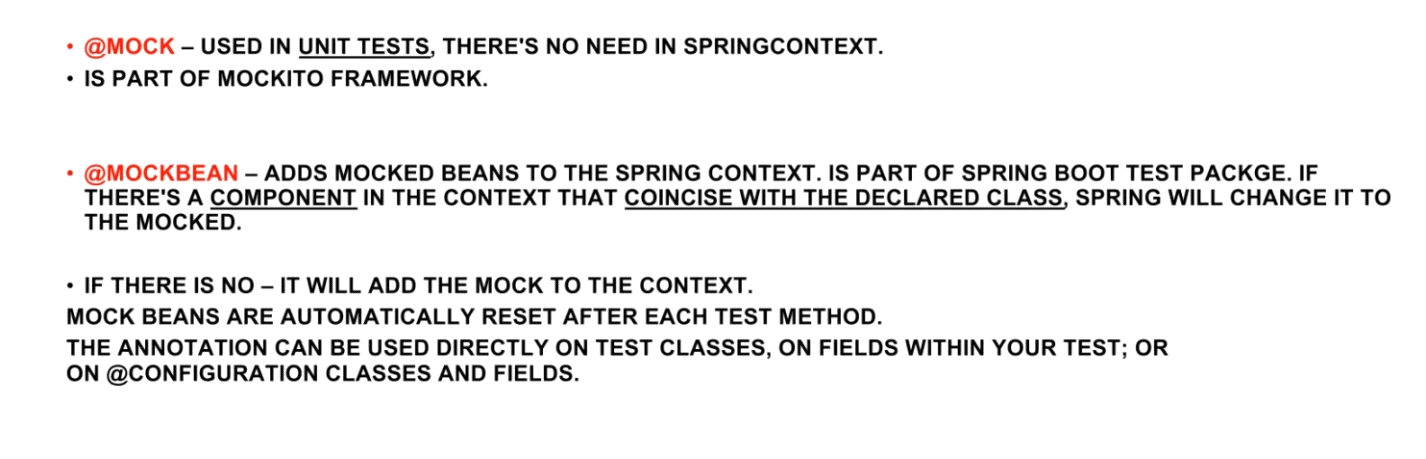
**Integration test**

* Testarea mai multor componente impreuna, fara dubluri.
* 
* Cand testam, nu e o idee buna sa punem anotatia @ComponentScan(), asa cum ea va scana si va crea beanuri de care la sigur nu vom avea nevoie. Cel mai bine e sa folosim @Import
* @Import – aici specificam anume concret ce Configuration sau Component vrem sa fie creat, nu toate dintr-un package!

De ex, mai cream un context in package unde e clasa de test. Cream o clasa Student in ea, care nu e component:

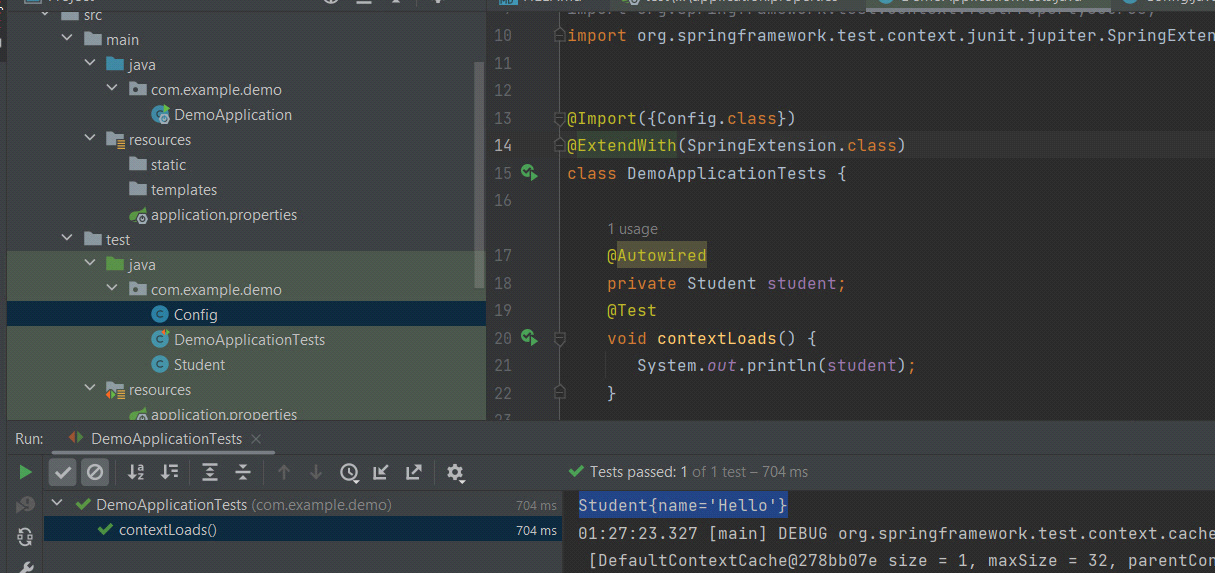
public class Student {  
 private String name = "Hello";  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "name='" + name + '\'' +  
 '}';  
 }  
}

