Sisteme Distribuite - Laborator 4
Servicii RESTful

Servicii RESTful - descriere generală

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

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

O resursă este reprezentată de **date** sau **funcţionalitate**, 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 operaţiile care se pot face pe o anumită resursă sunt bine definite de API-ul REST pus la dispoziţie în funcţie de aplicaţie.

Structura unui URI

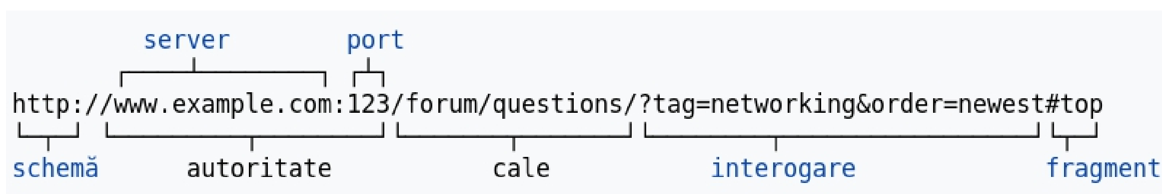


Figura 1 - Structura unui URI

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> <EXPLICAŢIE_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 redirecţionare: **`3XX`**
- eroare la client: **`4XX`**
- eroare la server: **`5XX`**

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

Maparea operaţiilor REST peste metodele HTTP

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

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

- (Create) 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**

Aplicaţie demonstrativă - agendă telefonică

Pentru a ilustra conceptele REST prezentate anterior, veţi crea o aplicaţie simplă care va gestiona o agendă telefonică ce conţine următoarele date:

- identificator unic
- nume
- prenume
- număr de telefon

Diagrama serviciului AgendaService

Schema RESTful

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

CALE	VERB HTTP	COD RĂSPUNS	SEMNIFICAŢIE
`/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
	`PATCH`	`204 NO CONTENT`	Clientul a cerut actualizarea unor informații a persoanei cu ID-ul **`id`**, iar server-ul a actualizat intrarea în agendă
		`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

Implementarea claselor

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

Structura proiectului este următoarea:

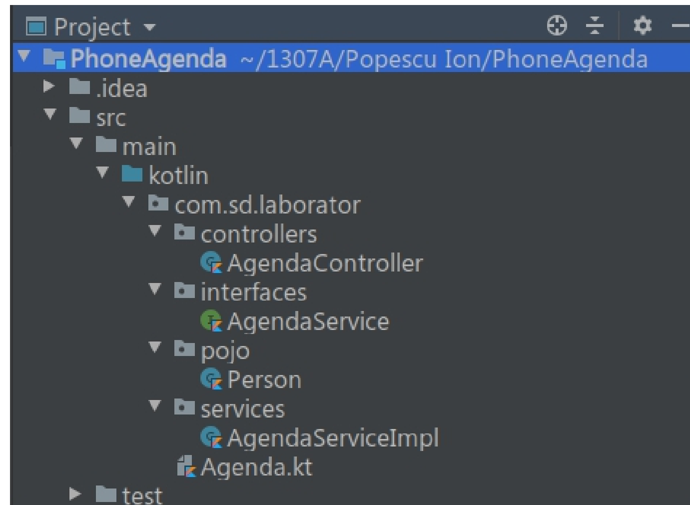


Figura 3 - Structura proiectului

Adăugați următoarea dependență (în tagul <dependencies>) în fișierul Maven pentru a include utilitarul “Json Patch” utilizat în operația de Patch.

```
<dependency>
     <groupId>com.github.java-json-tools</groupId>
     <artifactId>json-patch</artifactId>
     <version>1.13</version>
</dependency>
```

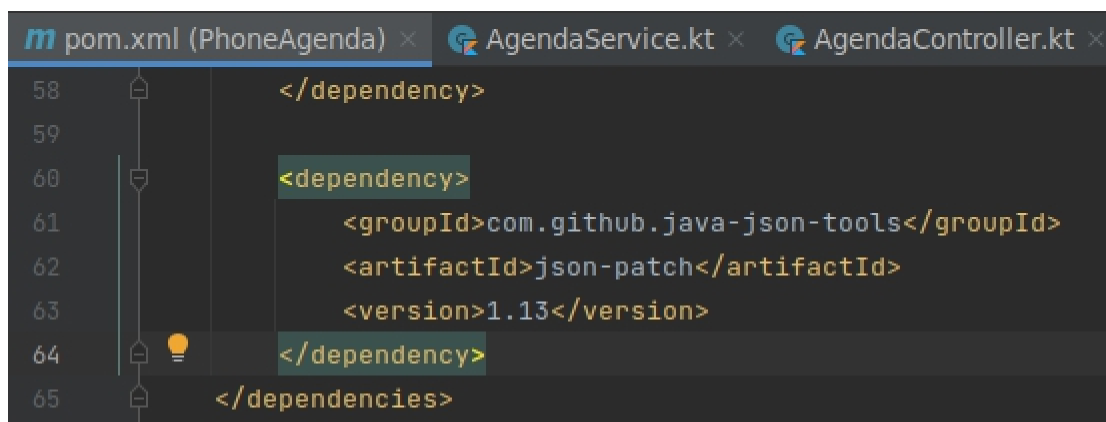


Figura 4 - Adăugarea dependenței “json-patch”

- 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 aplicaţiei, 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.

- interfaţa **`IAgendaService`**

```
package com.sd.laborator.interfaces

import com.sd.laborator.pojo.Person

interface IAgendaService {
    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 interfaţa ce expune sub formă de serviciu un set de operaţiuni care sunt folosite în continuare de *controller*. Implementările efective pentru metodele expuse se află în *bean*-ul **`AgendaService`**, explicat în cele ce urmează.

- serviciul **`AgendaService`**

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

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

@Service
class AgendaService : IAgendaService {
    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 **AgendaService** este expus sub formă de *bean* în contextul de execuţie Spring (de aici adnotarea **@Service**).

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

Iniţial, agenda telefonică este populată cu câteva date de test menţinute într-un obiect companion Kotlin (***companion object***). De ce? Pentru că acele date iniţiale sunt aceleaşi pentru orice instanţă s-ar crea în memorie, şi deci vectorul **initialAgenda** trebuie să aparţină clasei, şi nu instanţei clasei respective.

Implementările metodelor nu sunt complicate, ci de la sine înţelese, cu excepţia (poate) a metodei **searchAgenda**: aceasta filtrează elementele din colecţia **agenda**, pe baza unui predicat logic trimis funcţiei **filter**. Rezultatul funcţiei **filter** este tot o colecţie de acelaşi tip, deci un **ConcurrentHashMap**, dar care conţine doar elementele ce satisfac predicatul respectiv. Colecţia rezultată este transformată folosind funcţia 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.fasterxml.jackson.databind.ObjectMapper
```

```
import com.github.fge.jsonpatch.JsonPatch
import com.sd.laborator.interfaces.IAgendaService
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: IAgendaService

    @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.PATCH])
    fun patchPerson(@PathVariable id: Int, @RequestBody
patchOperations: JsonPatch): ResponseEntity<Unit> {
        _agendaService.getPerson(id)?.let {

            // aplicam operatiile de Patch peste obiectul gasit
            val objectMapper = ObjectMapper()
            val patchedPersonJsonNode =
patchOperations.apply(objectMapper.valueToTree(it))
            val patchedPerson =
objectMapper.treeToValue(patchedPersonJsonNode, Person::class.java)

            // updatam obiectul obtinut dupa operatia de patch
```

```
            _agendaService.updatePerson(it.id, patchedPerson)
            return ResponseEntity(Unit, HttpStatus.NO_CONTENT)
        } ?: 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.

Operația de actualizare (PUT)

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.

Operația de preluare (GET)

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 doreşte 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 colecţia internă şi o va servi clientului serializând obiectul ce o încapsulează în corpul răspunsului HTTP (împreună cu un cod **`200 OK`**).

Operația de actualizare (PATCH)

Considerând operaţia de actualizare a informațiilor unei persoane din agendă:

```
@RequestMapping(value = ["/person/{id}"], method =
[RequestMethod.PATCH])
    fun patchPerson(@PathVariable id: Int, @RequestBody
patchOperations: JsonPatch): ResponseEntity<Unit> {
        _agendaService.getPerson(id)?.let {
```

```
            // aplicam operatiile de Patch peste obiectul gasit
            val objectMapper = ObjectMapper()
            val patchedPersonJsonNode =
patchOperations.apply(objectMapper.valueToTree(it))
            val patchedPerson =
objectMapper.treeToValue(patchedPersonJsonNode, Person::class.java)

            // updatam obiectul obtinut dupa operatia de patch
            _agendaService.updatePerson(it.id, patchedPerson)
            return ResponseEntity(Unit, HttpStatus.NO_CONTENT)
        } ?: return ResponseEntity(Unit, HttpStatus.NOT_FOUND)
}
```

Față de operațiile anterioare, operația de Patch primește în Body o listă de acțiuni, dată sub forma unui obiect JsonPatch din utilitarul adăugat [7]. Majoritatea acțiunilor sunt formate din:

- op - operația care se va face pe entitate (ex: add, replace, remove, move, test)
- path - atributul entității selectat spre modificare
- value - noua valoare a atributului

Asemănător operațiilor anterioare, este preluat obiectul cu id-ul dat. Ulterior, întrucât metoda apply a clasei JsonPatch primește ca parametru un JsonNode și returnează un JsonNode, avem nevoie de un ObjectMapper pentru a transla obiectul Person la un JsonNode și invers.

După ce s-au aplicat operațiile de patch asupra obiectului, se efectuează operația de actualizare a obiect-ului.

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 aplicaţiei cu Postman

Având de a face cu un API de tip REST, testarea aplicaţiei implementate va presupune comunicaţii HTTP, cereri client de diverse tipuri în funcţie de ceea ce se doreşte din partea server-ului. De asemenea, corpul cererilor trebuie să conţină 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 aplicaţiei **Postman**. Descărcaţi-l de aici:

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

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

