**Create REST API cu @Controller**

* @RestController e creata din 2 anotatii:

@Controller

@ResponseBody – anotatia data spune ca Controllerul nu va returna pagini web, ci obiecte pe care le va transforma automat in JSON si le va pune si returna intr-un obiect de tip HttpResponse(extensia lui e ResponseEntity)

* @RestController tot vine din Spring MVC

**@PathVariable vs @RequestParam**

* @PathVariable – extrage datele din URI path, cele puse dupa / care formeaza linkul, numite **template variable**
* @GetMapping("/user/{id}")  
  public void pathVar(@PathVariable int id){  
   System.*out*.println(id);  
  }

[localhost:8080/user/10](http://localhost:8080/test/user/10)

* @RequestParam – extrage variabilele din query string, adica din parametrii trimisi dupa ?
* @GetMapping("/test/user")  
  public void param(@RequestParam("userId")int id){  
   System.*out*.println(id);  
  }

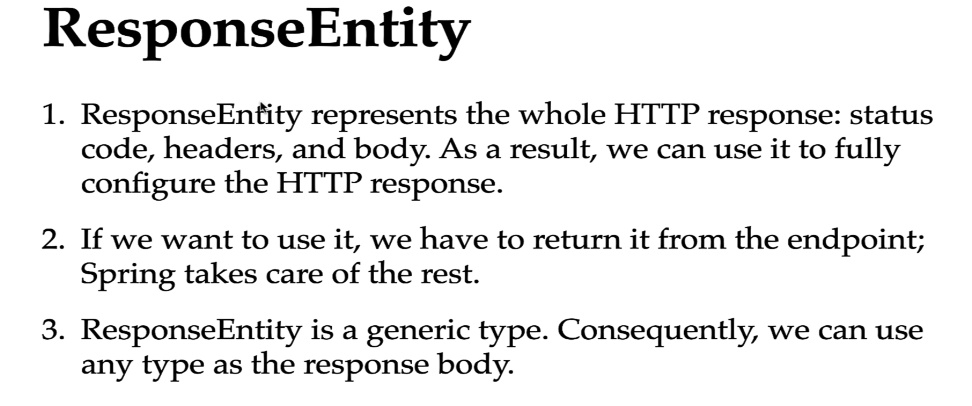
[localhost:8080/test/user?userId=100](http://localhost:8080/test/user?userId=100)

* Parametrii dupa ? se numesc **query parameters**

**@ResponseStatus**

* Daca noi nu returnam niciun status in metodele de la rest controller class, cu un ResponseEntity, atunci Spring automat va trimite un 200 response
* Alta metoda, decat sa folosim un ResponseEntity este de a folosi anotatia @ResponseStatus

**ResponseEntity**



new ResponseEntity<>(object, 200)

object – va fi pus in body la HTTP Response

* ResponseEntity class ofera si niste metode statice, unde nu mai trebuie sa returnam status code, ca:

- ok(T body)

- ok()

- accepted()

- noContent()

- ofNullable(T body)

- badRequest()

- notFound()

.....

@GetMapping("/test/user")  
public ResponseEntity<String> param(@RequestParam("userId")int id){  
 System.*out*.println(id);  
  
 return ResponseEntity.*ok*("Success");  
}

**Http response/request componente**

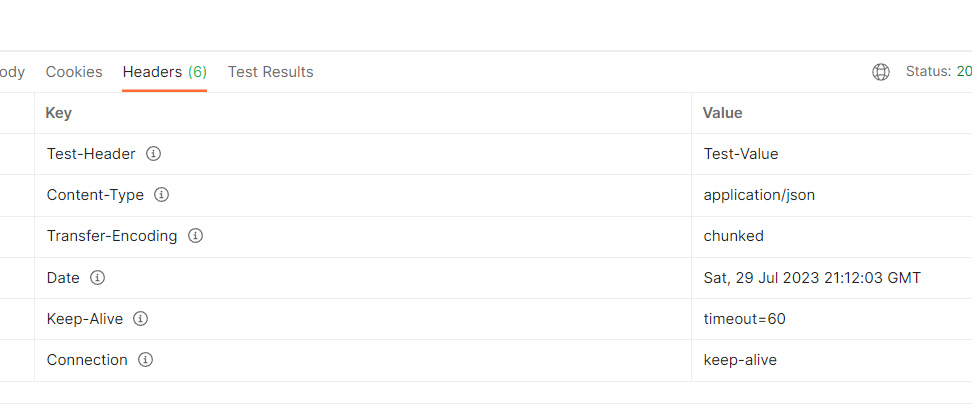
* Status Code
* Body – obiectul returnat
* Header