Si cream un config:

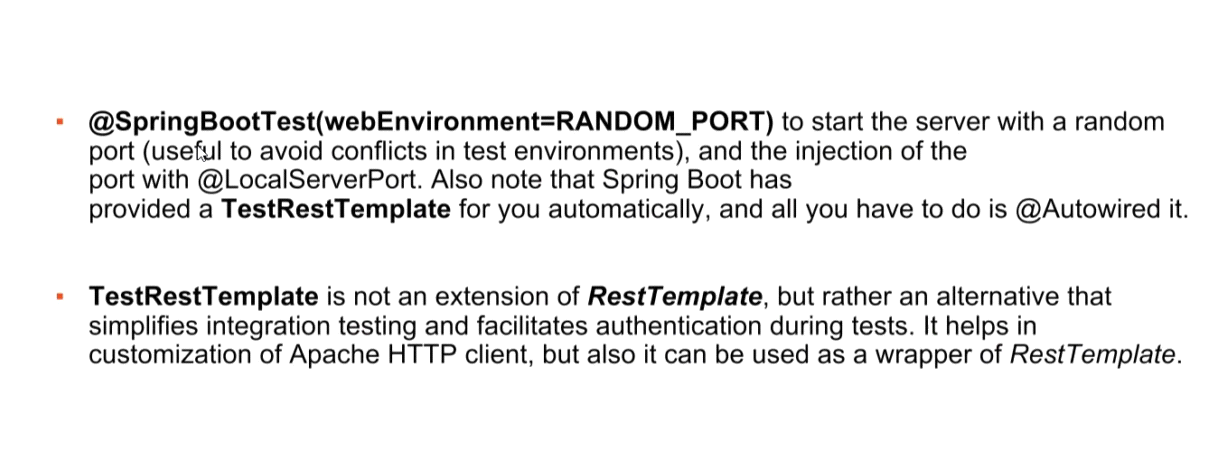
@Configuration  
public class Config {  
 @Bean  
 public Student student(){  
 return new Student();  
 }  
}

Si nu folosim @SpringBootTest, ca sa nu se porneasca context de la spring boot, ce are @ComponentScan automat pus.

@Import({Config.class})  
@ExtendWith(SpringExtension.class)  
class DemoApplicationTests {  
  
 @Autowired  
 private Student student;  
 @Test  
 void contextLoads() {  
 System.*out*.println(student);  
 }  
  
}



* @ExtendWith(SpringExtension.class) – adauga toate chestiile necesare de la Spring pentru a testa, si anume iata de ce testul va porni ca un spring app.
* @EnableJpaRepositories – in testare ea e folosita. Scaneaza toate repozitoriile JPA si creaza beanuri. E necesara daca nu folosim @SpringBootTest sau @SpringBootApplication
* @EntityScan – scaneaza doar claselee @Entity. E necesara pentru JpaRepository
* @Configuration  
  @EntityScan(basePackages = {"com.example.demo.entity"})  
  @EnableJpaRepositories(basePackages = "com.example.demo.entity")  
  public class Config {  
  }
* @TestPropertySource – properties pentru testare. Ea va suprascrie fisierul application.properties din spring boot aplication
* @Sql(file) – executa un script sql pentru test
* @SpringBootTest – ne scapa de a pune cam cea mai mare parte din anotatiile de sus



* @TestConfiguration – sunt configuratii ce nu sunt detectate de @ComponentScan si deci nici de @SpringBootApplication, ci pot fi folosite doar cu @Import
* @TestComponent – component ce nu pot fi depistate cu component scan, ci doar cu @Import

Atentie! @TestConfiguration odata importata, daca se intampla ca are beanuri identice cu cele din context de la Spring Boot, el le va rescrie!

