

Projektni Zadatak

Spring Boot Testcontainers integraciono testiranje

Student Nikola Tasić

Broj indeksa 3698

Fakultet informacionih tehnologija

OAS Softversko inženjerstvo, 3. godina

Školska 2020/21. godina

01.02.2020

Opis

Cilj projekta

Cilj ovog projekta je da napravi uvod i praktičnu demonstraciju integracionog testiranja Spring Boot serverskih aplikacija i baza podataka.

Motivacija

Motivacija koja stoji iza odabira ovakve teme je prosta činjenica da moderan softver ne može da bude konstruisan bez adekvatnog testiranja. Testiranje softvera iako biva sprovedeno najčešće od strane test inženjera počinje uvek sa programerom koji taj softver razvija. Naime smatram da jedinično testiranje treba da postane deo opšte kulture developera koji razvijaju bilo koji softver bilo to da je on na skali alata koji se koristi pri razvoju nekog velikog sistema ili sam veliki sistem. Jedinični testovi omogućavaju da bez previše muke proverimo validnost koda koji pišemo, omogućavaju lakoću refactoring-a, promenljivost, održivost i donekle dokumentuju sam kod. Ali ako ima toliko mnogo prednosti zašto je testiranje *taboo* među developerima? Jedan od razloga je lenjost... Pisanje testova podrazumeva pisanje koda i to u nekim slučajevima mnogostruko više linija koda pripada otpada na testove. Samim tim dolazimo do sledećeg problema a to je održavanje. Svaki napisan kod zahteva održavanje a tako i testovi. Svaka izmena logike u kodu zahteva promenu odgovarajućih testova tako da često developeri u brzini jednostavno ne ažuriraju testove i iz istog razloga ih ne pokreću. Time *source base* projekta polako bude pretrpan kodom koji ničemu ne služi i predstavlja samo nepotreban *clutter* i otežava razumevanje (sličnu situaciju imaju i komentari, dokumentacija...).

Integraciono testiranje

Jedinični testovi kao što smo rekli rešavaju dosta problema ali nekada jednostavno nisu dovoljni. Pojedinci koji su imali (ne)sreću da rade na ogromnim projektima najbolje znaju da pokretanje i proleženje svih jediničnih testova ne znači uvek da je softver validan i spreman da deployment. Tu na scenu stupaju integracioni testovi. Integracioni testovi služe da verifikuju da sve komponente koje međusobno komuniciraju to rade na način na koji je predviđeno bez defekata. Integracioni testovi predstavljaju verifikaciju svih akcija sistema koje imaju *side-effect* oni koji su upoznati sa funkcionalnim programiranjem znaće na sta mislim. Side-effect uticaj/promena koju je sistem ili program izvršio na spoljašnji sve bilo to upis podataka u bazu ili

OBEZBEĐENJE KVALITETA, TESTIRANJE I ODRŽAVANJE SOFTVERA

HTTP zahtev ka udaljenom serveru. Svaka akcija koja se sprovodi između dve ili više komponenata sistema spada u grupu akcija koje se mogu testirati integracionim testiranjem.

Problemi integracionog testiranja

Na papiru to deluje gotovo identično kao jedinično testiranje. Na primer imamo dve komponente sistema koje komuniciraju međusobno. Recimo dva HTTP servisa. Pokrećemo test koji šalje HTTP zahtev sa komponente 1 ka komponenti 2, ona vraća odgovor naš test prolazi. Ali šta ako ne možemo da pokrenemo obe komponente u isto vreme, šta ako komponenta nije uopšte deo sistema nego eksterni servis? Tu na scenu stupaju različite metode simuliranja produkcionog okruženja u kome možemo izvršavati testove. Jedna od trenutno najaktuelnijih tehnologija koja se bavi ovime je *kontejnerizacija* i njen najuticajniji predstavnik Docker.

Kontejnerizacija

Uvod

Kontejneri kao pojava nisu uopšte nova tehnologija iako mnogima u IT svetu deluje kao da jesu (retko ko je pre pojave Docker-a čuo za termin kontejnera na nivou operativnih sistema). Kontejneri su javili kao posledica **chroot** sistemskog poziva na UNIX operativnim sistemima. **chroot** komanda pravi odvojen *root* za izvršenje programa koji po pravilu treba da bude u svakom kontekstu odvojen od ostatka sistema i samim tim se dobija sigurnost pri izvršavanju potencijalno štetnih programa. Komanda je tokom vremena imala svoje iteracije pod nazivom *jails* ali najuticajnija primena je bila u okviru Sun Microsystems Solaris Zones. Kasnije dolazi do pojave LXC (Linux Containers) koji najzad 2013 godine dovode do pojave Dockera i takozvane kontejner revolucije. Kontejneri su omogućili da možemo sa relativno malo truda dobiti ponovljiva okruženja u kojima pokrećemo softver (ponovljiva u kontekstu svaki put kada pokrenemo kontejner sa nekim softverom dobiti identično okruženje). Jedna od posledica ovoga je da napokon svi developeri mogu da direktno na svojoj mašini imaju okruženje kao ono koje će softver imati na serveru na kome bude deploy-ovan.