**Header**

* Header ofera mai multe informatii despre request, putand fi setate de cel care creaza request.
* Header este folosit pentru a oferi mai multe informatii in http response/request despre response/request
* De exemplu, browserul la client trimite in request create, in header, numele la browser, OS si alte detalii care pot ajuta Serverul unde se duce request sa ofere o mai buna comunicare sau activitate la user
* Al exemplu este sa spunem in request ce tip de raspuns vrem, gen TEXT, JSON etc.
* Server poate trimite in header niste date pe care browser sa le foloseasca.
* ResponseEntity nu are propriu zis metode de a seta headers, dar BodyBuilder o poate face, si apoi BodyBuilder ne poate returna un ResponseEntity
* De ex:
* @GetMapping("/test/user")  
  public ResponseEntity<Integer> param(@RequestParam("userId")int id){  
   System.*out*.println(id);  
   return ResponseEntity.*ok*()  
   .header("Test-Header","Test-Value")  
   .body(id);  
  }

metoda ResponseEntity.ok() returneaza un **BodyBuilder** ce ofera metoda **header(String headerName, String... headerValues)** si putem seta numele pentru headers si valori

* Ele vor fi returnate aici:

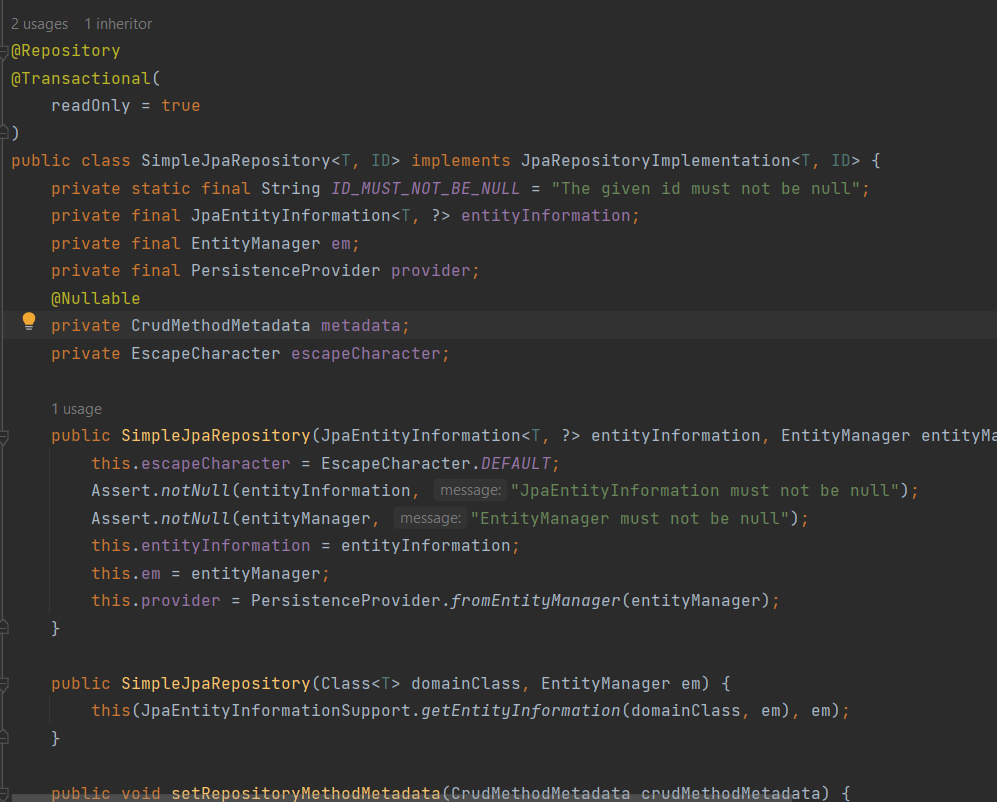




* ResponseEntity foloseste BodyBuilder, care e o interfata, dar il implementeaza in DefaultBuilder inner class, care e privata, deci putem trimite Header doar instantiind ResponseEntity