Fie in spring boot package clasa Student:

@Component  
public class Student {  
  
 @Override  
 public String toString() {  
 return "Acest student e din application package";  
 }  
}

Si acum fie in test package,intr-un packase separat, tot clasa student:

@TestComponent  
public class Student {  
 public String toString(){  
 return "Acest student e din test package";  
 }  
}

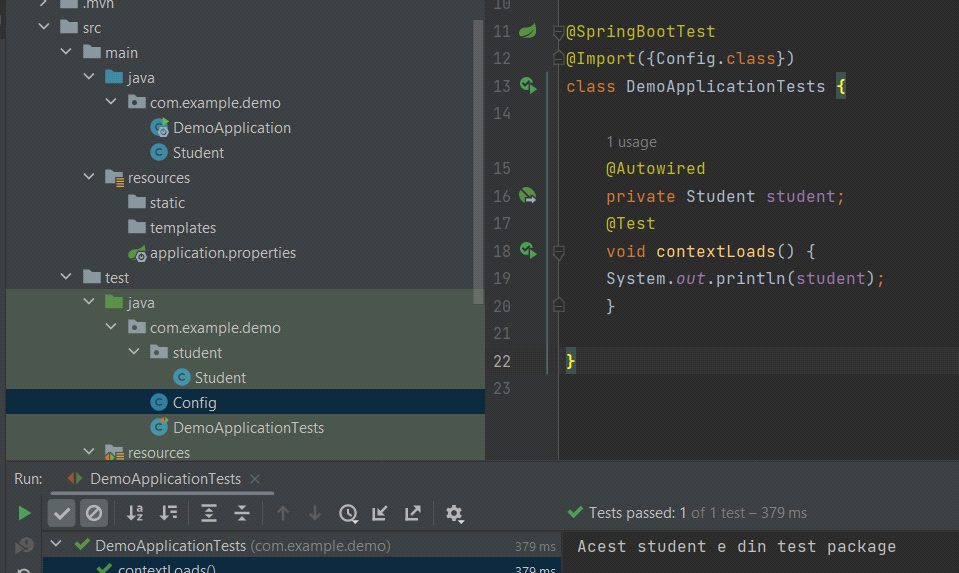
Si fie in test package configul:

@TestConfiguration  
@Import({Student.class})  
public class Config {  
  
}

Atentie! @TestConfiguration , chiar de punem @ComponentScan, nu va scana @TestComponent!

@SpringBootTest  
@Import({Config.class})  
class DemoApplicationTests {  
  
 @Autowired  
 private Student student;  
 @Test  
 void contextLoads() {  
 System.*out*.println(student);  
 }  
  
}

Acum, vom avea student din test package, caci @TestConfiguration a rescris beanul din spring boot context



* @ContextHierarchy(

[{@ContextConfiguration(classes={.....})](mailto:%7b@ContextConfiguration(classes=%7b.....%7d)),

[@ContextConfiguration(classes={.....})](mailto:%7b@ContextConfiguration(classes=%7b.....%7d)),

[@ContextConfiguration(classes={.....})](mailto:%7b@ContextConfiguration(classes=%7b.....%7d))}

)

Aceasta anotatie creaza din mai multe configuratii una singura.Ordinea in care le punem conteaza. Prima e parintele la toti, a 2 e copilul la prima, a 3 e copilul la a 2 si tot asa.

Acum, in rezultat se va crea un Context care le contine pe toate 3. Daca un context copil are exact acelasi bean ca un context parinte, el il va suprascrie si gata.

Fie 3 configuratii:

@Configuration  
public class Config1 {  
 @Bean  
 public String bean1(){  
 return "Config1";  
 }  
 @Bean  
 public String bean(){  
 return "Config1";  
 }  
}

@Configuration  
public class Config2 {  
 @Bean  
 public String bean2(){  
 return "Config2";  
 }  
 @Bean  
 public String bean(){  
 return "Config2";  
 }  
}

@Configuration  
public class Config3 {  
 @Bean  
 public String bean3(){  
 return "Config3";  
 }  
 @Bean  
 public String bean(){  
 return "Config3";  
 }  
}

Si clasa de test:

@ExtendWith(SpringExtension.class)  
@ContextHierarchy({  
 @ContextConfiguration(classes = {Config1.class}),  
 @ContextConfiguration(classes = {Config2.class}),  
 @ContextConfiguration(classes = {Config3.class})}  
)  
class DemoApplicationTests {  
  
 @Autowired  
 private ApplicationContext context;  
 @Autowired  
 @Qualifier("bean1")  
 private String bean1;  
  