Primena u integracionom testiranju

Kao što smo objasnili kontejnerizacija je omogućila ta ponovljiva okruženja koja možemo lako da kreiramo i brišemo po potrebi. Kada malo bolje razmislimo veoma brzo dolazimo do zaključka da veoma lako možemo da smestimo dve komponente u kontejner i testiramo ih u onom okruženju u kakvom će inače funkcionisati u produkcionom sistemu. Primer u ovom projektu predstavlja integraciono testiranje Spring Boot aplikacije i MariaDB SQL baze podataka.

Testcontainers i Docker

Docker je sistem koji omogućava lako kreiranje i manipulaciju kontejnerima kako za testiranje tako i za upotrebu u produkcionim okruženjima. Docker-machine je softver koji koristeći postojeće sisteme za virtuelizaciju (VirtualBox, VMWare ...) omogućava umrežavanje i lakšu interakciju sa kontejnerima. I na kraju Testcontainers predstavlja JVM wrapper oko ova dva navedena softvera. Testcontainers služi da omogući Java virtuelnoj mašini da interaguje sa kontejnerima kreiranim od strane *docker-machine*. Prilikom pokretanja testova Testcontainers odnosno JVM kreira okruženje unutar kontejnera i u njemu pokreće sve testove. U našem primeru to okruženje se sastoji u suštini od Tomcat8 i MariaDB 10.2 servera nad kojima vršimo integraciono testiranje. Obzirom na to da kontejnerizacija omogućava da rekreiramo identično okruženje produkcionom možemo da budemo sigurni ukoliko naši testovi prolaze da će sistem u produkciji funkcionisati kao što je predviđeno.

Implementacija

Podešavanje Docker-a i Docker Machine

Što se tiče konfiguracije Docker-a. Prateći dokumentaciju za odgovarajući operativni sistem lako instaliramo Docker. Za Windows i MacOS su dostupni grafički interfejsi. Docker machine takođe instaliramo prateći dokumentaciju. Jedina stvar na koju treba obratiti pažnju je to da moramo da imamo makar jedan sistem za virtuelizaciju instaliran (VirtualBox VMWare ...).

Testcontainers

Testcontainers .jar fajlovi moraju biti dostupni i u classpath-u što ako

OBEZBEĐENJE KVALITETA, TESTIRANJE I ODRŽAVANJE SOFTVERA

koristimo neki od dependency managera nije preterno veliki problem. U našem primeru to je Maven.

```
<dependency>
```

```
    <groupId>org.testcontainers</groupId>
    <artifactId>testcontainers</artifactId>
    <version>1.15.1</version>
    <scope>test</scope>
```

```
</dependency>
```

```
<dependency>
```

```
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>1.15.1</version>
    <scope>test</scope>
```

```
</dependency>
```

Takođe ako želimo da imamo olakšano kreiranje kontejnera za baze podataka trebalo bi da dodamo JDBC Testcontainer driver.

```
<dependency>
```

```
    <groupId>org.testcontainers</groupId>
    <artifactId>jdbc</artifactId>
    <version>1.15.1</version>
    <scope>test</scope>
```

```
</dependency>
```

```
<dependency>
```

```
    <groupId>org.testcontainers</groupId>
    <artifactId>mariadb</artifactId>
    <version>1.15.1</version>
    <scope>test</scope>
```

```
</dependency>
```

Ovakva konfiguracija takođe sa JUnit-om je dovoljna da možemo da konfigurishemo bazične integracione testove sa bazom podataka.

Spring Boot

Najjednostavnija konfiguracija zahteva od nas da izmenimo datasource url koji će kasnije Testcontainers pročitati i na osnovu njega kreirati kontejner sa inicijalizovanom bazom podataka.

```
spring.datasource.username=rzp-root
spring.datasource.password=rzp-root
spring.datasource.url=jdbc:tc:mariadb:10.2://127.0.0.1:3306\
/rzp-database?useUnicode=true&TC_INITSCRIPT=file:ddl/rzp.ddl
spring.datasource.driver-class-name=org.testcontainers.jdbc.ContainerDatabaseDriver
spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect
```

Na ovom primeru iz u datasource.url property ubacujemo *tc:* oznaku za protokol koja govori Testcontainers da treba da kreira kontejner na osnovu ostalih parametara. *TC_INITSCRIPT* parametar predstavlja relativnu putanju do SQL skripte koja služi za inicijalizaciju baze podataka. Nakon instanciranja kontejnera i kreiranje MariaDB servera Testcontainers će pokrenuti tu skriptu i inicijalizovati tabele. Naravno u toj skripti se mogu nalaziti i test podaci ali moja lična preferenca je da test podatke ubacujem i brišem u okviru samih testova.