```
./Postman
```

La pornire, veţi fi întrebaţi dacă doriţi să vă creaţi cont, să vă înregistraţi etc. Ignoraţi

toate aceste cereri până ajungeţi la interfaţa principală.

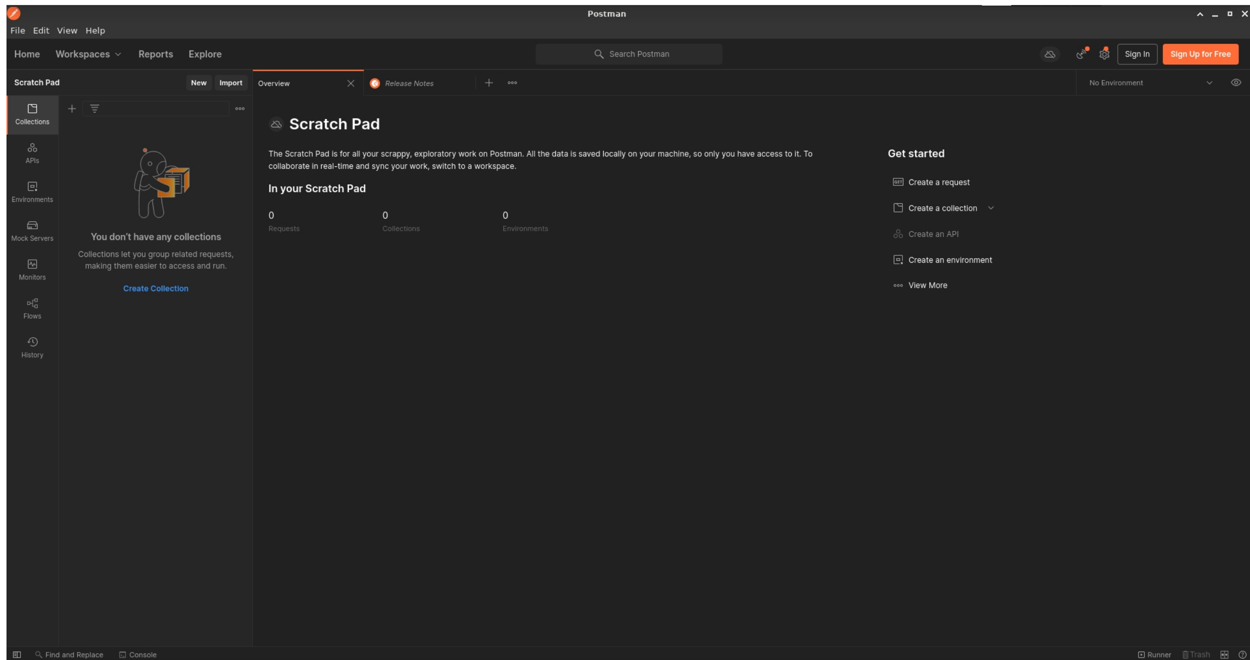


Figura 5 - Interfaţa Postman

Apăsaţi pe butonul de „New” şi selectaţi “Http Request” pentru a crea o cerere HTTP nouă.

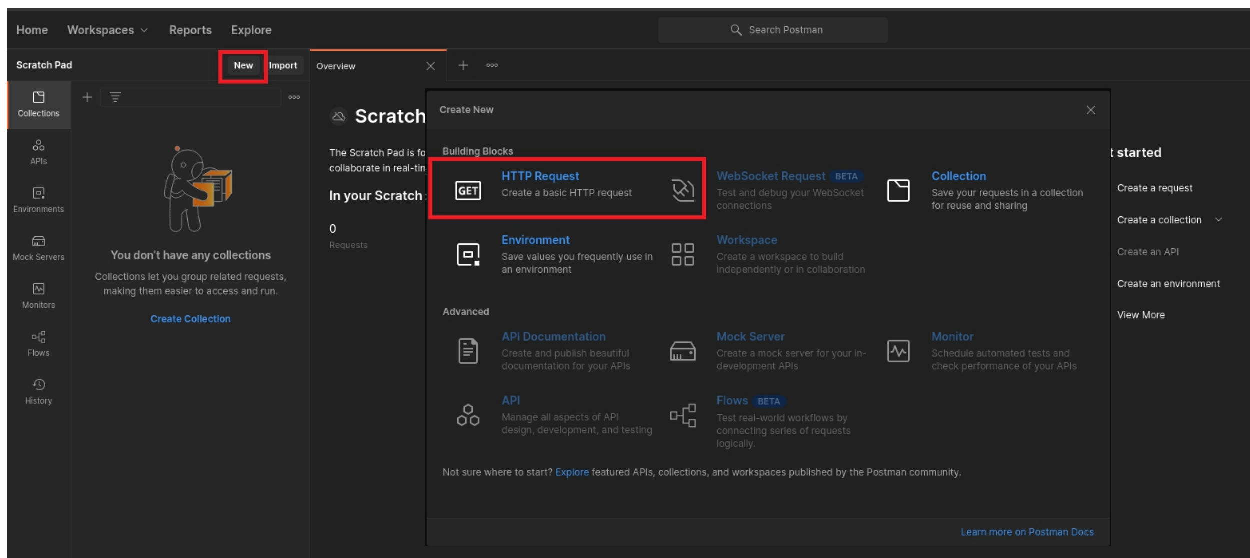


Figura 6 - Introducere cerere nouă în Postman

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

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

Pentru început, preluaţi întreaga listă de persoane din agenda telefonică (nu uitaţi să compilaţi proiectul şi să porniţi aplicaţia Spring Boot din IntelliJ).

Preluarea listei de persoane (GET)

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 și tip de cerere. Nu introduceți niciun parametru pentru moment, și apăsați butonul „**Send**” din partea dreaptă.

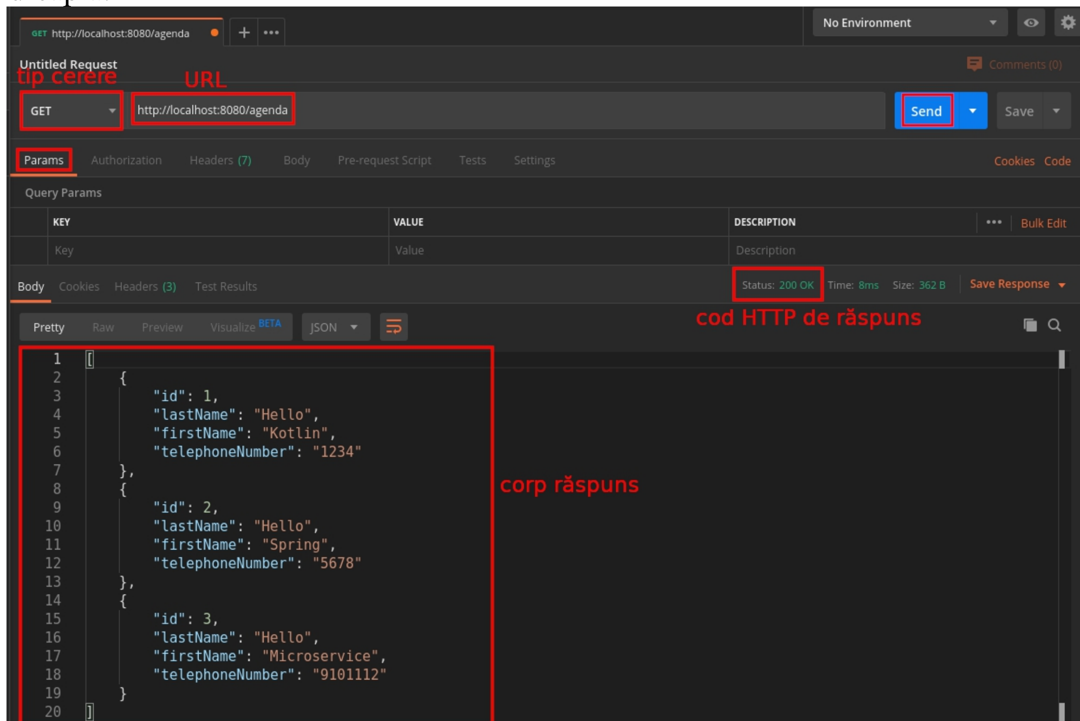


Figura 7 - Exemplu de cerere HTTP de tip GET

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`**

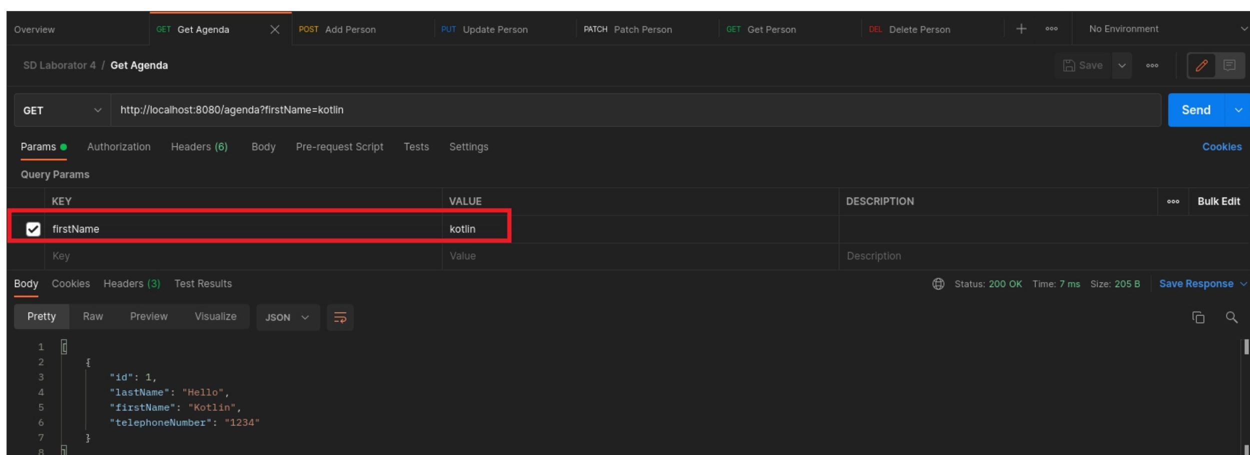


Figura 8 - Exemplu de cerere GET cu filtre

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

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

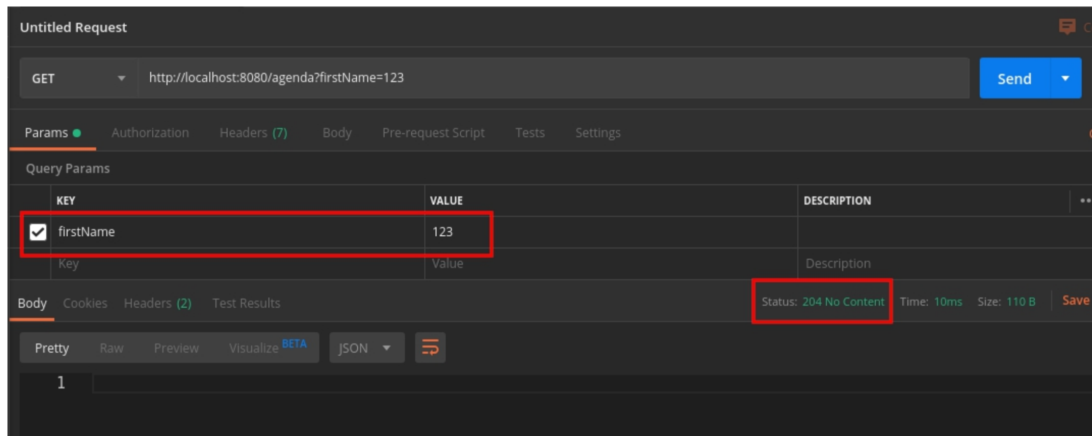
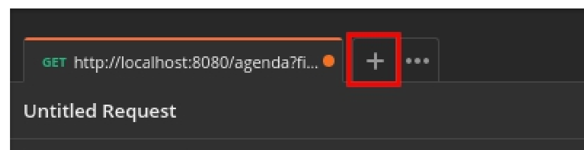


Figura 9 - Răspuns vid la cerere de tip GET

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.

Adăugarea unei persoane (POST)

Acum, încercaţi să adăugaţi o nouă intrare în agendă. Conform schemei REST a aplicaţiei, 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. Apăsaţi pe butonul „**plus**” pentru a deschide alt tab cu o cerere nouă.



Selectaţi tipul de cerere „POST” şi completaţi URL-ul: http://localhost:8080/person. Apoi, selectaţi tab-ul „**Body**”, bifaţi „**raw**” şi selectaţi ca tip de conţinut „**JSON**”. Completaţi corpul cererii astfel:

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

Apăsaţi pe butonul „**Send**” şi observaţi cum server-ul răspunde cu **`201 CREATED`**, deci resursa dorită a fost creată cu succes (vedeţi figura de mai jos).

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

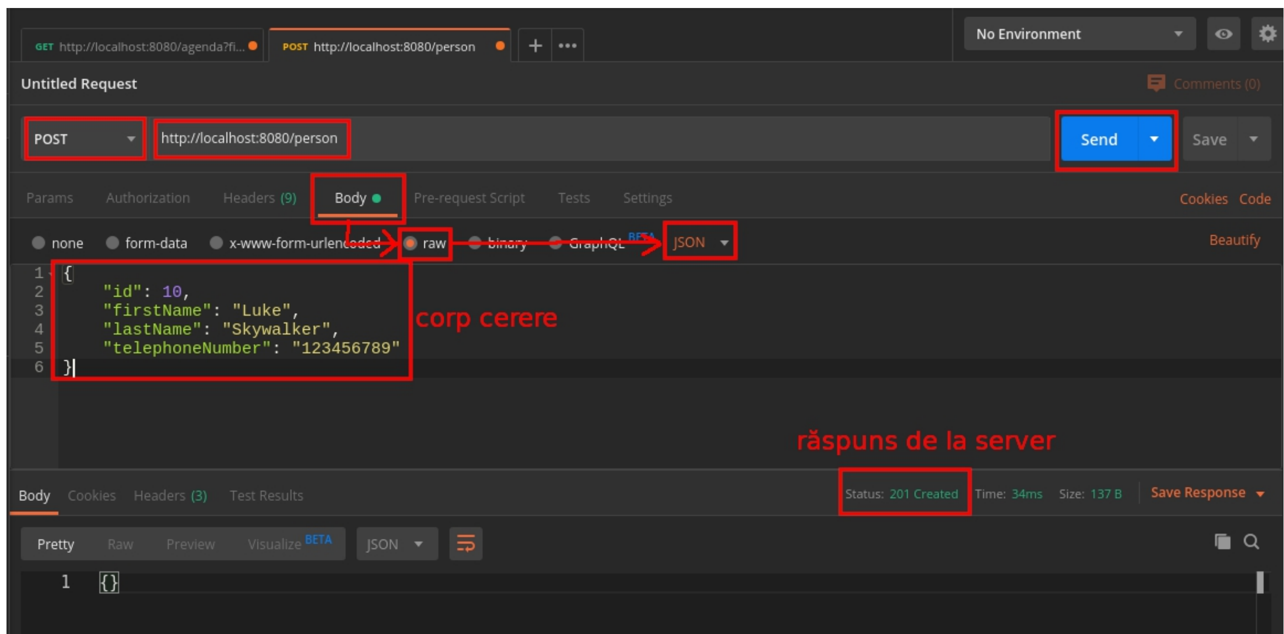


Figura 10 - Exemplu de cerere POST

Ştergerea unei persoane (DELETE)

Pentru a exemplifica şi tratarea cazurilor de excepţie, încercaţi să ştergeţi 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 colecţia de obiecte de tip **`Persoana`**.

De exemplu, ştergeţi persoana cu ID-ul **`1`**: apăsaţi din nou pe butonul „**plus**”, selectaţi **`DELETE`** ca şi tip de cerere, completaţi URL-ul cu http://localhost:8080/person/1, lăsaţi celelalte câmpuri necompletate şi apăsaţi „**Send**”.

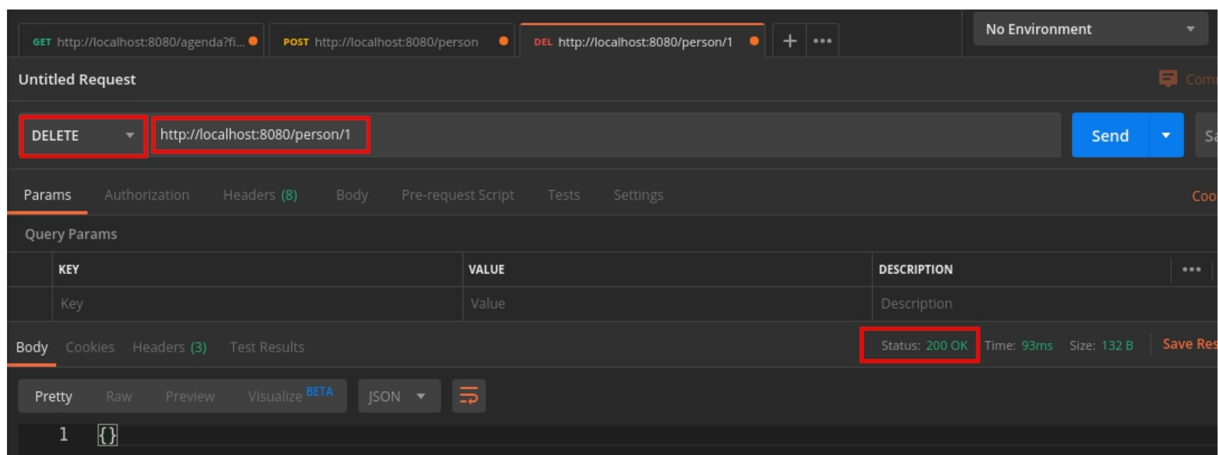


Figura 11 - Exemplu de cerere de tip DELETE

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

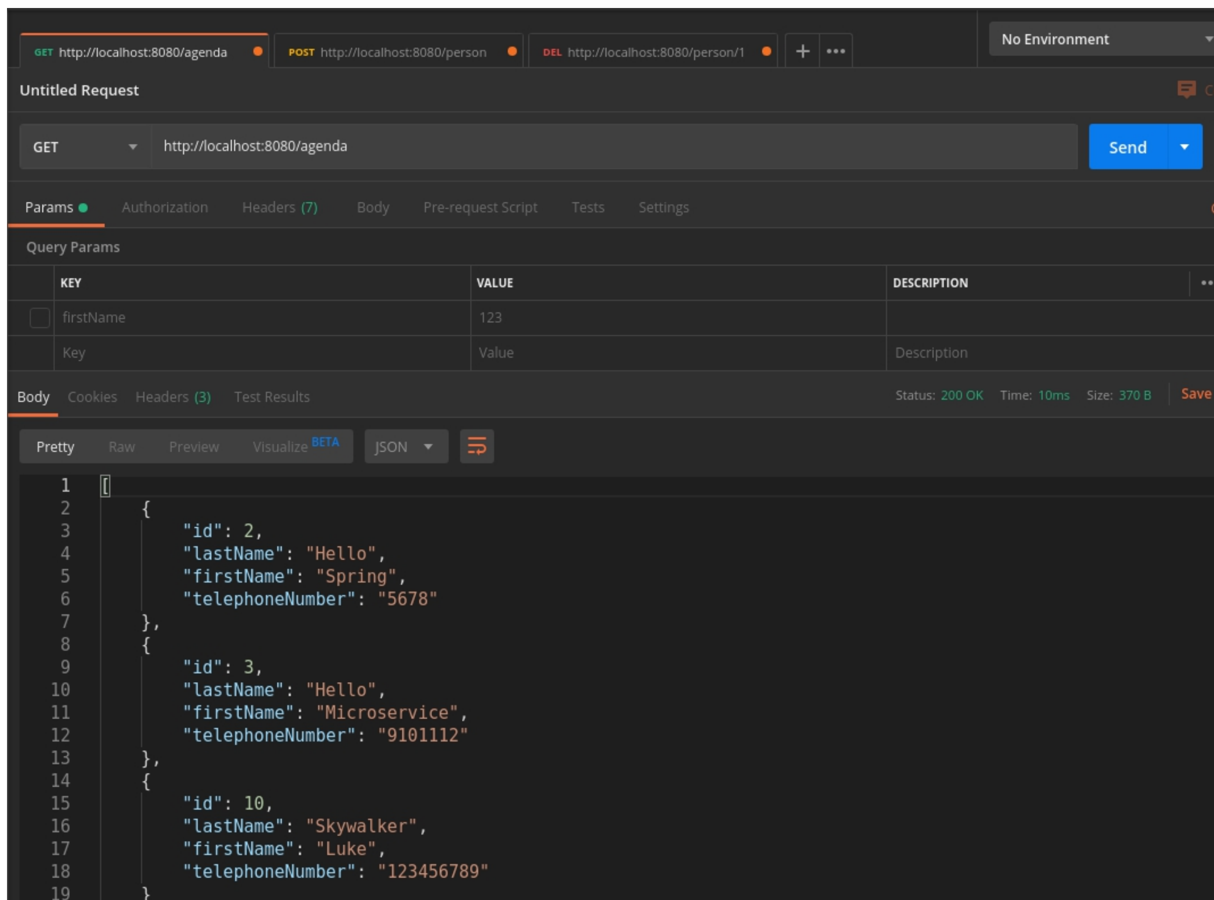


Figura 12 - Lista de persoane după ştergere

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

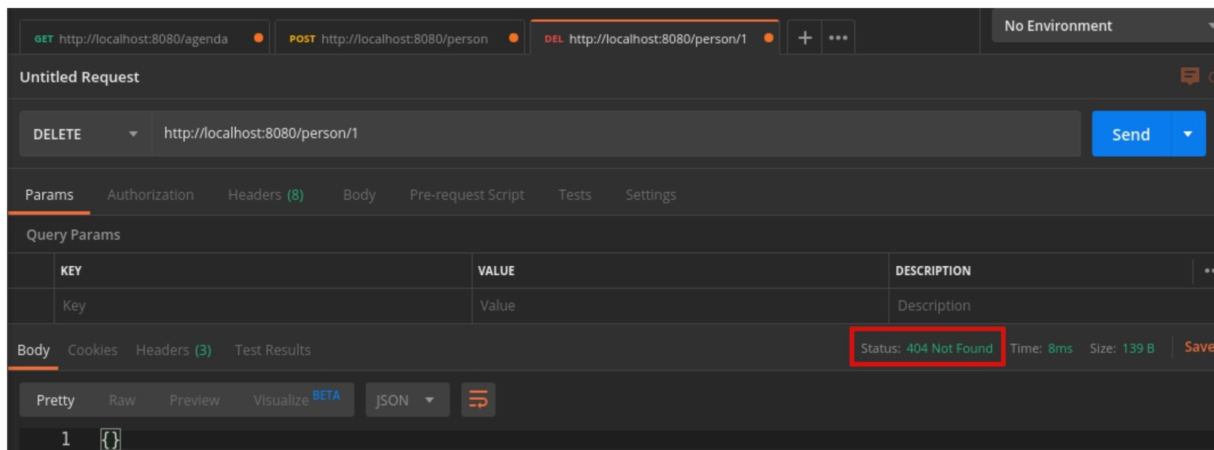


Figura 13 - Eroare la operaţia de DELETE

Actualizarea unei persoane (prin PATCH)

Operația de PATCH, după cum s-a explicat anterior, acceptă o listă de operații. În cadrul laboratorului vom testa operațiile de replace și add.

Se crează o nouă cerere în Postman de tipul PATCH. Vom actualiza persoana adăugată anterior cu id-ul 10.

Selectaţi tipul de cerere „PATCH" şi completaţi URL-ul: http://localhost:8080/person/10. Apoi,

selectaţi tab-ul „**Body**”, bifaţi „**raw**” şi selectaţi ca tip de conţinut „**JSON**”. Completaţi corpul cererii astfel:

```
[
    {
        "op":"add",
        "path":"/telephoneNumber",
        "value":"+40-202013"
    },
    {
        "op":"replace",
        "path":"/firstName",
        "value":"Mike"
    }
]
```

În urma operației, se vor actualiza atributele *firstName* și *telephoneNumber* al entității specificate în URL. Operația de *replace* în acest caz este aplicată atributului *firstName* a entității, iar operația de *add* este aplicată atributului *telephoneNumber.*

Notă: Operația de *add* din cadrul Patch este folosită în special pentru adăugarea în elemente de tip colecții. Întrucât în cazul nostru, atributul *telephoneNumber* este un șir de caractere, operația de *add* este echivalentă cu operația de *replace*.

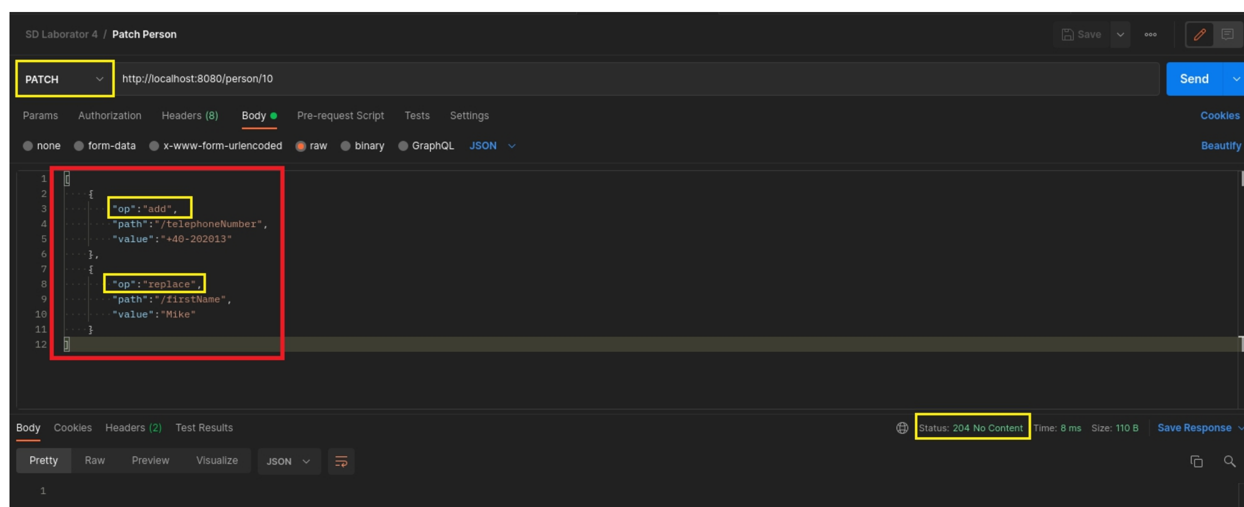


Figura 14 - Exemplu de actualizare a persoanei folosind PATCH

Pentru verificare, vom apela GET pe entitatea actualizată cu id-ul 10.

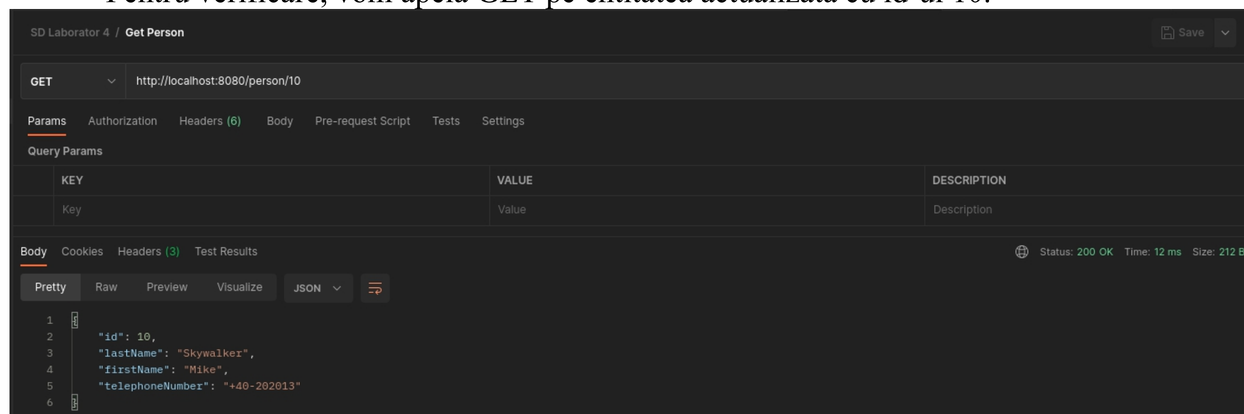
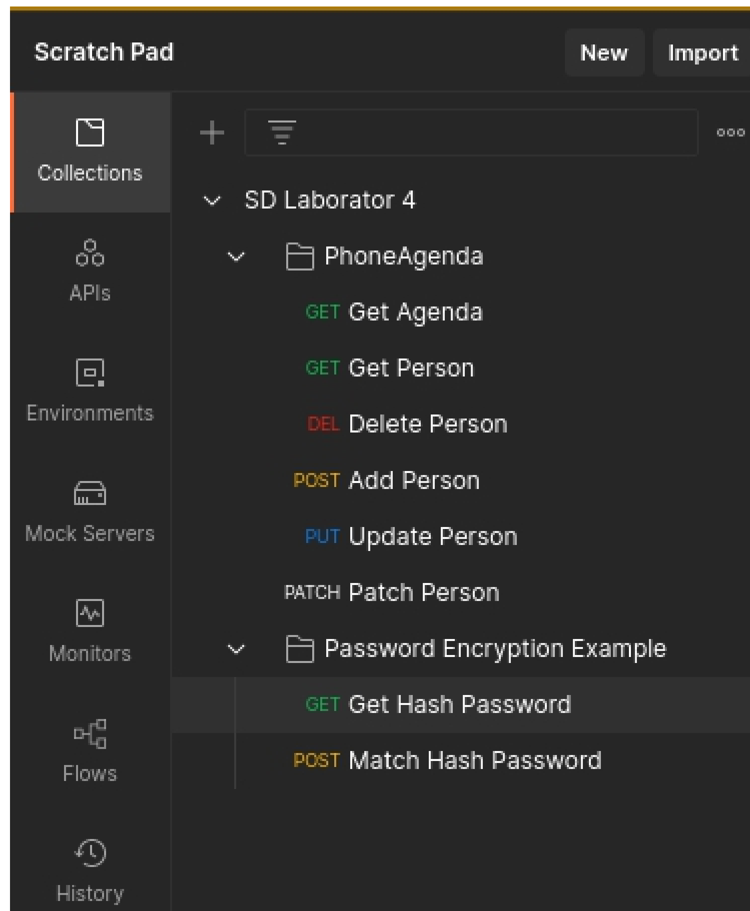


Figura 15 - Exemplu de cerere GET pe entitate

Celelalte teste pentru operaţiile şi cazurile de excepţie rămase sunt lăsate ca exerciţiu în

timpul laboratorului.

Notă: Postman permite salvarea cererilor în colecții, care pot fi exportate sub format json. Astfel se pot menține, urmări și partaja endpoint-urile testate.



Criptarea end-to-end prin HTTPS (cu TLS)

HTTP a fost folosit inițial ca protocol de comunicare pe Internet. Cu toate acestea, utilizarea crescută a HTTP pentru aplicațiile cu date sensibile a introdus nevoia de a adăuga măsuri de securitate. SSL (deja depășit) și succesorul său TLS au fost protocoale concepute pentru a oferi securitate orientată pe canalul de comunicare.

Protocolul TLS constă din 2 faze:

- stabilirea legăturii - TLS Handshake
- schimbul de date - TLS Record

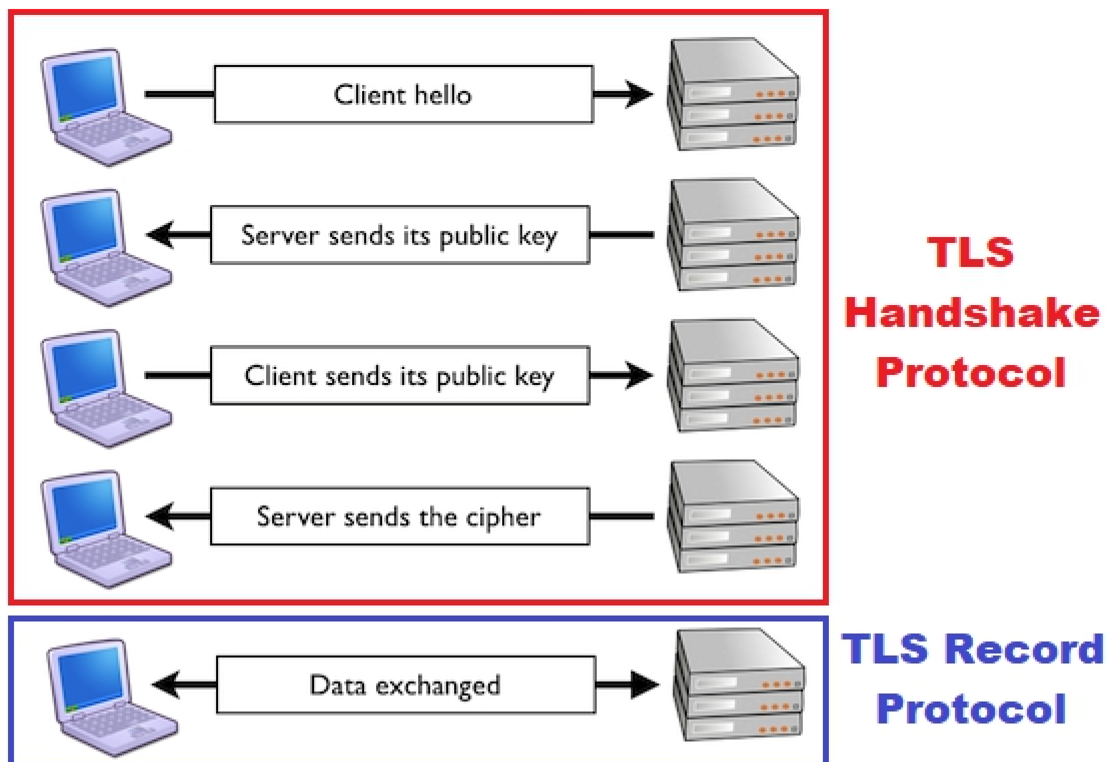


Figura 16 - Schema simplificată a procesului de comunicare prin TLS (src. cloudfare.com)

TLS Handshake Protocol

TLS Handshake este folosit în prima fază de comunicare, adică în faza de negociere a detaliilor de conexiune (aici se stabilesc versiunea de TLS utilizată în conexiune, suita de algoritmi de criptare a comunicării și ce mai este necesar.). În etapa de stabilirea a comunicației, când se face schimbul cheilor care vor fi ulterior utilizate, se folosește criptare asimetrică (de ex. algoritmul Elliptic Curve Diffie–Hellman), adică este folosită o cheie publică pentru criptarea mesajului, și o cheie privată pentru decriptarea lui.

Pasul 1: Clientul trimite un mesaj de "Hello" către server, în care este indicat versiunea de TLS și suita de algoritmi de criptare suportați de către client

Pasul 2: Serverul trimite un mesaj de "Hello" clientului în care este inclus certificatul de SSL, cheia publică a serverului și algoritmii de criptare selectați pentru comunicare. (atenție, nu confundați certificatul de SSL cu protocolul de SSL. Atât protocolul de SSL, cât și protocolul TLS folosesc un certificat de SSL)

Pasul 3: Clientul verifică validitatea certificatului de SSL.

Pasul 4: Clientul generează o cheie secretă. Această cheie este trimisă serverului într-un mesaj criptat cu cheia publică obținută la pasul anterior.

Pasul 5: Serverul decriptează mesajul cu cheia sa privată. Atât serverul, cât și clientul

folosesc cheia secretă pentru a genera aceeași cheie de sesiune.

Pasul 6: Serverul și clientul își trimit un mesaj de “Finished”pentru verificarea cheii de sesiune

TLS Record Protocol

În cadrul protocolului de TLS Record are loc transmisia mesajului în fragmente criptate cu un algoritm simetric ales la faza inițială (de ex. AES, DES, RC4).

Acest protocol asigură o conexiune securizată care este caracterizată prin două proprietăți:

- conexiunea este privată (teoretic deoarece se presupune că algoritmii de criptare simetrici folosiți nu pot fi sparți)
- conexiunea este de încredere (blocul transportat conține un mesaj de integritate obținut pe baza unor funcții de hash (de ex. SHA3 (SHA1,2 si MD5 sunt oale și ulcele de mult))

Certificatul SSL

Un certificat este o declarație semnată digital de la o entitate (persoană, companie șamd.), care ne asigură de faptul că cheia publică (și alte informații) ale unei alte entități au legătură cu viața reală - cam ca un buletin de identitate. Informațiile furnizate de acesta pot fi utilizate și în cadrul procesului de nonrepudiere a informației vehiculate. Când datele sunt semnate digital, semnătura poate fi verificată pentru a verifica integritatea și autenticitatea datelor. Integritatea înseamnă că datele nu au fost modificate sau manipulate și autenticitatea înseamnă că datele provin de sursa a cărei identitate este definită în certificat (aceasta este abordarea PKI) dar există variații fără furnizor de încredere.

Pentru ca un certificat SSL să fie valid, el trebuie să fie obținut de la o autoritate de certificare (eng. Certificate Authority sau CA). Un CA este o organizație externă, care are rolul de furnizor de încredere recunoscut de ambele părți (cum este statul în cazul buletinului), care generează și eliberează certificate SSL. CA va semna, de asemenea, digital certificatul cu propria sa cheie privată, permițând dispozitivelor client să-l verifice.

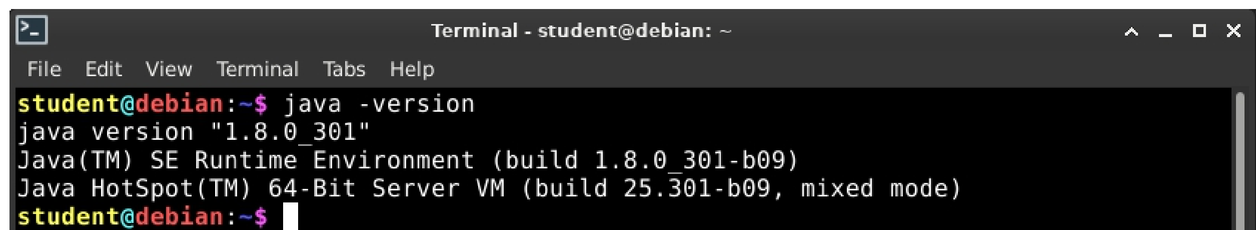
Implementare HTTPS în cadrul aplicației

Generarea certificatului SSL

În continuare vom genera un certificat SSL (self-signed), folosind utilitarul keytool din JDK1.8. Keytool este un utilitar de gestionare a cheilor și certificatelor. Aceasta permite utilizatorilor să-și administreze propriile perechi de chei publice/private și certificate asociate.

Rulați următoarea comandă pentru a verifica setarea JAVA_HOME-ului si versiunea de java configurată:

```
java -version
```



În cazul în care aveți eroarea “command not found”, instalați și configurați JDK1.8 conform laboratorului 1.

Ulterior, tastați următoarea comandă pentru generarea unei perechi de chei criptografice, care vor fi folosite pentru crearea unui certificat SSL și stocarea acestuia într-un keystore sub forma unui fișier.

```
keytool -genkeypair -alias PhoneAgenda -keyalg RSA -keysize 4096 -
storetype PKCS12 -keystore PhoneAgenda.p12 -validity 3650 -storepass
sd_lab_04 -deststoretype pkcs12
```

Parametrii utilizați în comandă au următoarea explicație:

1. **genkeypair** - operație de generare la o pereche de chei (publică și privată). Cheia publică va fi folosită în cadrul certificatului SSL
2. **keyalg** - algoritmul folosit pentru generarea cheilor
3. **keysize** - dimensiunea în biți a cheilor generate
4. **storetype** - tipul keystore-ului generat
5. **validity** - perioada de validitate în zile a certificatului generat
6. **storepass** - parola keystore-ului generat

După introducerea comenzii, introduceți informațiile cerute în formular (pot fi diferite ca cele prezentate în imagine.)

```
Terminal - student@debian: ~/Documents/temp
File  Edit  View  Terminal  Tabs  Help
student@debian:~/Documents/temp$ keytool -genkeypair -alias PhoneAgenda -keyalg RSA -keysize 4096 -storetype
 PKCS12 -keystore PhoneAgenda.p12 -validity 3650 -storepass sd_lab_04 -deststoretype pkcs12
What is your first and last name?
  [Unknown]:  student
What is the name of your organizational unit?
  [Unknown]:  AC
What is the name of your organization?
  [Unknown]:  TUIasi
What is the name of your City or Locality?
  [Unknown]:  Iasi
What is the name of your State or Province?
  [Unknown]:  Iasi
What is the two-letter country code for this unit?
  [Unknown]:  RO
Is CN=student, OU=AC, O=TUIasi, L=Iasi, ST=Iasi, C=RO correct?
  [no]:  yes

student@debian:~/Documents/temp$ ls
PhoneAgenda.p12
student@debian:~/Documents/temp$
```

Pentru verificarea conținutului fișier-ului generat tastați:

```
keytool -list -v -keystore PhoneAgenda.p12
```

```
Terminal - student@debian: ~/Documents/temp
File  Edit  View  Terminal  Tabs  Help
student@debian:~/Documents/temp$ keytool -list -v -keystore PhoneAgenda.p12
Enter keystore password:          <-- sd_lab_04
Keystore type: PKCS12
Keystore provider: SunJSSE

Your keystore contains 1 entry

Alias name: phoneagenda
Creation date: Nov 4, 2021
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=student, OU=AC, O=TUIasi, L=Iasi, ST=Iasi, C=RO
Issuer: CN=student, OU=AC, O=TUIasi, L=Iasi, ST=Iasi, C=RO
Serial number: 83d8588
Valid from: Thu Nov 04 23:13:01 EET 2021 until: Sun Nov 02 23:13:01 EET 2031
Certificate fingerprints:
         SHA1: F4:FE:64:A2:7B:0C:D8:2A:25:E4:5D:7B:AA:61:6E:81:85:23:35:66
         SHA256: 5D:9B:7D:F2:67:BC:27:7D:41:05:35:0C:22:6C:C3:BB:F5:E7:7C:E2:F5:4D:E6:C0:B6:88:24:84:8B:B0:B
B:CE
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 4096-bit RSA key
Version: 3

Extensions:

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 0D 81 1E A5 D9 BA 29 5F   24 29 13 66 2E 2A E5 E8  ......)_$).f.*..
0010: 19 79 99 01                                        .y..
]
]



*******************************************
*******************************************


student@debian:~/Documents/temp$
```

Pentru mai multe detalii despre comanda keytool, introduceți:

```
man keytool
```

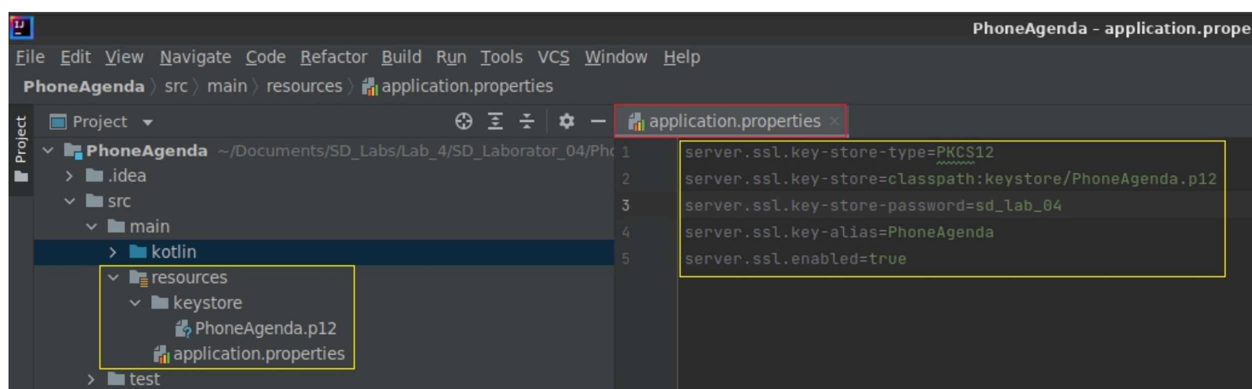
Utilizarea certificatului în aplicație

Creați în cadrul proiectului în ***src/main***, directorul ***resources***. În cadrul directorului respectiv:

- adăugați fișierul **application.properties**, care este utilizat ca fișier de configurare a serverului. Adăugați în acest fișier următoarele rânduri:

```
server.ssl.key-store-type=PKCS12
server.ssl.key-store=classpath:keystore/PhoneAgenda.p12
server.ssl.key-store-password=sd_lab_04
server.ssl.key-alias=PhoneAgenda
server.ssl.enabled=true
```

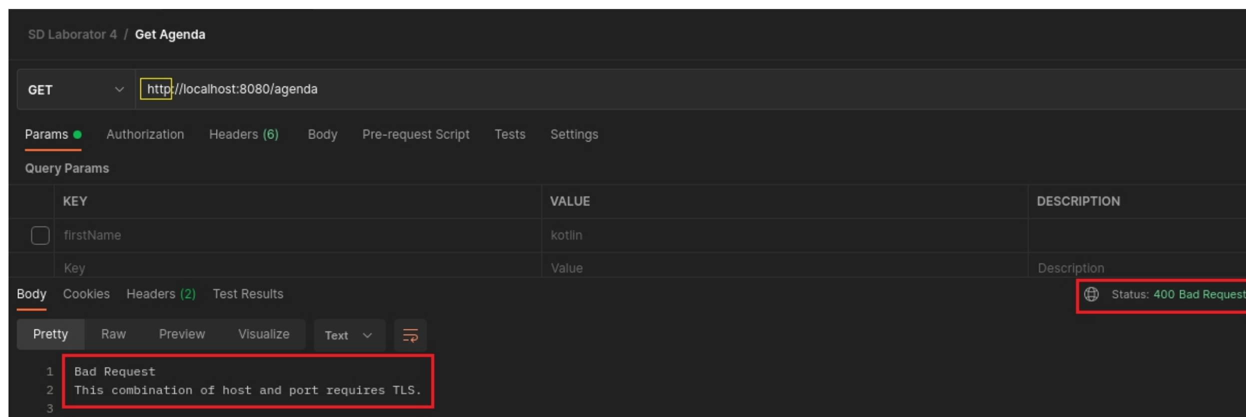
- creați folderul **keystore** în folderul **resources**, iar în el copiați fișierul generat de către utilitarul **keytool.**



În acest moment, Spring va activa conectorul de HTTPS a aplicației, iar toți pașii descriși anterior vor fi implementați de către framework.

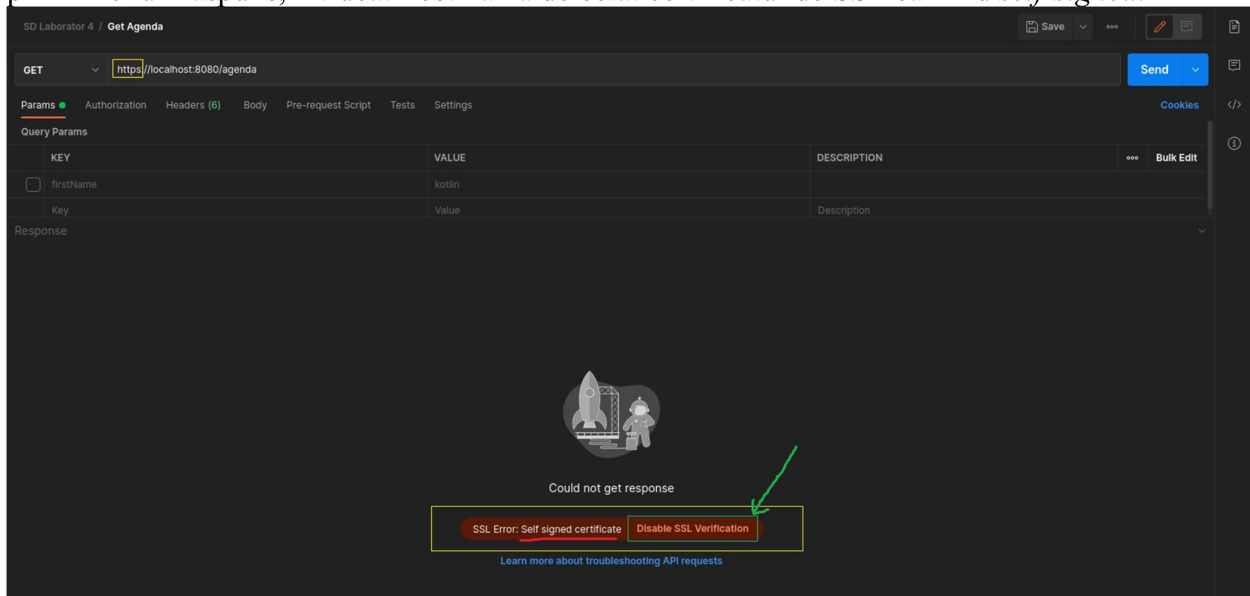
Testarea cu Postman

Dacă încercați să accesați endpoint-urile cu protocolul HTTP veți primi un răspuns de tipul Bad Request, întrucât conectorul de HTTP este deconectat (se poate implementa programatic redirectarea cererilor de tip HTTP către ruta corespunzătoare cu HTTPS).



Înlocuiți **http** din url cu **https** și trimiteți cererea. În prima fază veți observa că nu veți

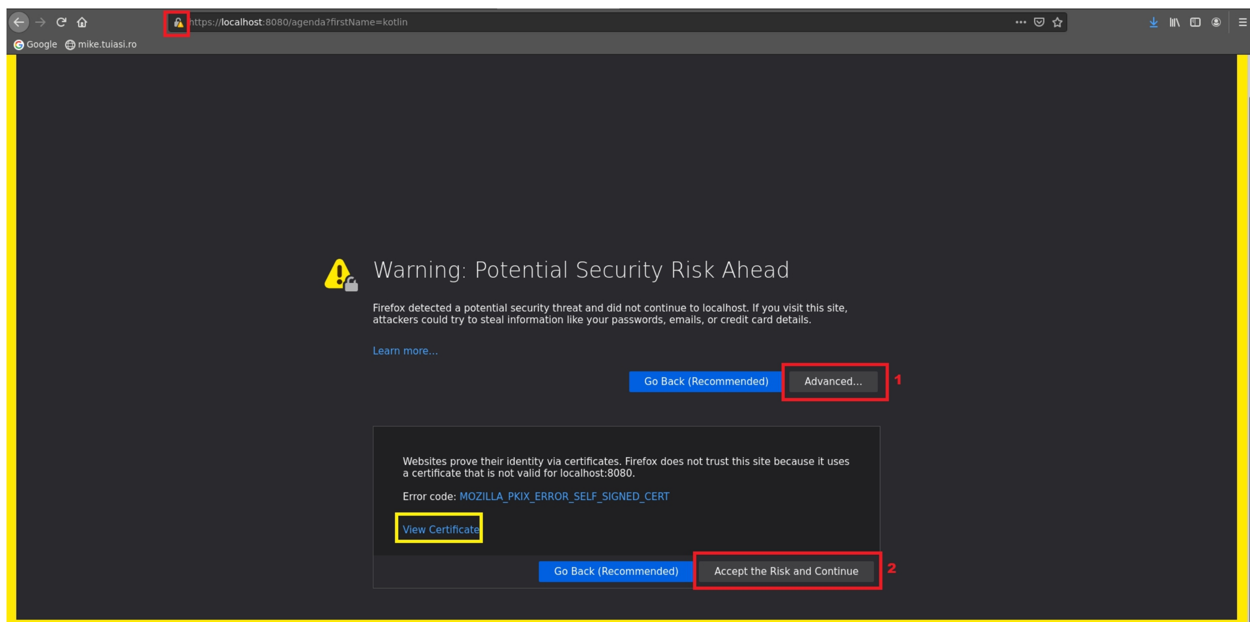
primi nici un răspuns, întrucât Postman a detectat certificatul de SSL ca fiind *self-signed*.



Apăsați "Disable SSL Verification" și retrimiteți cererea.

Testarea cu browser-ul (Firefox)

Dacă încercați să accesați un url de HTTPS din clientul de browser, veți primi inițial aceeași restricție ca în cazul Postman, și anume că certificatul SSL este invalid (nu este semnat de o autoritate recunoscută).



Dați click pe "Advanced" -> "Accept the Risk and Cotinue".

Modificați și testați celelalte endpoint-uri cu HTTPS.

Criptarea parolelor în cadrul aplicațiilor

Păstrarea în condiții sigure a parolelor utilizatorului este o componentă minimală pentru orice aplicație web. Din motive de securitate, este recomandat ca parolele să fie păstrate în forma unui hash (abordarea clasică oarecum depășită dar încă utilizată). Teoretic acest lucru previne posibilitatea ca cineva care obține acces neautorizat la baza de date să poată recupera parolele fiecărui utilizator din sistem.

Funcțiile de hash

O funcție de hash este un algoritm care preia o cantitate arbitrară de date (input) și produce o ieșire de dimensiune fixă a textului numită valoare hash.

Astfel, când un utilizator încearcă să intre într-o aplicație, este utilizată funcția de hash asupra parolei furnizate de acesta. Apoi rezultatul este comparat cu valoarea hash (asociată parolei curente) din baza de date. Dacă acestea coincid, atunci autentificarea are loc cu succes.

O funcție hash are următoarele proprietăți:

- transformare efectuată este **unidirecțională**, ceea ce înseamnă că este practic imposibil să obții valoarea de intrare din valoarea de hash obținută

- o modificare a unui bit din datele de intrare, ar trebui teoretic să genereze un rezultat imprevizibil și schimbat semnificativ (**difuzie**)

- o intrare trebuie să genereze mereu aceeași valoare hash (**determinism**)

- ar trebui să fie complicată găsirea a două valori de intrare care produc același rezultat (**rezistență la coliziune**).

- valoarea hash ar trebui sa fie neprevizibilă față de valoarea de intrare

O vulnerabilitate dată de valorile hash generate în cadrul aplicațiilor, este că utilizatorii cu aceeași parolă vor avea stocată în baza de date aceiași valoare de hash. Pentru a preveni acest lucru, la generarea valorii de hash, se va utilizat un șir de caractere generat aleatoriu (eng. salt), care va fi concatenat la parolă, și generat o valoare de hash în baza acesteia.

Aplicație demonstrativă

În continuare vom implementa o mică aplicație cu scopul de a testa:

1. aplicarea funcției de hash conform unui algoritm de hashing dat ca parametru
2. potrivirea unei valori de hash cu valoarea de intrare

Implementarea claselor

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

Modelul de structură a pachetelor este următorul:

- **business**
  - helpers
  - interfaces
  - models
  - services
- **presentation**
  - config
  - controllers
  - utils

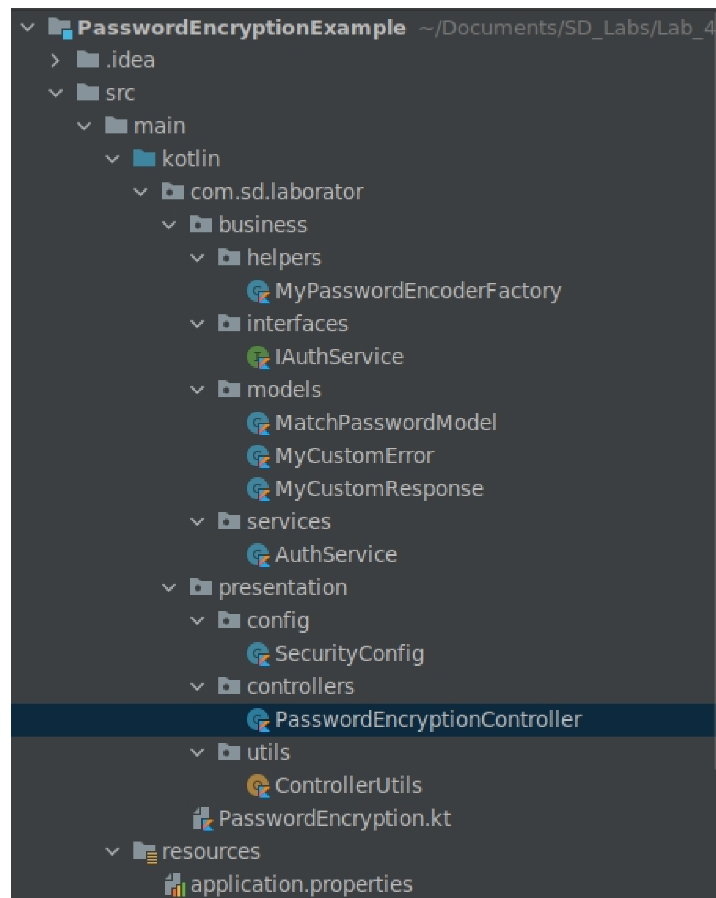


Figura 18 - Structura proiectului

Adăugați următoarea dependență (în tagul <dependencies>) în fișierul Maven pentru a include modulul de securitate. De aici vom folosi clasele pentru criptarea parolelor.

```
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

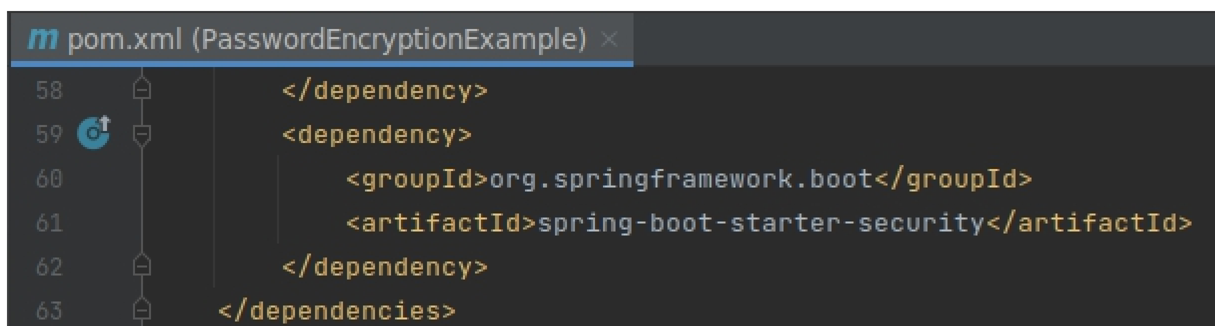


Figura 19 - Adăugarea dependenței “spring-boot-starter-security”

- clasa business.helpers.**`MyPasswordEncoderFactory`**

```
package com.sd.laborator.business.helpers

import
org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder
```

```
import org.springframework.security.crypto.password.*
import
org.springframework.security.crypto.scrypt.SCryptPasswordEncoder

class MyPasswordEncoderFactory {
    companion object{
        private var _encoders: Map<String, PasswordEncoder> =
hashMapOf(
            "bcrypt" to BCryptPasswordEncoder(),
            "ldap" to LdapShaPasswordEncoder(),
            "MD4" to Md4PasswordEncoder(),
            "MD5" to MessageDigestPasswordEncoder("MD5"),
            "noop" to NoOpPasswordEncoder.getInstance(),
            "pbkdf2" to Pbkdf2PasswordEncoder(),
            "scrypt" to SCryptPasswordEncoder(),
            "SHA-1" to MessageDigestPasswordEncoder("SHA-1"),
            "SHA-256" to MessageDigestPasswordEncoder("SHA-256"),
            "sha256" to StandardPasswordEncoder(),
        )

        fun getPasswordEncodings(): Set<String>{
            return _encoders.keys
        }

        fun getEncoder(encodingId: String): PasswordEncoder?{
            return _encoders[encodingId]
        }
    }
}
```

Această clasă o vom folosi la operația de hashing, pentru alegerea PasswordEncoder-ului dintr-un dicționar preinițializat în care cheile reprezintă algoritmi de hashing.