 @Autowired  
 @Qualifier("bean2")  
 private String bean2;  
 @Autowired  
 @Qualifier("bean3")  
 private String bean3;  
  
 @Autowired  
 @Qualifier("bean")  
 private String bean;  
 @Test  
 void contextLoads() {  
 System.*out*.println(bean1);  
 System.*out*.println(bean2);  
 System.*out*.println(bean3);  
 System.*out*.println(bean);  
 }  
  
}

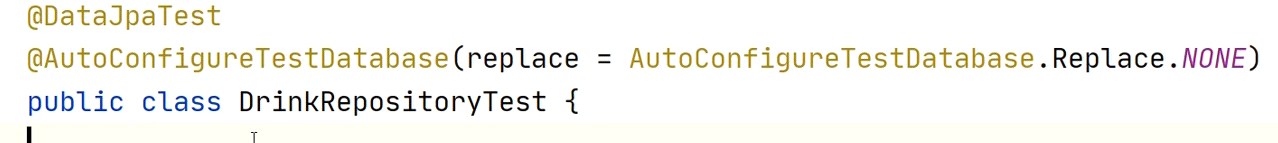


Putem da si @Autowired la un ApplicationContext, caci va fi unul creat din toate cele 3

**Atentie! Un child poate avea doar un parinte! Dar un parinte oricati children**

* @DataJpaTest – anotatia data se focuseaza doar pe a testa JPA componente. Ea va dezactiva configuratia totala la application context si va crea unul ce are doar ceea ce e necesar pentru a testa componente JPA. In mod default, ea va scana doar @Entity si JpaRepository si @Repository. Va incerca sa creeze beanuri pentru a se conecta la o In Memory Data Base

Daca totusi nu vrem ca el sa creeze beanuri pentru o In Memory DB, ci sa o ia pe cea din aplicatia Spring, folosim @AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)



Fiecare Test va avea o transactie proprie. Totodata, ea creaza si un TestEntityManager si EntityManager si JdbTemplate.

@ExtendWith(SpringExtension.class)  
@PropertySource("/application.properties")  
@DataJpaTest  
@Sql("/script.sql")  
  
class DemoApplicationTests {  
  
 @Autowired  
 TestEntityManager entityManager;  
 @Autowired  
 TeacherDAO teacherDAO;  
 @Autowired  
 JdbcTemplate jdbcTemplate;  
 @Autowired  
 EntityManager entityManager2;  
  
  
 @Test  
 void test1() {  
 teacherDAO.save(new Teacher(80,"Test","Test",5));  
 Assertions.*assertEquals*("Mititiuc",entityManager.find(Teacher.class,10).getFirstName());  
 Assertions.*assertDoesNotThrow*(() -> teacherDAO.findByFirstName("Test"));  
 }  
 @Test  
 public void test2(){  
 System.*out*.println(teacherDAO.findByFirstName("Test"));  
 Assertions.*assertNull*(() -> teacherDAO.findByFirstName("Test"));;  
 }  
  
}

Anotatia @Sql poate fi pusa si la nivel de clasa, asa fiecare test va executa acel query**. Dupa fiecare test, baza de date se curata automat!!!.**

* TestEntityManager tot foloseste un EntityManager
* Anotatia

@AutoConfigureMockMvc

Configureaza un MockMvc si poate fi folosita doar cu @SpringBootTest

* Totusi, aici trebuie iar sa avem toate beanurile din aplicatie pentru a testa rest controllerul, dar poate nu avem nevoie de toate
* Anotatia

@WebMvcTest

tot creaza un MockMvc, dar nu necesita @SringBootTest, asa cum ea va scana doar @Controller si @ControllerAdvice, dar daca controllerul are dependente, trebuie oferite cu MockBean

@WebMvcTest  
@TestPropertySource("/application.properties")  
  
class DemoApplicationTests {  
 @MockBean  
 TeacherDAO teacherDAO;  
  
 @Autowired  
 MockMvc mockMvc;  
 @Test  
 public void test() throws Exception {  
 *when*(teacherDAO.findById(10)).thenReturn(Optional.*of*(new Teacher(10,"Test","Test",5)));  
  
 mockMvc.perform(MockMvcRequestBuilders.*get*("/api/10"))  
 .andExpect(*jsonPath*("$.id",*is*(10)));  
 }  
  
}

* MockMvcResultMatchers

El are metodele status(), isOk() etc.