Testiranje

Kreiranje JUnit klasa i testova je veoma trivijalno koristeći Testcontainers. Potrebna je samo anotacija *@Testcontainers* a ostatak je već konfigurisan u application.properties.

```
@Testcontainers
@SpringBootTest
@ActiveProfiles("test")
class UserServiceTests {
    @Autowired
    UserRepository userRepository;
    @Autowired
    RoleRepository roleRepository;
    @Autowired
    UserService userService;
    @Autowired
    RoleService roleService;
```


OBEZBEĐENJE KVALITETA, TESTIRANJE I ODRŽAVANJE SOFTVERA

Kao što se može primetiti radimo direktno umetanje zavisnosti bez kreiranja mock objekata jer u kontejneru se zapravo instancira i inicijalizuje ceo Spring Context tako da ono što na kraju dobijemo su realni Bean-ovi onakvi kakvi će postojati u produkcionom okruženju. Kada imamo ovakav setup veoma je lako napisati testove koji će upisivati ili čitati podatke iz baze.

@Test

```
void test_findByUsername() {
    User user = getUser();
    user = userService.save(user);
    User foundUser = userService.findByUsername(user.getUsername());

    assertEquals(user.getUsername(), foundUser.getUsername());
    assertEquals(user, foundUser);
}
```

@Test

```
void test_update(){
    User user = getUser();
    user = userService.save(user);
    String expectedUsername = "newUsername";
    user.setUsername(expectedUsername);
    user = userService.update(user);

    assertEquals(expectedUsername, user.getUsername());
}
```

@Test

```
void test_uploadFile() throws IOException {
    MultipartFile file = getMultipartFile(mockFile, contentType: "image/png");
    Media media = mediaService.upload(file);

    assertNotNull(media);
    assertTrue(new File(media.getUri()).exists());
}
```

@Test

```
void test_deleteFile() throws IOException {
    MultipartFile file = getMultipartFile(mockFile, contentType: "image/png");
    Media media = mediaService.upload(file);

    assertNotNull(media);

    mediaService.deleteById(media.getId());

    assertFalse(new File(media.getUri()).exists());
}
```

Na identičan način kao što bi to radili prilikom jediničnih testova možemo kreirati *setup* i *teardown* metode.

OBEZBEĐENJE KVALITETA, TESTIRANJE I ODRŽAVANJE SOFTVERA

```
@AfterEach
void teardown() {
    postRepository.deleteAll();
    tagRepository.deleteAll();
    categoryRepository.deleteAll();
}

@BeforeEach
void classSetup() throws IOException {
    mockFile = new File(pathname: "test.png");
    createMockImage(mockFile);
}
```

Pokretanje testova

Pokretanje testova vrši SpringRunner u kombinaciji sa Testcontainers. U ovom primeru imamo bazičnu konfiguraciju koja pokreće redom sve testove (testovi takođe mogu biti paralelizovani ili podešeni da koriste više kontejnera).

```

  Δ / - - - - ( ) - - - - \ \ \ \
( ) \ | - - - - \ \ \ \ \ \ \ \
\ \ \ \ | | | | | | | | | | \ \ \ \
  | - - - - | | | | | | | | | | \ \ \ \
  :: Spring Boot ::      (v2.3.2.RELEASE)

