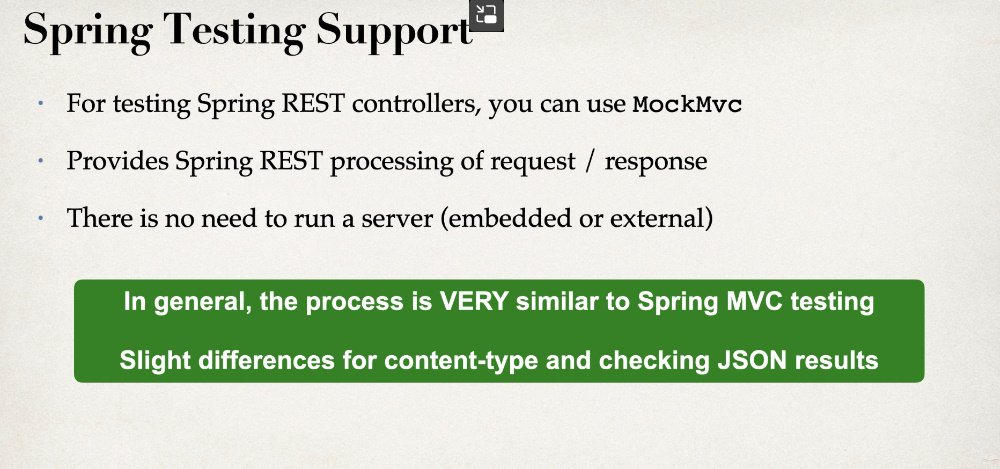
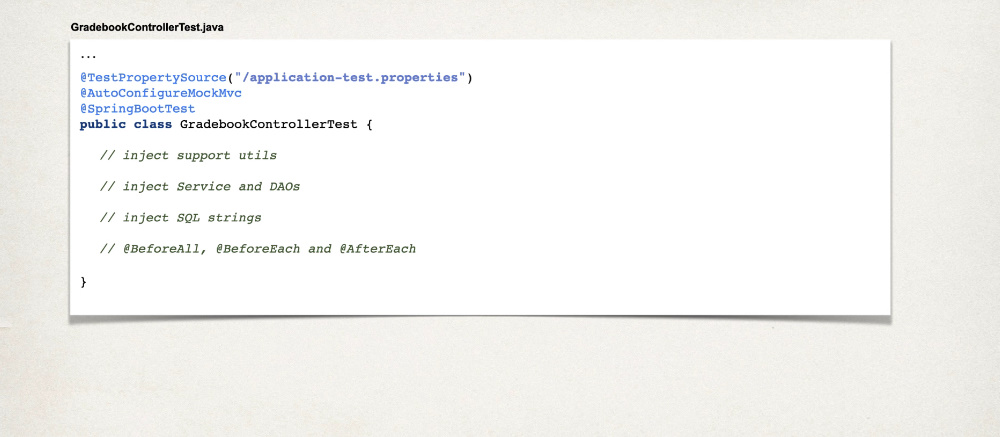
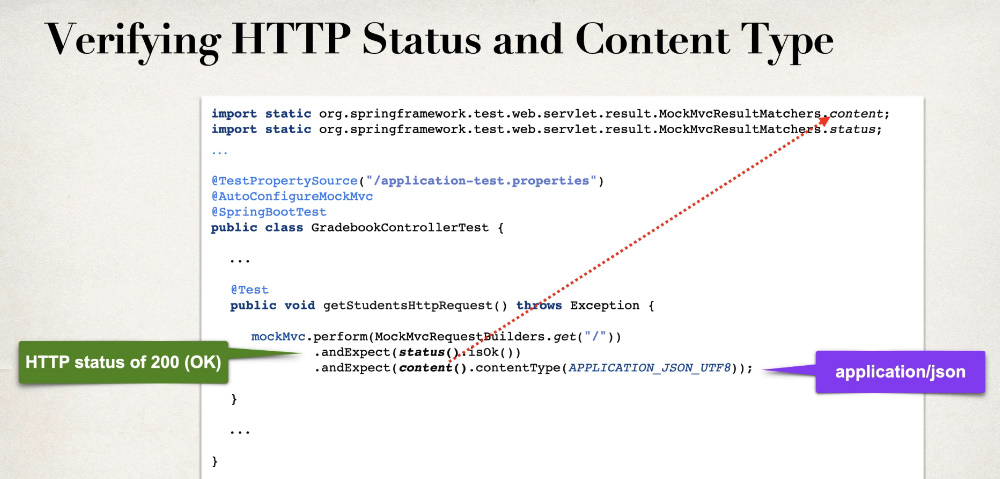
**Configurare**