**TestRestTemplate**

* Pentru a-l putea folosi, e necesar sa facem asta:
* @SpringBootTest(webEnvironment= SpringBootTest.WebEnvironment.*RANDOM\_PORT*)

**SpringBootTest** nu porneste in mod default un embeded server, de aceea requesturile catre controller ar esua. Acest parametru va porni un server embeded cu un port random.

**Asa, se va generea un port random si un bean TestRestTemplate**

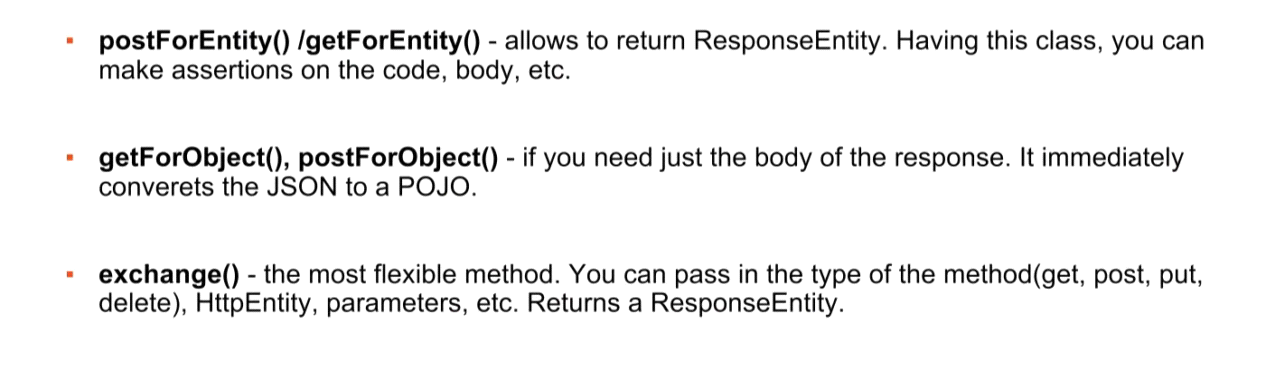
@Autowired  
TestRestTemplate testRestTemplate;  
@LocalServerPort  
int randomServerPort;

Primim obiectul TestRestTemplate si portul.

* getForObject(“url”,Clasa.class) – returneaza obiectul de la acest url
* getForEntity(“url”,Clasa.class) – returneaza un ResponseEntity
* postForEntity(“url”,object,Clasa.class)
* postForObject(“url”,object,Clasa.class)

@SpringBootTest(webEnvironment= SpringBootTest.WebEnvironment.*RANDOM\_PORT*)  
@TestPropertySource("/application.properties")  
class DemoApplicationTests {  
  
 @Autowired  
 private TeacherDAO teacherDAO;  
 @Autowired  
 TestRestTemplate testRestTemplate;  
 @Autowired  
 ObjectMapper objectMapper;  
 @LocalServerPort  
 int randomServerPort;  
  
  
 @Sql("/script.sql")  
 @Test  
 public void get() {  
 Teacher teacher = testRestTemplate.  
 getForObject("http://localhost:"+randomServerPort+"/api/10", Teacher.class);  
 System.*out*.println(teacher);  
 }

@Test  
 public void post() {  
 Teacher teacher1 = new Teacher(100,"Unexisted","Teacher",50);  
 Teacher teacher2 = testRestTemplate.postForObject("http://localhost:"+randomServerPort+"/api",teacher1,Teacher.class);  
 System.*out*.println(teacher2);  
  
 }  
  
}



**TestRestTemplate nu necesita url complet, gen putem pune doar "/api/users", dar RestTemplate necesita full, ca "**[**www.mySite.com/api/users**](http://www.mysite.com/api/users)**"**

**JSONAssert**

* Are doar metodele AssertEquals si AssertNotEquals
* Metodele au urmatorul prototipuri:

**(Mesaj,StringJsonexpected,StringJsonGiven,JSONCompareMode.TipulAles)**

JSONCompareMode.LENIENT – chiar daca JSON oferit contine mai multe fielduri ca cel expected, dar le contine pe toate ale lui, testul oricum trece

JSONCompareMode.STRICT – JSON oferit trebuei sa contina strict ce are cel expected