Ca referință s-a folosit metoda **createDelegatingPasswordEncoder** din **PasswordEncoderFactories** din cadrul modului "spring-boot-starter-security", care creează un **DelegatingPasswordEncoder,** pe care îl vom folosi la operația de potrivire a parolelor.

- clasa **`business.models.MatchPasswordModel`**

```
package com.sd.laborator.business.models

data class MatchPasswordModel(
    var password: String = "",
    var hashPassword: String = "",
    var encodingId: String = "",
)
```

Această clasă o vom folosi la operația de matching, care va fi descrisă ulterior.

- clasa **`business.models.MyCustomError`**

```
package com.sd.laborator.business.models

import java.text.SimpleDateFormat
import java.util.*
```

```
data class MyCustomError(
    var status: Int,
    var error: String?,
    var message: String?,
    var timestamp: String = SimpleDateFormat("yyyy-MM-dd
HH:mm:ss.SSS").format(Date())) {
}
```

Această clasă o vom folosi pentru tratarea datelor de intrare invalide (ex. algoritm de hashing nerecunoscut), precum și la încapsularea eventualelor excepții care pot apărea.

- clasa **business.models.MyCustomResponse**

```
package com.sd.laborator.business.models

data class MyCustomResponse<T> (
    var successfulOperation: Boolean,
    var code: Int,
    var data: T,
    var error: String? = "",
    var message: String? = "",
)
```

Clasa dată va fi folosită ca metodă de comunicare dintre nivelul de business si nivelul de prezentare a serviciului dat. Ea conține și codul HTTP returnat ca răspuns. În cazul în care va apărea o eroare/excepție, se va stabili valoarea atributului *successfulOperation la false* și se vor completa atributele error și message cu detaliile corespunzătoare. În caz de succes, *successfulOperation* va fi inițializat cu *true,* codul setat pe 2xx, iar atributul generic *data* - pe răspunsul operației.

- interfaţa **business.interfaces.IAuthService**

```
package com.sd.laborator.business.interfaces

import com.sd.laborator.business.models.MatchPasswordModel
import com.sd.laborator.business.models.MyCustomResponse

interface IAuthService {
    fun encodePassword(password: String, encodingId: String):
MyCustomResponse<Any>
    fun matchPassword(matchModel: MatchPasswordModel):
MyCustomResponse<Any>
}
```

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

- serviciul **business.services.AuthService**

```
package com.sd.laborator.business.services

import com.sd.laborator.business.interfaces.IAuthService
```

```
import com.sd.laborator.business.helpers.MyPasswordEncoderFactory
import com.sd.laborator.business.models.MatchPasswordModel
import com.sd.laborator.business.models.MyCustomResponse
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.security.crypto.password.PasswordEncoder
import org.springframework.stereotype.Service


@Service
class AuthService: IAuthService {
    @Autowired
    private lateinit var _delegatingPasswordEncoder: PasswordEncoder

    override fun encodePassword(password: String, encodingId: String):
MyCustomResponse<Any>{
        try{
            // returneaza encoder-ul de parole conform algoritmului
specificat
            val passwordEncoder =
MyPasswordEncoderFactory.getEncoder(encodingId)
            if (passwordEncoder != null){
                return MyCustomResponse(
                    successfulOperation = true,
                    code=200,
                    data=passwordEncoder.encode(password)
                )
            }

            //id invalid al encoder-ului specificat
            return MyCustomResponse(
                successfulOperation = false,
                code = 400,
                data = Unit,
                error = "Invalid encodingId=${encodingId}.",
                message = "Choose from:
${MyPasswordEncoderFactory.getPasswordEncodings().joinToString(", ")}",
            )
        } catch (e: Exception){
            return MyCustomResponse(
                successfulOperation = false,
                code = 500,
                data = Unit,
                error = e.message
            )
        }
    }

    override fun matchPassword(matchModel: MatchPasswordModel):
MyCustomResponse<Any> {
        return try{
            // adaugam encodingId-ul in parola hash-uita pentru a fi
delegat un PasswordEncoder corespunzator
            val currentHashPasswordWithAlg =
"{${matchModel.encodingId}}${matchModel.hashPassword}"

            MyCustomResponse(
```

```
                successfulOperation = true,
                code=200,
                data= hashMapOf("areMatched" to
_delegatingPasswordEncoder.matches(matchModel.password,
currentHashPasswordWithAlg))
            )
        } catch (e: Exception){
            MyCustomResponse(
                successfulOperation = false,
                code = 500,
                data = Unit,
                error = e.message
            )
        }
    }
}
```

În cadrul acestei clase injectarea atributului *_delegatingPasswordEncoder* se va face în clasa *SecurityConfig* pe care o vom explica ulterior. Atributul dat va fi folosit la operația de potrivire a parolei originale cu o valoare hash dată ca un parametru.

Operația de verificare a potrivirii parolei originale cu valoarea de hash dată ca un parametru

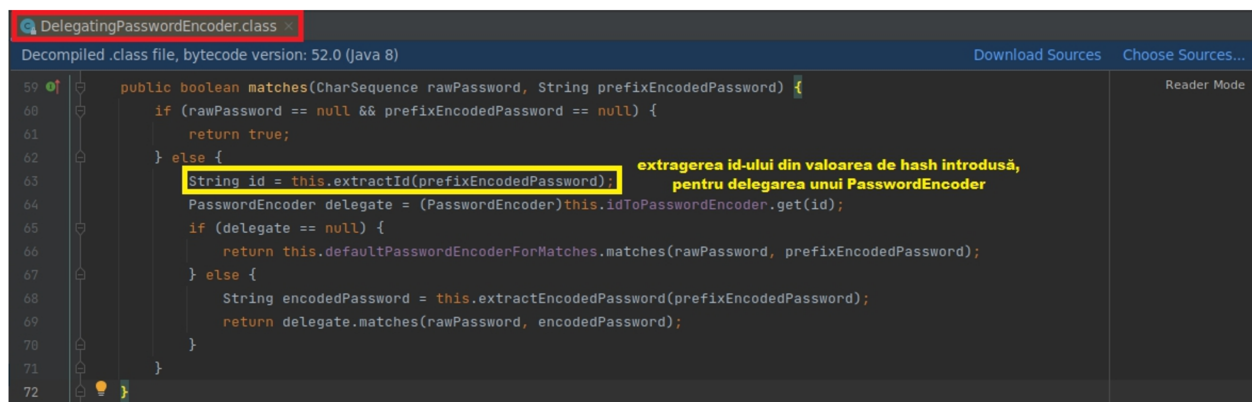


Figura 20 - Metoda *matches* din clasa decompilată *DelegatingPasswordEncoder*

Dacă atragem o privire la clasa decompilată **DelegatingPasswordEncoder**, observăm că în cadrul operației de *matches* este extras id-ul algoritmului folosit, care va fi utilizat ulterior pentru a delega un PasswordEncoder corespunzător. Această valoare de hash trebuie să aibă forma **"${encodingId}${actualHash}"** , unde *encodingId* poate lua o valoare din mulțimea cheilor dicționarului clasei **MyPasswordEncoderFactory,** care a fost explicat anterior.

În cadrul metodei de *matchPassword* este adăugat encodingId-ul la valoarea de hash dată ca intrare, pentru a putea fi recunoscut de către delegatul de PasswordEncoder. Ulterior este apelată operația de *matches* a acestui delegat. În funcție de execuția operației, este încapsulat răspunsul sub forma unui obiect *MyCustomResponse* ,

Operația de criptare a parolei cu un algoritm dat ca parametru

În cadrul acestei operații, este extras PasswordEncoder-ul corespunzător *encodingId*-ului dat ca parametru, folosind fabrica *MyPasswordEncoderFactory*. În cazul în care id-ul dat ca parametru este invalid, este returnat un răspuns cu codul 400 și un mesaj corespunzător (încapsulat bineînțeles într-un obiect MyCustomResponse). În caz de succes este returnat hash-ul

obținut.

- clasa de configurări **presentation.config.SecurityConfig**

```
package com.sd.laborator.presentation.config

import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.http.HttpMethod
import
org.springframework.security.config.annotation.web.builders.HttpSecuri
ty
import
org.springframework.security.config.annotation.web.configuration.WebSe
curityConfigurerAdapter
import
org.springframework.security.crypto.factory.PasswordEncoderFactories
import org.springframework.security.crypto.password.PasswordEncoder


@Configuration
class SecurityConfig : WebSecurityConfigurerAdapter() {
    @Bean
    fun passwordEncoder(): PasswordEncoder {
        return
PasswordEncoderFactories.createDelegatingPasswordEncoder()
    }

    override fun configure(http: HttpSecurity) {
        http
            // Disable la "cross site request forgery" pentru a
permite operatiile de POST/PUT/PATCH
            .csrf().disable()

            // Aici puteti autoriza anumite rute (ex. pe baza unui
token).
            // In cazul nostru sunt permise accesul endpoint-urilor
fara autorizare
            .authorizeRequests()
            .antMatchers(HttpMethod.GET, "/**").permitAll()
            .antMatchers(HttpMethod.DELETE, "/**").permitAll()
            .antMatchers(HttpMethod.POST, "/**").permitAll()
            .antMatchers(HttpMethod.PUT, "/**").permitAll()
            .antMatchers(HttpMethod.PATCH, "/**").permitAll()
    }
}
```

În acest fișier a fost definit *Bean*-ul pentru atributul *_delegatingPasswordEncoder* al serviciului **AuthService** explicat anterior. De asemenea, în acest fișier a fost configurată autorizarea rutelor (în cazul nostru nu va exista nici o autorizare, deci au fost marcate cu *permitAll*).

- clasa **presentation.utils.ControllerUtils**

```
package com.sd.laborator.presentation.utils

```

```
import com.sd.laborator.business.models.MyCustomError
import com.sd.laborator.business.models.MyCustomResponse
import org.springframework.http.ResponseEntity

object ControllerUtils {
    // Construieste mesajul in functie de
MyCustomResponse.successfulOperation
    // Daca successfulOperation = true, construieste cu data din mesaj
    // Daca successfulOperation = false, construieste mesajul de
eroare
    fun makeResponse(response: MyCustomResponse<Any>):
ResponseEntity<Any> {
        return if (response.successfulOperation)

ResponseEntity.status(response.code).body(response.data)
        else {
            val err = MyCustomError(response.code, response.error,
response.message)
            ResponseEntity.status(response.code).body(err)
        }
    }
}
```

Această clasă statică este utilizată în controller, și are rolul de a construi răspunsul HTTP pe baza clasei generice definite de noi *MyCustomResponse*. În cazul în care *successfulOperation* este inițializat cu *false*, este creat în corpul mesajului întors, un obiect de tipul *MyCustomError* (asemănător celui returnat de Spring Boot).

- controllerul **presentation.controllers.PasswordEncryptionController**

```
package com.sd.laborator.presentation.controllers

import com.sd.laborator.business.interfaces.IAuthService
import com.sd.laborator.business.models.MatchPasswordModel
import com.sd.laborator.presentation.utils.ControllerUtils
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.*


@RestController
class PasswordEncryptionController {
    @Autowired
    private lateinit var _authService: IAuthService

    @RequestMapping(value = ["/encrypt"], method = [RequestMethod.GET])
    fun encryptPassword(
        @RequestParam() password: String,
        @RequestParam() encodingId: String): ResponseEntity<Any>
    {
        val response = _authService.encodePassword(password,
encodingId)
        return ControllerUtils.makeResponse(response)
    }
```

```
    @RequestMapping(value = ["/match"], method = [RequestMethod.POST])
    fun matchPassword(@RequestBody matchModel: MatchPasswordModel):
ResponseEntity<Any>
    {
        val response = _authService.matchPassword(matchModel)
        return ControllerUtils.makeResponse(response)
    }
}
```

Clasa dată este controller-ul aplicației, și în ea sunt definite endpoint-urile pentru cele două operații explicate anterior.