* Putem testa un @RestController tot folosind **MockMvc**
* 
* 
* Uneori e posibil sa avem nevoie sa lucram direct cu baza de date chiar din test, de aceea punem anotatia @Transactional
* Pentru a obtine o referinta la un EntityManager, putem folosi si @Autowired, dar asta nu e o idee prea buna, mai bine folosim @PersistenceContext
* @PersistenceContext se va asigura sa existe un singur EntityManager per thread. Fiecare call la aplicatia noastra se face printr-un thread, si daca se fac mai multe calluri, vor fi deci mai multe threaduri, dar toate vor avea acelasi EntityManager, si asta poate fi rau de tot, caci toti userii vor accesa acelasi EntityManager. Doar entitymanaer e exact ca Session, una per application, dar anotatia asta va face sa nu mai fie doar una
* Pentru a testa un RestController, avem adesea nevoie de un ObjectMapper, si il putem obtine cu @Autowired. Jackson are grija sa creeze un obiect ObjectMapper

@Autowired  
ObjectMapper objectMapper;

**Verify HTTP Status and Content Type**



**MediaType APPLICATION\_JSON\_UTF8 = MediaType.APPLICATION\_JSON\_UTF8**

E depreciata! Folosim fara \_UTF8

**ATENTIE:**

mockMvc.perform(MockMvcRequestBuilders.*post*("/")  
 .contentType(MediaType.*APPLICATION\_JSON*)

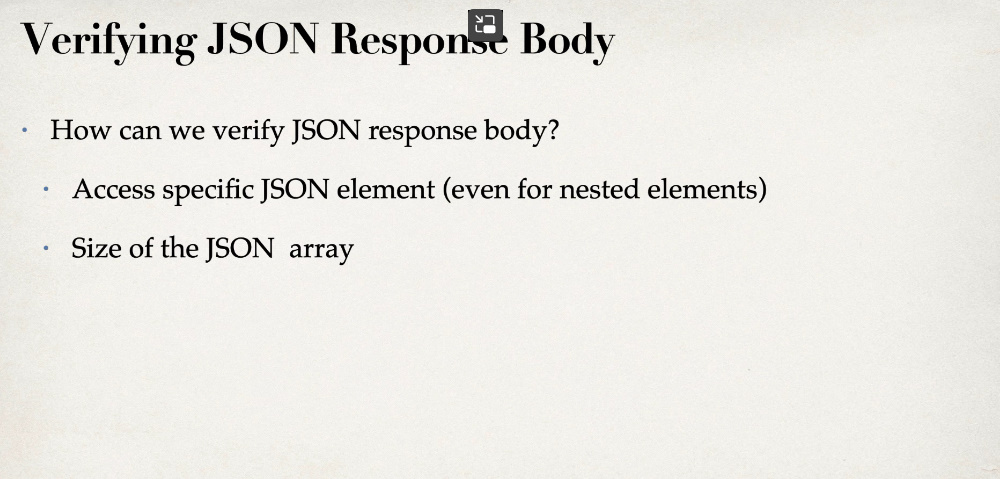
.content(objectMapper.writeValueAsString(collegeStudent)))