**(StringJsonexpected,StringJsonGiven,JSONCompareMode.TipulAles)**

**(StringJsonexpected,StringJsonGiven,boolean)**

true – strict

false - LENIENT

* Ordinea in care sunt puse fieldurile nu conteaza
* @Test  
  public void testJSON() throws JSONException {  
   String json1 = "{name:\"Eduard\",age:19}";  
   JSONAssert.*assertEquals*("Test","{age:19,name:\"Eduard\"}",json1, JSONCompareMode.*STRICT*);  
  }
* Putem compara si array, dar atentie, indiferent ca punem LENIENT sau STRICT, itemele din array trebuie sa fie identice.
* Pentru a compara size in array:
* @Test  
  public void testJSON() throws JSONException {  
   String json1 = "{name:[1,2,3,4,5,6,7,8]}";  
   JSONAssert.*assertEquals*("{name:[2,8]}",json1, new ArraySizeComparator(JSONCompareMode.*STRICT*));  
  }

name:[2,8] – numarul de elemente din array trebuie sa fie intre 2 si 6, inclusiv 2 si 6

**Profiles**

* Profiles ne permit sa alegem cand anume anumite beanuri sa fie create
* Profiles ne permit sa activam diferite beanuri in diferite situatii in diferite perioade
* Un profile poate controla 2 lucruri:
* beanurile ce sa fie incarcate
* properties
* Spring creaza un default Profile. Asa dar, orice bean care nu are specificat un profile concret vor avea ca si cum automat @Profile("default”). Spring creaza automar un profile numit “default” si toate beanurile fara profile specificat apartin lui
* Putem modifica numele la default profile:

spring.profiles.default=altNume

* Putem pune @Profile(“nume”) deasupra la

@Bean, @Configuration, @Repository, @Service etc.

* Putem folosi si ! pentru a arata ca nu apartine unui profile:

@Profile("!prod")

@Configuration

**public** **class** NonProductionProfileConfiguration {

@Bean

**public** DataSource h2DataSource() {*/\*...\*/*}

}

* Putem crea si fisire properties specifice diferitor profile. Tot ce trebuie sa facem e sa cream un fisier .properties sub forma:

Fisier-{NumeProfil}.properties:

* *application.properties* – contains properties applicable to all environments
* *application-local.properties* – configure the application when *local* profile is activated
* *application-dev.properties* – configure the application when *dev* profile is activated
* *application-prod.properties* – configure the application when *prod* profile is activated
* Putem crea configuratii in fisierul application.properties folosind delimitatorul **spring.config.activate.on-profile**=Nume

**application.name**=My Application

*#--- local profile ---#*

**spring.config.activate.on-profile**=local

**spring.datasource.driver-class-name**=org.h2.Driver

**spring.datasource.url**=jdbc:h2:mem:db;DB\_CLOSE\_DELAY=-1

**spring.datasource.username**=sa

**spring.datasource.password**=

*#--- dev profile ---#*

**spring.config.activate.on-profile**=dev

**spring.datasource.driver-class-name**=com.mysql.cj.jdbc.Driver

**spring.datasource.url**=jdbc:mysql://localhost:3306/db

**spring.datasource.username**=root

**spring.datasource.password**=password

Dar atentie:

# Valid usecase

spring.profiles.active=prod

# Invalid usecase

spring.config.activate.on-profile=prod

spring.profiles.active=dev

* Putem crea si grupuri de profiles:

**spring.profiles.group.prod**=prodJms,prodDatabase,prodJndi

With above profile group definition, activating the *prod* profile will activate *prodJms*, *prodDatabase* and *prodJndi* as well.

* Pentru a activa un profile, sau mai multe:

**spring.profiles.active**=dev

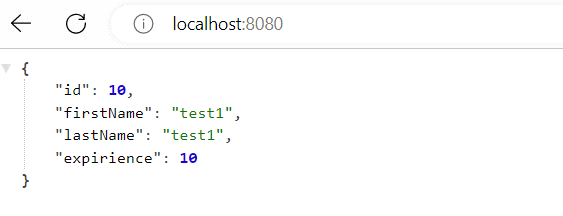
Exemplu:

@SpringBootApplication  
public class DemoApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(DemoApplication.class, args);  
 }  
  
 @Profile("test1")  
 @Bean  
 public Teacher teacher1(){  
 return new Teacher(10,"test1","test1",10);  
 }  
 @Profile("test2")  
 @Bean  
 public Teacher teacher2(){  
 return new Teacher(20,"test2","test2",20);  
 }  
}

@RestController  
public class RestControllerApp {  
 @Autowired  
 private Teacher teacher;  
  