**SimpleJpaRepository**

O implementare pentru JpaRepository este SimpleJpaRepository

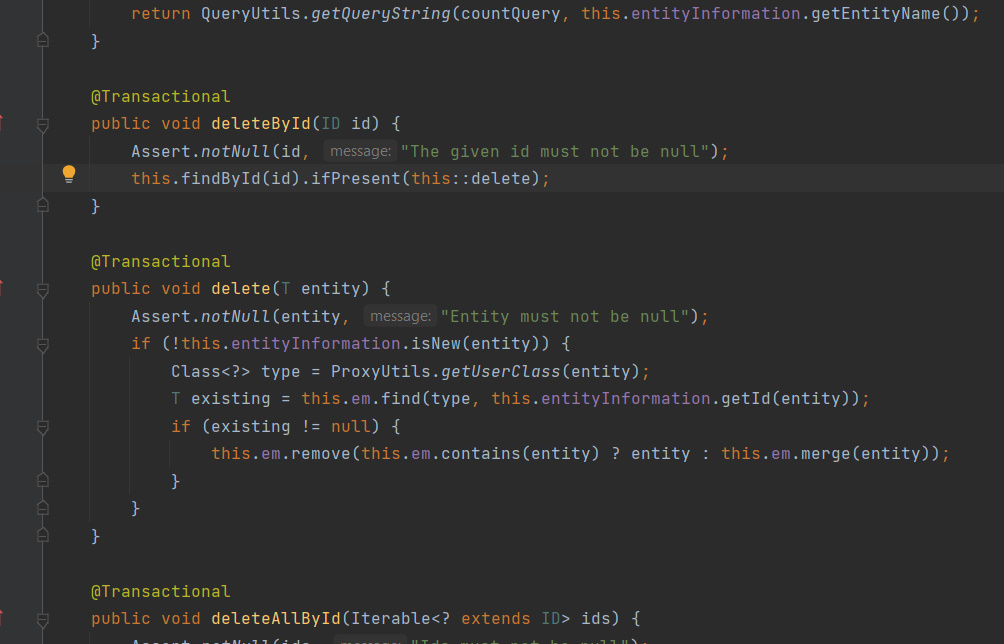


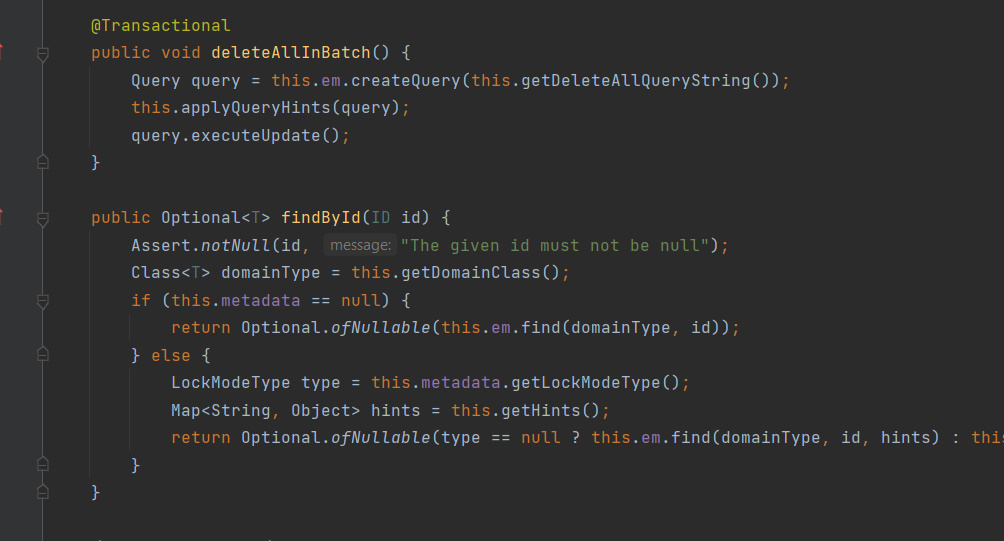
- Vedem ca foloseste EntityManager

- Deja are @Repository, deci nu mai anotam si interfata noastra cu @Repository

- @Transactional are readOnly = true, deci metodele care nu definesc @Transactional sunt automat doar readOnly. Iata de ce, daca cream o metoda de a noastra in JpaRepository creat, ea va fi automat cu anotatia @Transactional(readOnly = true), asa ca va trebuie sa o punem noi chiar

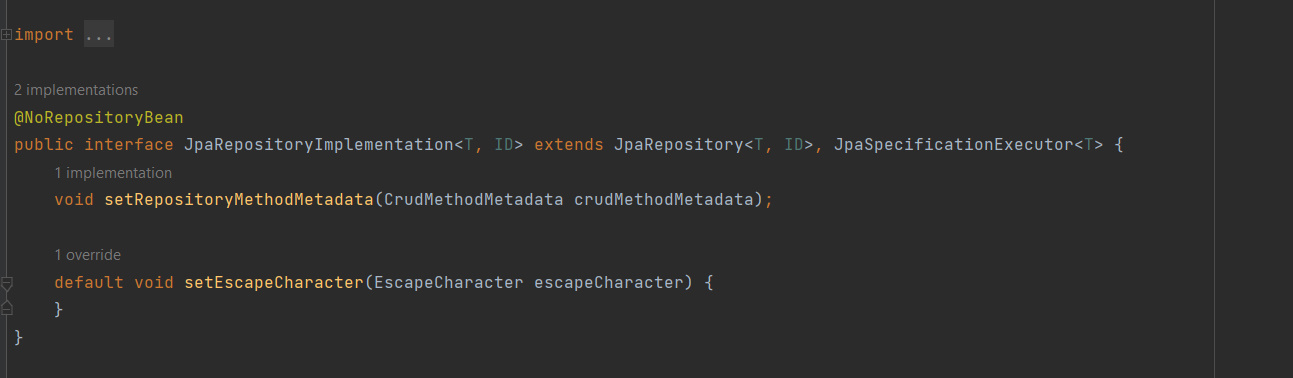
- Unele metode ce nu sunt doar readOnly folosesc @Transactional ca sa suprascrie cel pus la clasa cu readOnly = true







si JpaRepositoryImplementation



**Service Layer**

Services sunt cele care pastreaza Bussiness code sau logic