- fișierul **application.properties**

```
server.port=2030
```

De data aceasta, vom schimba portul aplicației pe 2030. În cazul anterior, aplicația se executa pe portul configurat implicit, și anume 8080.

Punctul de intrare al aplicației Spring - **PasswordEncryption.kt**

```
package com.sd.laborator

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

@SpringBootApplication
class PasswordEncryption

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

Notă informativă: Majoritatea obiectelor de *PasswordEncoder*, returnate de fabrica *MyPasswordEncoderFactory* (conform clasei decompilate *PasswordEncoderFactories*) sunt marcate ca fiind *deprecated.* Astfel nu este recomandată utilizarea lor în aplicațiile reale, datorită faptului că nu sunt sigure. În schimb, conform documentației oficiale, este recomandată utilizarea următoarelor clase de tip *PasswordEncoder*: *BCryptPasswordEncoder, Pbkdf2PasswordEncoder sau SCryptPasswordEncoder.*

O alegere mai bună se consideră a fi *DelegatingPasswordEncoder*, întrucât poate fi configurat să folosească la operația de hashing un algoritm dorit, iar la operația de matching acesta deleagă un *PasswordEncoder* în funcție de tipul hash-ului dat spre potrivire.

Testarea aplicației

Testarea se va face prin intermediul aplicației Postman. Se recomandă crearea unei noi colecții, în care vor fi salvate cele 2 cereri pentru aplicația curentă.

- operația de criptare a parolei cu un algoritm dat ca parametru

Creați un nou *request* în Postman de tipul GET, iar la URL-ul cererii introduceți: http://localhost:2030/encrypt?password=Imi place SD&encodingId=SHA-256

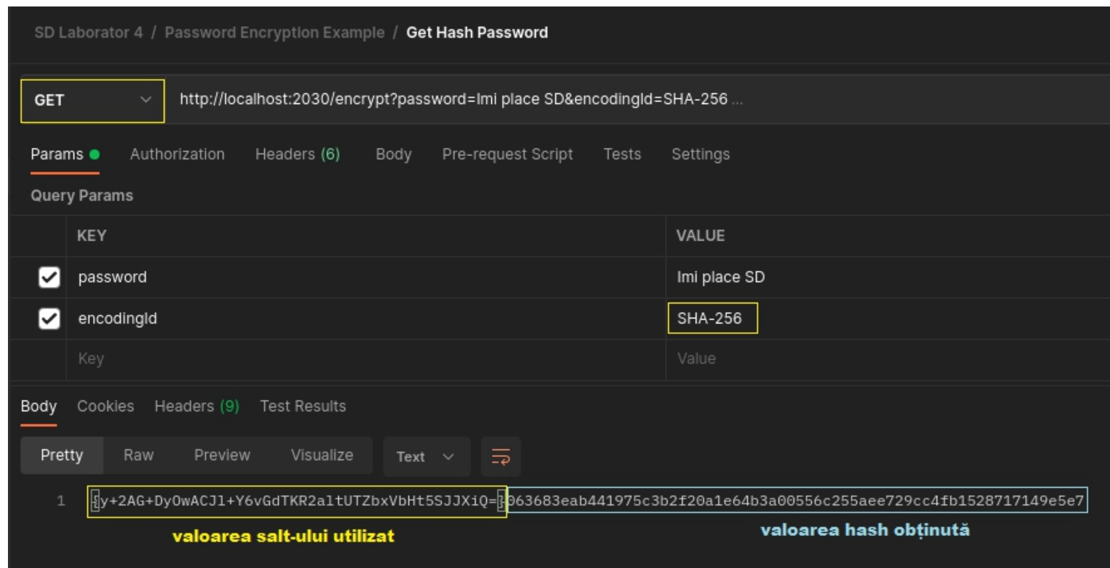


Figura 21 - Exemplu de criptare a parolei folosind funcția de hash SHA-256

În acest exemplu, răspunsul are forma "{$salt}{$hash}", unde *salt* este un șir generat aleatoriu (după cum am precizat anterior) de către *PasswordEncoder*-ul din modulul de securitate adăugat.

În cazul în care testăm cu un *encodingId* invalid, vom primi răspunsul încapsulat într-un obiect *MyCustomError*.

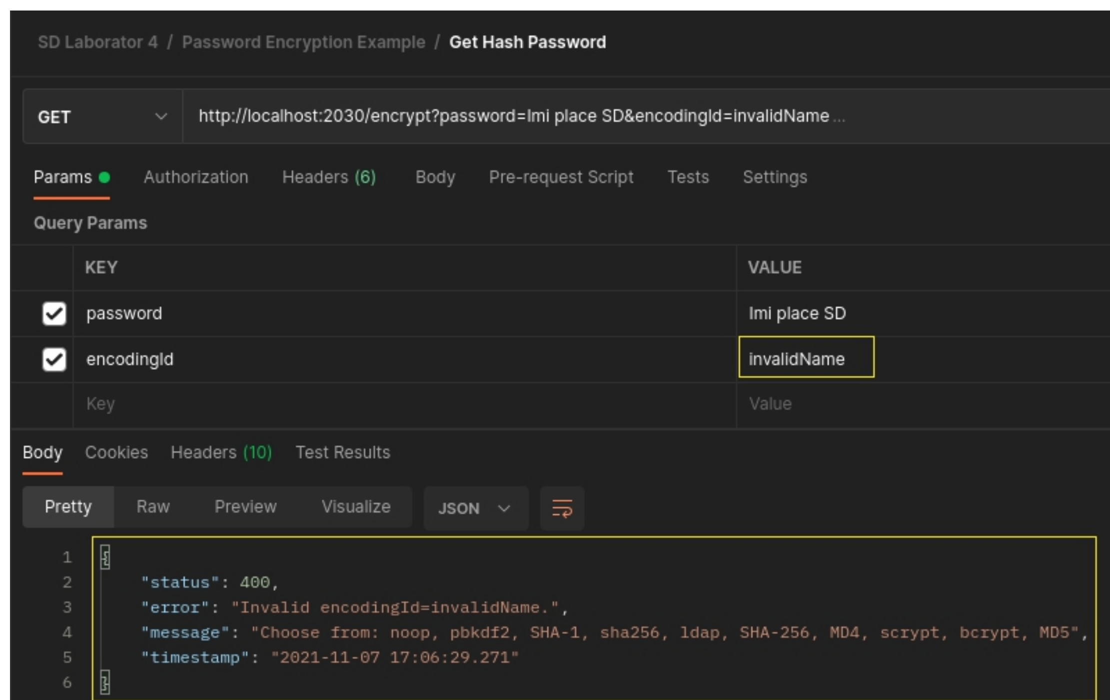


Figura 22 - Exemplu de cerere cu parametri invalizi

- operația de potrivire a parolei

  Creați altă cerere în Postman de tipul POST, către endpoint-ul http://localhost:2030/match. În corpul cererii puneți un hash obținut la pasul anterior, împreună cu parola originală și *encodingId*-ul folosit. Exemplu:

```
{
    "password": "Imi place SD",
    "hashPassword":
"{y+2AG+DyOwACJl+Y6vGdTKR2altUTZbxVbHt5SJJXiQ=}063683eab441975c3b2f20a1e64b3a00556c255aee729cc4fb1528717149e5e7",
    "encodingId": "SHA-256"
}
```

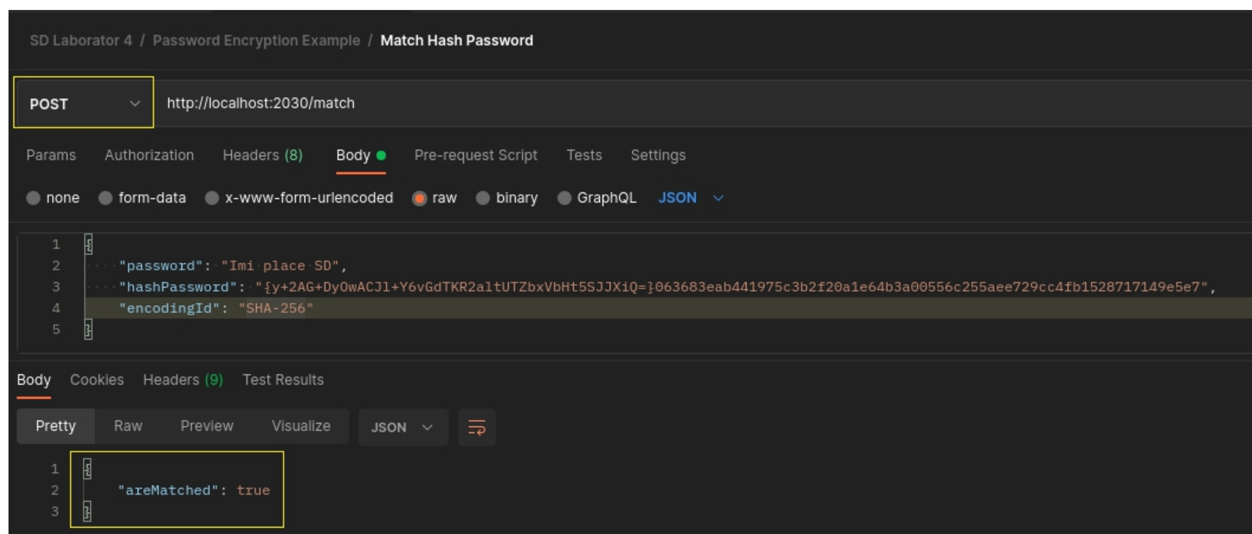


Figura 22 - Exemplu de cerere de potrivire

Dacă veți modifica un caracter din parola originală, sau din hash, atunci potrivirea nu va avea loc cu succes.

Testați și celelalte funcții de hash, definite în clasa *MyPasswordEncoderFactory*.

Aplicaţii şi teme

Teme de laborator

1. Analizati codul exemplu al aplicaţiei Agendă şi apoi extrageţi atât diagrama de servicii (conform cu regulile de reprezentare prezentate la curs. Apoi extrageţi şi diagrama de clase respectând regulile din standardul UML. Realizaţi o analiză privind respectarea principiilor SOLID.
2. Adăugaţi o interfaţă grafică simplă bazată pe un formular HTML, cu care să puteţi implementa operaţiile GET şi POST. Modificaţi apoi HTML-ul astfel încât să se poată trimite către program interogări HTTP de tip PUT şi DELETE (cu iubitul vostru JavaScript - folosiţi fetch sau AJAX).

Temă pentru acasă

Proiectaţi (deci diagramă de servicii şi diagramă de clase) şi implementaţi o aplicaţie cu servicii RESTful pentru gestiunea cheltuielilor curente pentru membrii a unei familii. Fiecare membru va avea contul lui accesibil printr-un nume și parolă. Vor exista următoarele categorii de cheltuieli: întreținere, mâncare, distracție, scoală și personale (e.g. îmbrăcat, încălțat, tuns...). Se va mai crea un serviciu suplimentar care primește un şir de date, în criptează cu un algoritm simetric din bibliotecă (cum ar fi AES) și întoarce rezultatul. Acest serviciu va fi utilizat pentru criptarea datelor personale care intră sub incidența GDPR (minimal numele și prenumele) înainte de a fi salvate în baza de date. Parola va fi un hash (alegeţi voi o funcţie din bibliotecă) asupra numelui de intrare în cont combinat cu parola. Pentru aplicaţia respectivă, creaţi şi o interfaţă folosind framework-ul Flask din Python.

Documentaţie utilă:

- documentaţia 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
[7]: JsonPatch - http://jsonpatch.com/
[8] : TLS Protocol (RFC-5246) - https://datatracker.ietf.org/doc/html/rfc5246
[9]: About Password Hashing - https://docs.oracle.com/cd/E63029_01/books/Secur/secur_secadaptauth015.htm
[10]: Delegating Password Encoder - https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/password