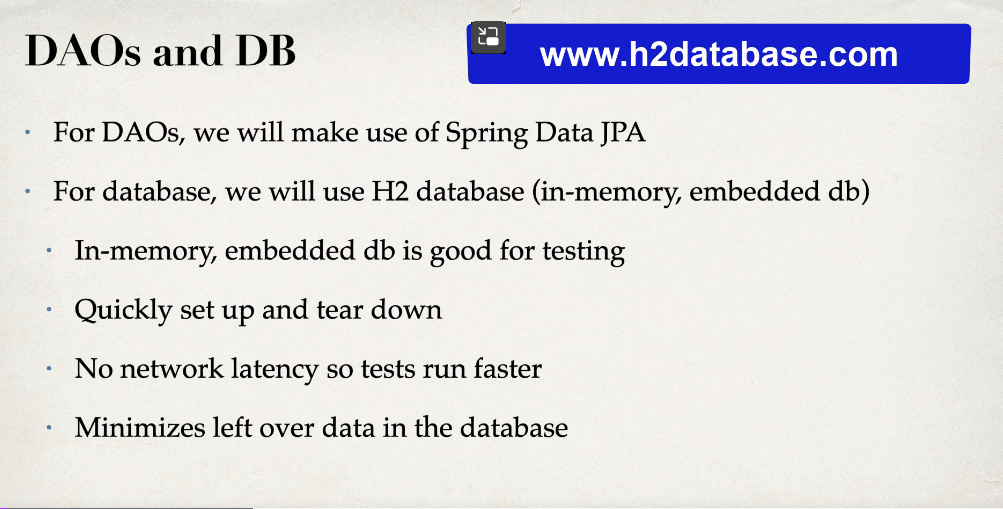
* @PathVariable poate fi folosit si in @Controller:
* @GetMapping("/studentInformation/{id}")  
   public String studentInformation(@PathVariable int id, Model m) {  
   return "studentInformation";  
   }

**H2 DataBase**

* H2 DataBase – In memory database.
* E buna pentru testare
* 
* Pentru a-i spune lui Spring ca vrem sa folosim o embeded database, trebuie dependenta:

<dependency>  
 <groupId>com.h2database</groupId>  
 <artifactId>h2</artifactId>  
 <scope>runtime</scope>  
</dependency>

* Odata ce spring va vedea ca avem o dependenta cu o H2 database, el va configura automat o DataSource pentru a ne conecta la aceasta baza de date din memorie
* Baza de date va fi creata pe baza la @Tabel si @Entity create de noi
* Asa proiectul nostru va avea o H2 dependecy, si Spring va crea automat baza de date data.

Iata de ce, codul de jos va functiona:

@SpringBootTest(classes = MvcTestingExampleApplication.class)   
public class StudentAndGradeServiceTest {  
  
 @Autowired  
 StudentAndGradeService studentService;  
 @Autowired  
 StudentDao studentDao;  
 @Test  
 public void createStudentService(){  
 studentService.createStudent("Mititiuc","Eduard","edikutsu2002@mail.ru");  
 CollegeStudent student = studentDao.findByEmailAddress("edikutsu2002@mail.ru");  
  
 *assertEquals*("edikutsu2002@mail.ru",student.getEmailAddress(),"Find by email");  
  
 }  
}

* Deci, desi nu avem nicio conexiune creata pentru o baza de date propriu zisa, Spring s-a asigurat sa creeze una automat pentru H2 database din temporar memory, si deci in StudentDao, cand vom da @AutoWired la EntityManager, Spring va face injectia cu acel DataSource creat automat pentru ea.

@Repository  
public class StudentDao{  
 @Autowired  
 private EntityManager entityManager;  
  
 public void save(CollegeStudent collegeStudent){  
 entityManager.persist(collegeStudent);  
 }  
public CollegeStudent findByEmailAddress(String emailAddress){  
 Query query = entityManager.createQuery("FROM CollegeStudent WHERE emailAddress = ?1").setParameter(1,emailAddress);  
 return (CollegeStudent) query.getSingleResult();  
}  
}



Putem folosi si Spring Data

public interface StudentDao extends JpaRepository<CollegeStudent, Integer> {  
CollegeStudent findByEmailAddress(String emailAddress);  
}

