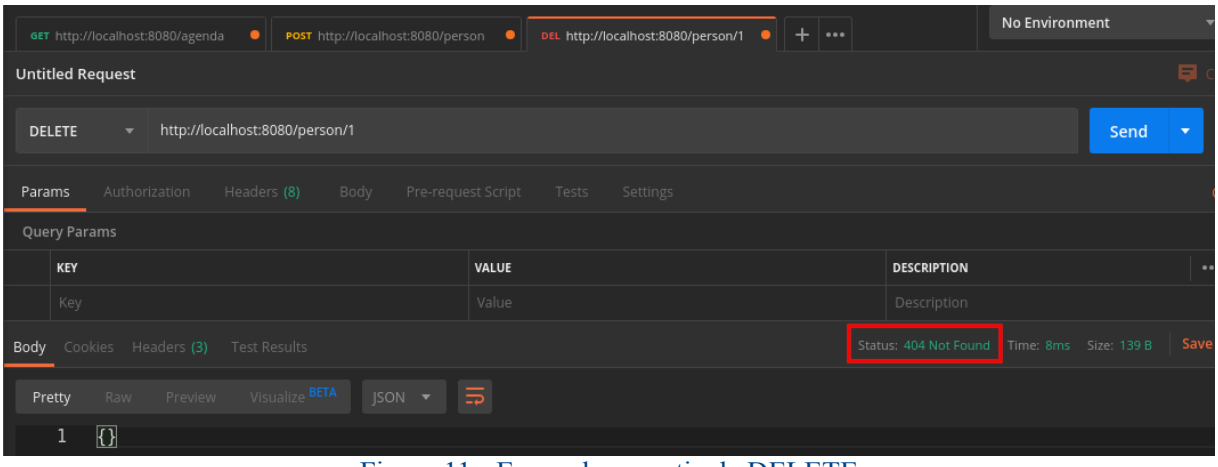


Figura 11 - Eroare la operația de DELETE

Celelalte teste pentru operațiile şi cazurile de excepție rămase sunt lăsate ca exercițiu în timpul laboratorului.

Aplicații şi teme

Teme de laborator

1. Extrageți diagrama de clase din codul exemplu al aplicației **Agendă**, disponibil în

laborator.

2. Adăugați o interfață grafică simplă bazată pe un formular HTML, cu care să puteți implementa operațiile **GET** și **POST**. Modificați apoi HTML-ul astfel încât să se poată trimite către program interogări HTTP de tip **PUT** și **DELETE** (cu iubitul vostru JavaScript - folosiți **fetch** sau **AJAX**).

Temă pentru acasă

Proiectați și implementați o aplicație cu servicii RESTful pentru **gestiunea cheltuielilor curente pentru membrii a unei familii** (nivel de complexitate asemănător cu cel din laborator). Pentru aplicația respectivă, creați și o interfață folosind ***framework***-ul **Flask** din Python.

Documentație utilă:

- documentația Flask: https://flask.palletsprojects.com/en/1.1.x/
- cum se instalează Flask: https://flask.palletsprojects.com/en/1.1.x/installation/#install-flask
- **Building REST APIs with Flask** - Kunal Relan
- **Flask Web Development** - Miguel Grinberg

Bibliografie

[1]: Creating a RESTful Web Service with Spring Boot - https://kotlinlang.org/docs/tutorials/spring-boot-restful.html
[2]: What Are RESTful Web Services? - https://javaee.github.io/tutorial/jaxrs001.html
[3]: Hypertext Transfer Protocol - https://tools.ietf.org/html/rfc7231
[4]: PATCH Method for HTTP - https://tools.ietf.org/html/rfc5789
[5]: HTTP request methods - https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
[6]: Obiecte companion în Kotlin - https://kotlinlang.org/docs/reference/object-declarations.html#companion-objects