 @GetMapping("/")  
 public Teacher index(){  
 return teacher;  
 }

}

spring.profiles.active=test1



**default profile este mereu activat automat, indiferent de ce profile mai activam. Doar ca daca in applicatiom.properties activam un profile, orium se iau in calcul proprietatile din application.properties, doar ca cele din .properties la acel profile le vor rescrie pe cele din application.properties, exact asa cum functioneaza profiles in maven**

De ex, fie application.properties

spring.datasource.url=jdbc:mysql://localhost:3306/test  
spring.datasource.username=testuser  
spring.datasource.password=Frb2eshox!  
  
test.value = Hello  
  
spring.profiles.active=test

si application-test.properties

test.value = test profile

Aplicatia va rula, se va conecta la baza de date specificata in application.properties, dar test.value va fi suprascris de application-test.properties

**ArgumentCaptor**

* Este o alternatica pentru Mockito.verify(), dar care ne permite sa
* Fie il cream cu new, fie folosim @Captor
* Argument captor este folosit pentru a obtine parametrul ce a fost trimis unei metode, si a face difertie teste cu el. De ex:

@Test  
public void test(){  
 Teacher teacher = new Teacher();  
 teacher.setFirstName("Test");  
 course.setTeacher(teacher);  
 System.*out*.println(teacher);  
 Mockito.*verify*(course).setTeacher(teacherArgumentCaptor.capture());  
 System.*out*.println(teacherArgumentCaptor.getValue());  
  
  
 Assertions.*assertEquals*("Test",teacherArgumentCaptor.getValue().getFirstName());

Assertions.*assertTrue*(teacher==teacherArgumentCaptor.getValue());

}

Cand am folosit teacherArgument.capture(), in teacherArgumentCaptor s-a salvat parametrul care a fost trimis metodei setTeacher() anterior, si deci putem apoi sa il obtinem ca sa facem asserturi.

* Acum, vedem ca teacher si cel din captor sunt obiecte identice
* Chiar si referinele lor sunt identice, deci captor a obtinut acel parametru deja utilizat si l-a salvat.
* Totodata, verify() s-a asigurat ca s-a capturat un argument, adica a fost executata metoda setTeacher() odata

**ArgumentCaptor multiple parameters**

* **Putem avea si mai multe argumente captate:**

@Test  
public void test(){  
 List<Teacher> list = List.*of*(new Teacher(),new Teacher());  
 list.get(0).setFirstName("Teacher1");  
 list.get(1).setFirstName("Teacher2");  
  
 course.setTeacher(list.get(0));  
 course.setTeacher(list.get(1));  
  
 Mockito.*verify*(course, *times*(2)).setTeacher(teacherArgumentCaptor.capture());  
  
  
  
 List<Teacher> testList = teacherArgumentCaptor.getAllValues();  
  
 Assertions.*assertEquals*(list.get(0),testList.get(0));  
 Assertions.*assertEquals*(list.get(1),testList.get(1));  
  
  
}

Anume times(2) verifica daca sunt 2 argumente captate. Daca nu punem times(2), ca sa verifice daca e unul captat, dar sunt 2, tot primim exceptie.

Fara argumentCaptor, ar fi trebui sa verificam fiecare argument cu metoda data

@Test  
public void test(){  
 List<Teacher> list = List.*of*(new Teacher(),new Teacher());  
 list.get(0).setFirstName("Teacher1");  
 list.get(1).setFirstName("Teacher2");  
  
 course.setTeacher(list.get(0));  
 course.setTeacher(list.get(1));  
  
 Mockito.*verify*(course).setTeacher(list.get(0));  
 Mockito.*verify*(course).setTeacher(list.get(1));  
   
}

**Dar asa, folosim un singur verify() si ArgumentCaptor deodata va primi argumentele si le va si numara**

**Exemplu mai practic**

In exemplele de mai sus, importanta lui ArgumentCaptor nu este foarte bine evidentiata. Dar, iata un exemplu care arata cum este important.