Aici .contentType() se asigura ca datele trimise de noi sa fie transformate in obiect, ca de ex cand e @ModelAttribute,@RequestBody . Cand .contentType() se afla in perform(), ea influenteaza cum datele ajung la controller, si anume ce date sunt specificate in .content()

mockMvc.perform(MockMvcRequestBuilders.*get*("/"))  
 .andExpect(*status*().isOk())  
 .andExpect(*content*().contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect(*jsonPath*("$",*hasSize*(2)));

aici, content().contentType() se va asigura ca datele returnate sa fie in format JSON. content() returneaza datele de la metoda post si le memoreaza, ca mai apoi sa putem folosi alt andExpect pentru a lucra cu ele. Deoarece content.cotentType() e in afara la perform, ea influenteaza datele care vin de la controller

**Verify Response Body**



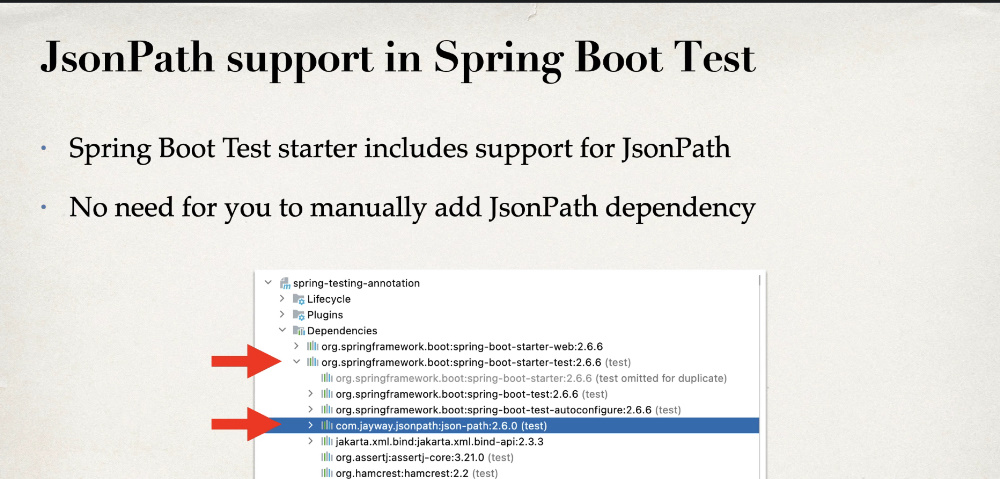
Sau folosind JSON Path

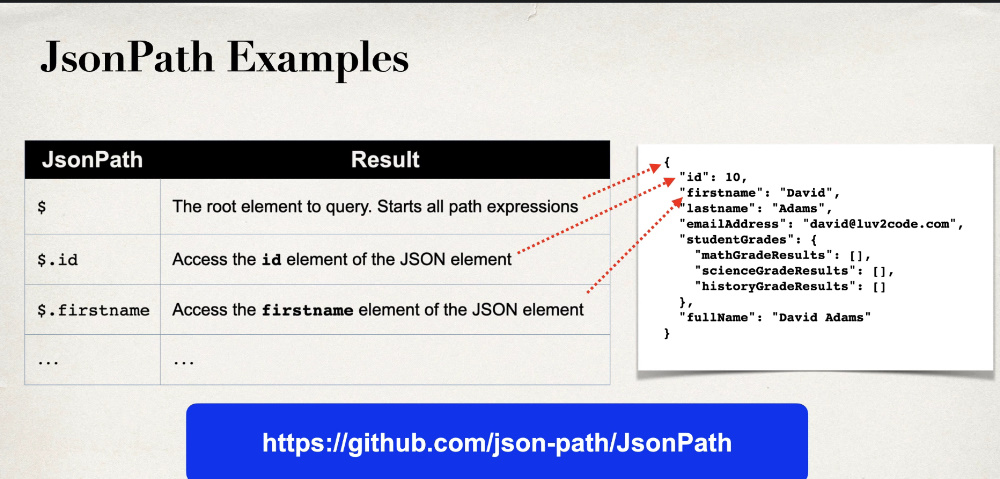
**JSON Path**



Anume JSON Path ne ajuta sa accesam elementele JSON si sa facem comparatii.