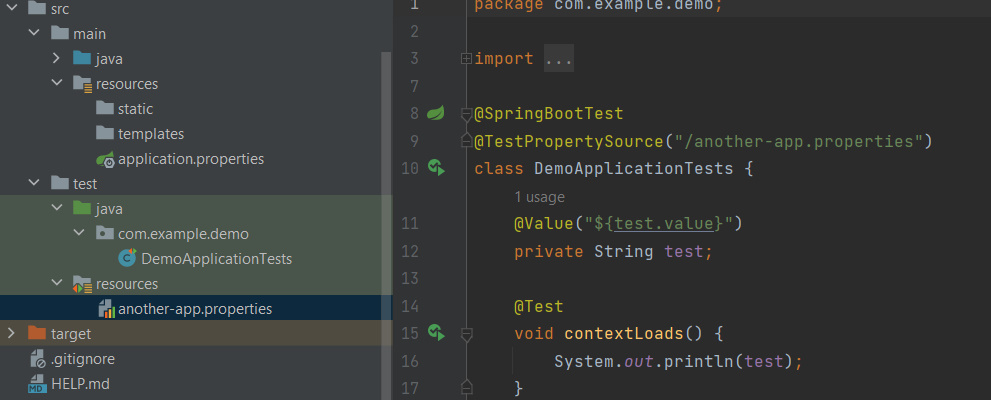
**@TestPropertySource**

* Anotatia data se pune la clasa de test, si in ea punem numele vreunui .proeprties file, pe care vrem sa il folosim in test

@SpringBootTest(classes = MvcTestingExampleApplication.class)  
@TestPropertySource("/another-application.properties")  
public class StudentAndGradeServiceTest {  
  
}

* Anotatia data permite sa adaugam un .properties file ce sa suprascrie proprietatile din application.properties.

De ex fie asa structura:





si application.properties si another-app.properties au o property test.value

**application.properties**

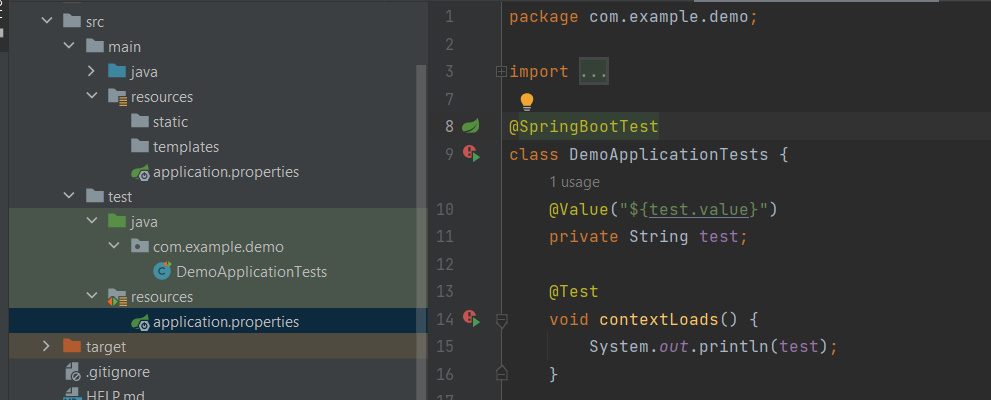
spring.datasource.url=jdbc:mysql://localhost:3306/test  
spring.datasource.username=testuser  
spring.datasource.password=Frb2eshox!  
  
test.value = Hello

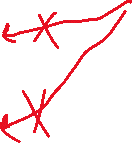
**another-app.properties**

test.value = Hello!!!

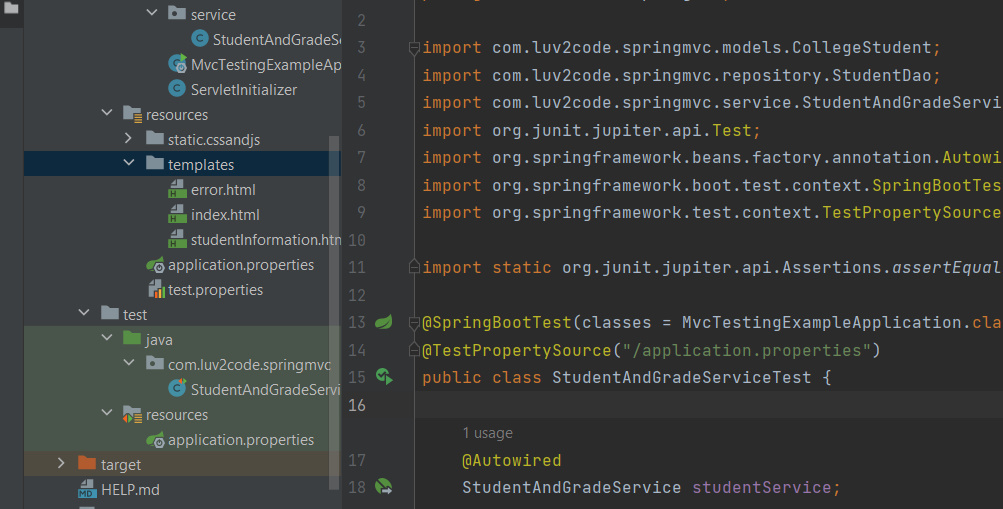
application.properties are ceva properties pe care another-app.properties nu le are. Intotdeauna spring boot test va lua properties din application.properties, indiferent ca punem noi vreun @TestPropertySource(“classpath:application.properties”) sau nu. Anotatia @TestPropertySource e facuta ca sa adaugam fisiere .properties care sa suprascrie ceea ce e in application.properties