Fie clasa Email:

@Component  
public class Email {  
 private int id;  
 public String emailAddress;  
  
 public Email(int id, String emailAddress) {  
 this.id = id;  
 this.emailAddress = emailAddress;  
 }  
  
 @Override  
 public String toString() {  
 return "Email{" +  
 "id=" + id +  
 ", emailAddress='" + emailAddress + '\'' +  
 '}';  
 }  
}

**Fie clasa Deliver, ce trimite un email ca si cum:**

@Component  
public class Deliver {  
 public Email sendEmail(Email email){  
 return email;  
 }  
}

Fie serviciul MyService ce creaza si trimite email:

@Service  
public class MyService {  
 @Autowired  
 private Deliver deliver;  
 private Random random = new Random()

;  
  
 public Email createAndSendEmail(String domain){  
 Email email;  
 if(domain.equals(".ru"))  
 email = new Email(random.nextInt(),"eduard@mail.ru");  
 else  
 email = new Email(random.nextInt(),"eduard@yahoo.com");  
 return deliver.sendEmail(email);  
 }  
}

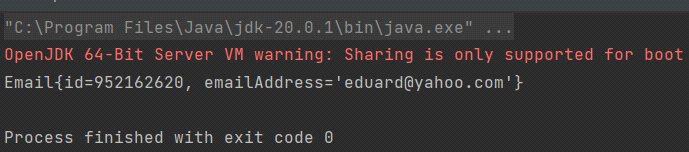
* **Acum,** Deliver va fi un Mock cand vom face teste, deci metoda lui sendEmail() va returna null;
* Acum vrem cumva sa vedem daca email trimis cu sendEmail() a fost scris corect, dar nu putem, caci email se creaza in interiorul metodei createAndSendEmail() si deci, nu putem nici folosi when(deliver.sendEmail(...)).thenReturn(...) caci trebuie deja sa oferim un obiect Email, dar nu putem sti care va fi id nicidecum
* Nici nu putem folosi verify(deliver).sendEmail(...) caci iar nu stim ce email va fi trimis si nu avem referinta sau obiect creat de noi, pur si simplu nu il putem trimite.
* Anume aici ne va ajuta ArgumentCaptor. El ne permite sa capturam obiectele care au fost parametri in metodele lui Deliver

@ExtendWith(MockitoExtension.class)  
class DemoApplicationTests {  
 @Mock  
 Deliver deliver;  
 @InjectMocks  
 MyService myService;  
 @Captor  
 ArgumentCaptor<Email> argumentCaptor;  
  
 @Test  
 public void test(){  
 myService.createAndSendEmail("en");  
 Mockito.*verify*(deliver).sendEmail(???)  
 }  
  
}

**Deci, nu putem sti ce email a fost pus in sendEmail()**

**In plus, acel return va returna null, caci deliver e un mock si metodele sale nu fac nimic**

@Test  
public void test(){  
 myService.createAndSendEmail("en");  
 Mockito.*verify*(deliver).sendEmail(argumentCaptor.capture());  
 System.*out*.println(argumentCaptor.getValue());  
}

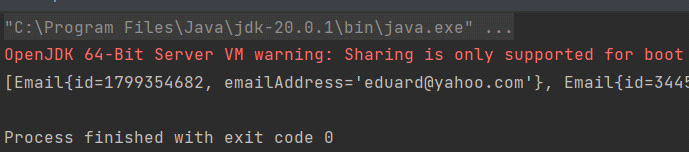


Vedem ca am primit parametrul ce a trecut prin metoda sendEmail a lui deliver

Cand am scris argumentCaptor().capture(), s-au capturat in ArgumentCaptor parametrii trecuti prin metoda sendEmail()

@Test  
public void test(){  
 myService.createAndSendEmail("en");  
 myService.createAndSendEmail("ru");  
  
 Mockito.*verify*(deliver,Mockito.*times*(2)).sendEmail(argumentCaptor.capture());  
 System.*out*.println(argumentCaptor.getAllValues());  
}

Aici vedem ca au fost capturati toti parametri trecuti prin metoda obiectului:



**AssertJ**

* Orice test incepe cu assertThat(Object) si e urmat de . metoda
* **isEqualTo(Object)** - compara 2 obiecte folosind metodele equals() ale lor
* **isEqualToComparingFieldByFieldRecursively(obj)** - compara nu dupa equals() la obiectele de comparat, dar dupa equals() la fiecare field ale lor
* isTrue/False()
* isEmpty()
* contains(obj) - pentru iterable
* isNotEmpty() - pentru liste etc.
* startsWith(obj) - testeaza daca primul element al unei liste e obj
* doesNotContainNull() - testeaza daca nu e niciun null in lista sau etc.
* isInterface() - verifica daca e interfata

assertThat(Runnable.class).isInterface();

**Exception Assert**

* assertThatThrownBy(lambda Expression).isInstanceOf:

*assertThatThrownBy*(() -> validationService.validatePaymentId(null)).isInstanceOf(IllegalArgumentException.class);