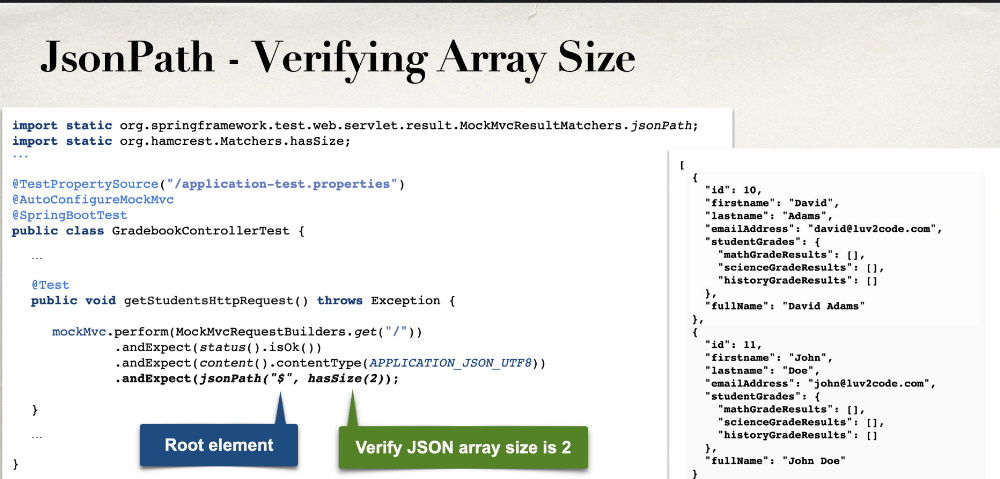
* Spring-Boot-Test include si JSON Path, deci nu mai trebuie sa cream o dependenta pentru el:



* 

$ e root element, adica obiectul propriu zis. El anume cuprinde ce e intre { }

$.id – elementul id din obiectul JSON

* 

jsonPath(“$”,hasSize(1)) – returneaza true daca dimensiunea la tot obiectul e 1, adica tot requestul ce scoate studentii din baza de date va returna un obiect cu un student in el, ceva asa [ {... }]

jsonPath(“$.id”,is(1)) – returneaza true daca id = 1

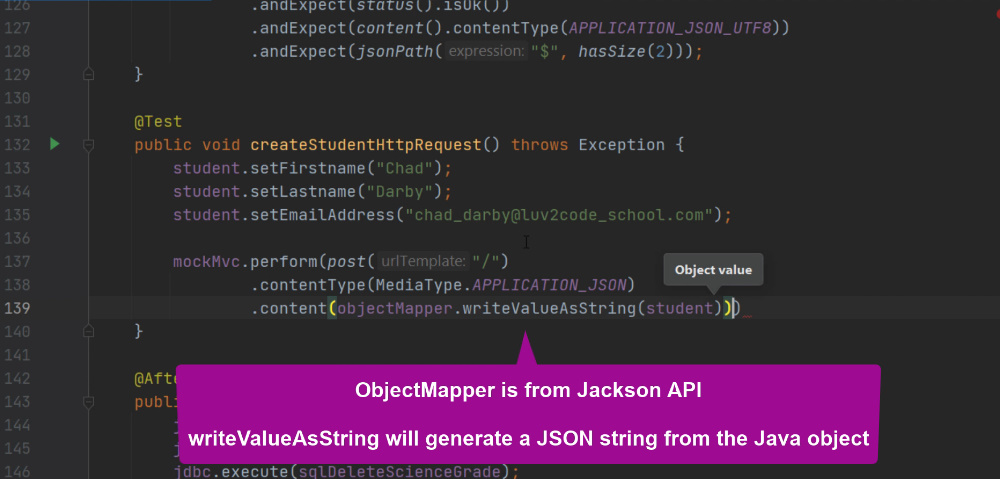
* andExpect() – expect e ca o conditie ce trebui sa fie indeplinita

@Test  
public void getStudentHttpRequest() throws Exception {  
 mockMvc.perform(MockMvcRequestBuilders.*get*("/"))  
 .andExpect(*status*().isOk())  
 .andExpect(*content*().contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect(*jsonPath*("$",*hasSize*(1)));  
}