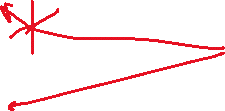
* Deci, toate properietatile pentru a ne conecta la baza de date vor fi preluate din application.properties, doar ca test.value va fi suprascris de another-app.proeperties. Deci, application.properties e mereu luat, si ne e exclus niciodata.
* Dar, daca cream un application.properties in resources de la test, application.properties din main va fi ignorat, caci e logic ca il vrem doar pe cel din test





* Folderul test poate avea si el resources. Orice clasa din el, adica /test/java va comunica mereu cu resources de la main. Deci, daca test nu are resources folder, orice resursa se va cauta in main/resources/
* Dar, daca test are un folder resources, atunci clasele din test/java vor cauta resursele intai in resources din test folder si daca nu sunt pe acolo le cauta in main/resource. E logic ca daca avem un fisier in /test/resources cu acelasi nume ca in /main/resources il vom folosi pe cel din /test/resources/

****



* E logic ca o aplicatie va avea cel putin o baza de date reala, si acum apare problema cum sa facem ca testele sa se conecteze doar la H2 DB.
* Pentru asta, trebuie sa cream un application.properties separat,anume pentru clasele de test. In el fie cream o conexiune la H2, cum e aratat mai jos, fie il lasam gol si ea automat se va crea. Dar nu uitam de

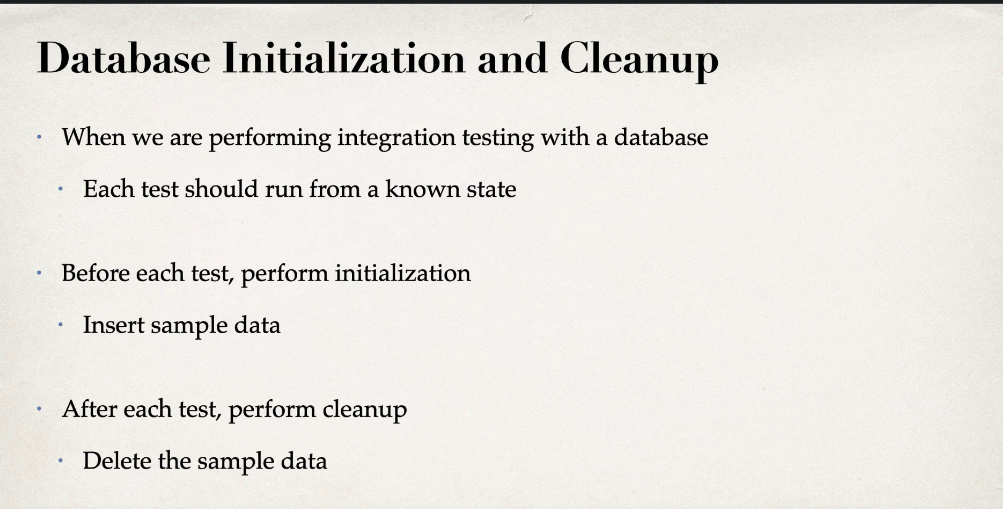
@TestPropertySource(“nume.properties”),daca vrem sa suprascriem anumite proeperties, sau cream un application.proeprties gol in test si asa se va considera ca noi nu oferim detaliile la nicio baza de date.