* EntityManager e bun daca vrem sa adaugam studenti, sau obiecte in baza de date, si sa vedem apoi cum merg testele. Putem face asa si cu DAO, dar cu EntityManager e mai sigur si usor.
* @Test  
  public void getStudentHttpRequest() throws Exception {  
   collegeStudent.setFirstname("Mititiuc");  
   collegeStudent.setLastname("Eduard");  
   collegeStudent.setEmailAddress("edikutsu2002@mail.ru");  
   entityManager.persist(collegeStudent);  
   entityManager.flush();  
    
   mockMvc.perform(MockMvcRequestBuilders.*get*("/"))  
   .andExpect(*status*().isOk())  
   .andExpect(*content*().contentType(MediaType.*APPLICATION\_JSON*))  
   .andExpect(*jsonPath*("$",*hasSize*(2)));  
  }

Observatie. Obeictul mockMvc, la executarea metodei perform() si apoi andExpect() se returneaza un MvcRestul, dar aici nu avem nevoie de niciun MvcResult, asa oricum andExpect() va face ca testul sa cada daca nu e true.

**Post test**



objectMapper.writeValueAsString(student) – aceasta metoda transforma obiectul student in JSON, dar anume ca String. Deci pur si simplu ceea ce vedem in browser in JSON va fi String si gata. Nu putem folosi pur si simplu objectMapper.writeValue(object),deoarece metoda are nevoie si de un fisier de tip File, in care sa scrie datele in JSON, dar noi vrem ca el deodata sa ne returneze aceste date in JSON, fara a le scrie undeva, ca String.

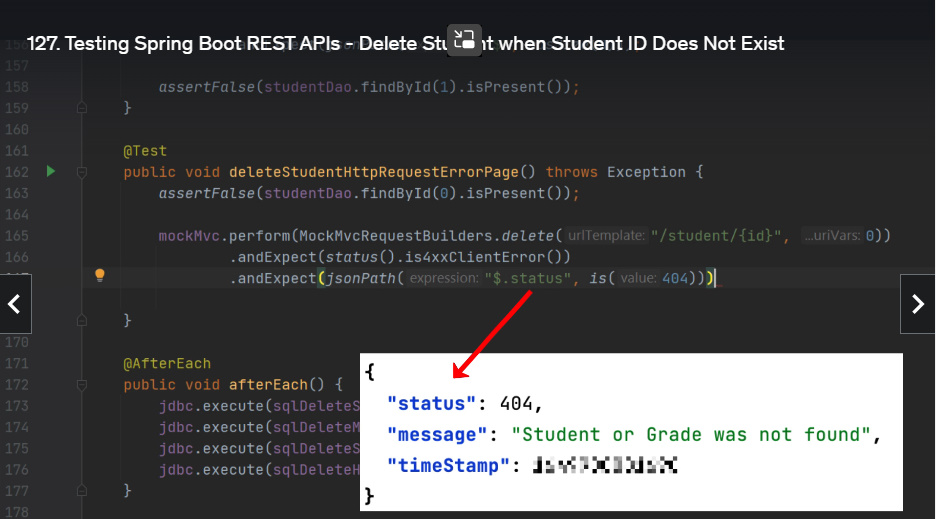
**Test wrong request**

* andExpect(status().isOk()) verifica daca statusul returnat de request e unul oke, adica 2\*\*, dar daca de ex dam un delete request pentru a sterge un id, si acel id e invalid, pai statusul nu mai e oke, e deja o eroare 400 si ceva, sau 4\*\*., creata de client.
* Pentru asta, folosim:

andExpect(status().is4xxClientError())

* Mai apoi, s-ar putea sa vedem daca obiectul JSON de eroare returnat e cel asteptat, si sa vedem ce id e setat in el sau ce mesaj:

andExcept(jsonPath(“$.field”,is(“valoare”)))



mockMvc.perform(MockMvcRequestBuilders.*delete*("/student/{id}",0))  
 .andExpect(*status*().is4xxClientError())  
 .andExpect(*jsonPath*("$.status",*is*(404)))  
 .andExpect(*jsonPath*("$.message",*is*("Student or Grade was not found")));