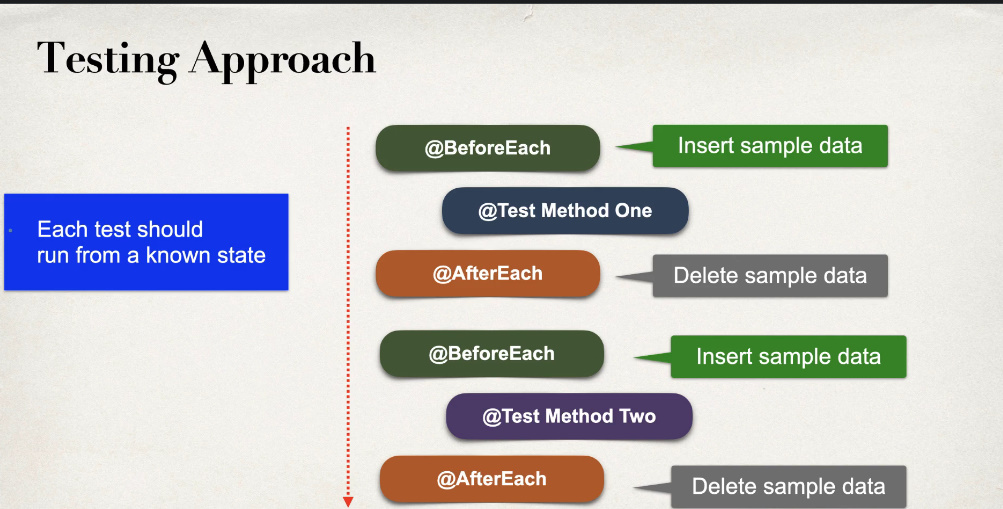
* Daca vrem sa o facem prin beanuri, cream un configuration class special si il specificam in @SpringBootTest:
* @Bean  
  public EmbeddedDatabase dataSource(){  
   return new EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.*H2*).build();  
   }

Dar nu e necesar sa facem un Bean pentru asta, asa cum ea si automat se face.

**Atentie!Clasa din proiectul de baza, ce are @SpringBootApplication si metoda main() mereu va da start la @SpringBootTest. Ca o includem noi sau nu in classes, daca numele la package e acelasi, ea mereu va fi inclusa, caci doar trebuie sa se porneasca cumva springboot si pentru test, si deci e necesara ea.**

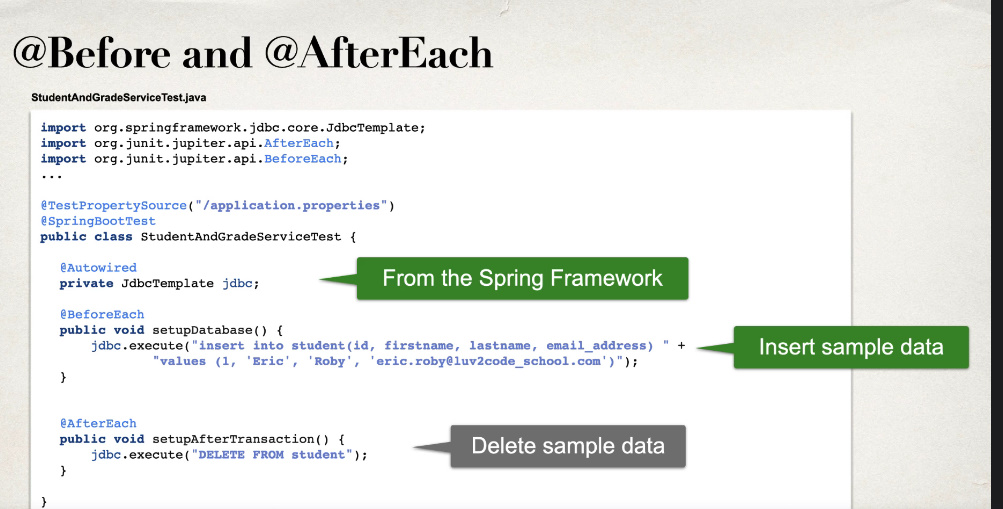
**DataBase Initialisation and CleanUp**



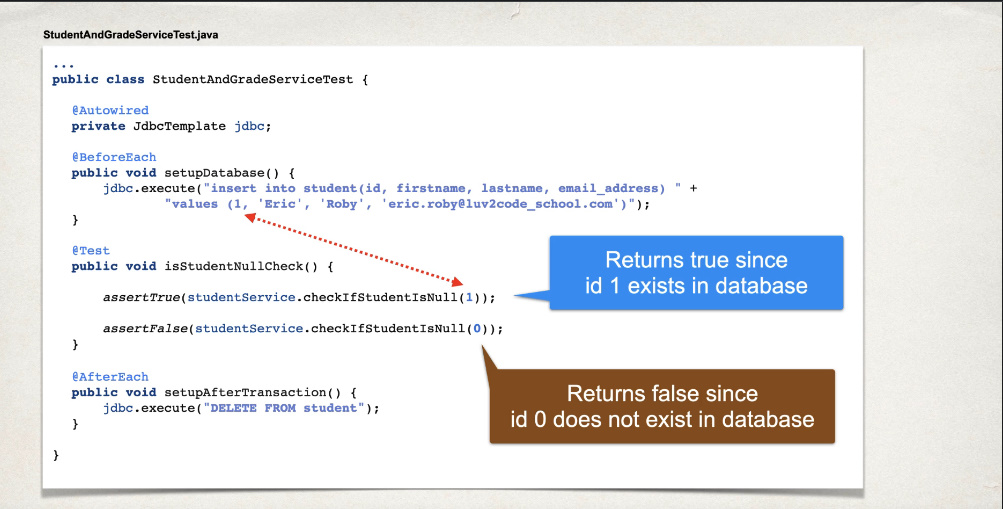


**JdbcTemplate**

* JdbcTemplate este din Spring Framework, si ne permite sa executam query in baza de date asa cum facem in baza de date chiar. Nu este doar pentru teste, o putem folosi si in app chiar.
* Deci, de asta putem pune @Autowired, caci si asa deja exista o conexiune creata de Spring cu H2
* Metoda pentru a executa un query este: **execute(“query”)**



Deci, fiecare test se va asiguraa ca inainte de executia lui, sa se insereze ceva date si apoi la finisarea lui sa se stearga.



**Config H2**

* In mod normal, nu trebuie sa scriem nimic in application.properties pentru a face o conexiune la H2, dar daca vrem, putem totusi scrie asta:
* spring.datasource.url=jdbc:h2:mem:testdb  
  spring.datasource.driverClassName=org.h2.Driver  
  spring.datasource.username=sa  
  spring.datasource.password=   
  spring.datasource.initialization-mode=*always*spring.jpa.database-platform=org.hibernate.dialect.H2Dialect  
  spring.h2.console.enabled=true  
  spring.jpa.hibernate.ddl-auto=create-drop  
  spring.jpa.show-sql = true

initialization-mode – specifica cand spring sa creeze o schema pentru embeded DB.

jpa.database-platform – seteaza dialect.

h2.console.enabled – activeaza web consola

jpa.hibernate.ddl-auto = create-drop --> face ca atunci cand se creaza baza de date embeded, sa se stearga orice tabel existent, si cand programul se incheie, sa se stearga baza de date complet.

**@Primary**

Spring @Primary annotation is **used to give a higher preference to the marked bean when multiple beans of the same type exist**

**@SQL**

* Uneori s-ar puteaa avea nevoie sa executam un query dintr-un sql file, nu sa il scriem direct in jdbcTemplate.execute()
* Deci,cream fisierul .sql undeva
* Acum, anotatia @Sql(“fisier.sql”) va face ca query sa din fisier.sql sa fie executat inaintea la @Test:

@Sql("/insertData.sql")  
@Test  
public void getGradebookService(){  
 Iterable<CollegeStudent> iterable = studentService.getGradebook();  
 List<CollegeStudent> collegeStudentList = new ArrayList<>();  
  
 for(CollegeStudent collegeStudent : iterable){  
 collegeStudentList.add(collegeStudent);  
 }  
 *assertEquals*(5,collegeStudentList.size());  
}

**Mai multe baze de date**

* Uneori, datasource, entitymanager sunt create ca beanuri in application context la aplicatie(clasa cu @SpringBootApplication), si clasa care contine testurile mereu va executa acel application context.
* De ex fie clasa de test:

@SpringBootTest(classes = {MvcTestingExampleApplication.class})  
@TestPropertySource(value = "/application-test.properties")  
@AutoConfigureMockMvc  
@SuppressWarnings(value = "unchecked")  
public class GradebookControllerTest {

* In MvcTestExampleApplication.class se gasesc beanuri de tip DataSource, LocalContainerEntityManagerFactoryBean, care sunt pentru conectarea la baza de date. Asta nu e nicio problema.
* Tot ce trebuie sa facem, e sa cream un application-test.properties sau orice nume vrem sa-i dam, si adaugam datele pentru h2 data base:

spring.datasource.url=jdbc:h2:mem:testdb  
spring.datasource.driverClassName=org.h2.Driver  
spring.datasource.username=sa  
spring.datasource.password=password  
spring.sql.init.mode=*always*spring.jpa.database-platform=org.hibernate.dialect.H2Dialect  
spring.h2.console.enabled=true  
spring.jpa.hibernate.ddl-auto=create-drop  
spring.jpa.show-sql = true

* Clasa mereu va folosi data base din .properties, nu din configuration class
* Putem in genere in application-test.properties sa nu punem tot aceea de sus. Oricum ele se vor inregistra automat.