```

```

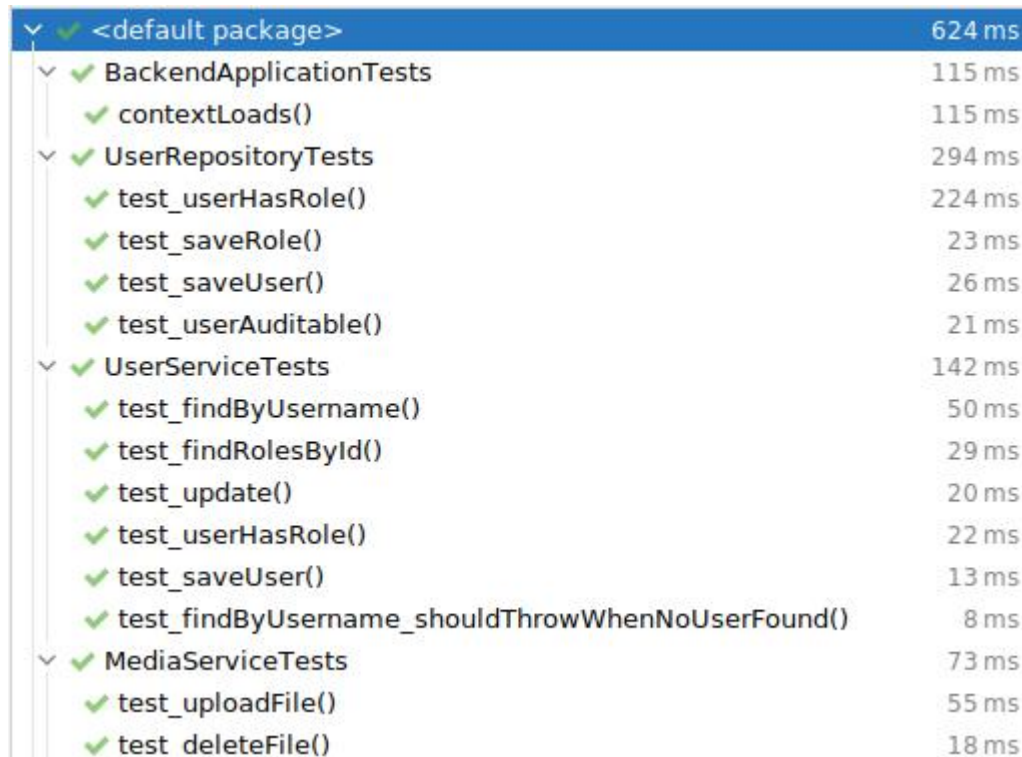
2021-02-01 18:19:06.558 INFO 1660462 --- [main] c.e.b.repository.UserRepositoryTests : Starting User
2021-02-01 18:19:06.558 INFO 1660462 --- [main] c.e.b.repository.UserRepositoryTests : The following
2021-02-01 18:19:06.689 INFO 1660462 --- [main] s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping
2021-02-01 18:19:06.718 INFO 1660462 --- [main] s.d.r.c.RepositoryConfigurationDelegate : Finished Spri
2021-02-01 18:19:06.787 INFO 1660462 --- [main] trationDelegate$BeanPostProcessorChecker : Bean 'org.spr
2021-02-01 18:19:06.794 INFO 1660462 --- [main] trationDelegate$BeanPostProcessorChecker : Bean 'methods
2021-02-01 18:19:06.853 INFO 1660462 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing
2021-02-01 18:19:06.857 INFO 1660462 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-2
2021-02-01 18:19:06.923 WARN 1660462 --- [main] o.t.utility.TestcontainersConfiguration : Attempted to
2021-02-01 18:19:07.139 INFO 1660462 --- [main] t.d.DockerMachineClientProviderStrategy : Found docker-
2021-02-01 18:19:07.744 INFO 1660462 --- [main] o.t.utility.DockerMachineClient : Docker-machin

2021-02-01 18:19:58.380 DEBUG 1660462 --- [main] org.hibernate.SQL :
select
    user0_.user_id as user_id1_8_0_,
    user0_.created_date as created_2_8_0_,
    user0_.last_modified_by as last_mod3_8_0_,
    user0_.last_modified_date as last_mod4_8_0_,
    user0_.record_status as record_s5_8_0_,
    user0_.about as about6_8_0_,
    user0_.display_name as display_7_8_0_,
    user0_.email as email8_8_0_,
    user0_.first_name as first_na9_8_0_,
    user0_.last_name as last_na10_8_0_,
    user0_.password as passwor11_8_0_,
    user0_.username as usernam12_8_0_
from
    user user0_
where
    user0_.user id=?

```


OBEZBEĐENJE KVALITETA, TESTIRANJE I ODRŽAVANJE SOFTVERA

Na kraju izvršenih testova dobijamo identičan prikaz kao kod običnih jediničnih testova. Klikom na pojedinačni test IntelliJ prikazuje deo standardnog izlaza koji je dobijen tokom izvršenja izabranog testa.



✓ <default package>	624 ms
✓ BackendApplicationTests	115 ms
✓ contextLoads()	115 ms
✓ UserRepositoryTests	294 ms
✓ test_userHasRole()	224 ms
✓ test_saveRole()	23 ms
✓ test_saveUser()	26 ms
✓ test_userAuditable()	21 ms
✓ UserServiceTests	142 ms
✓ test_findByUsername()	50 ms
✓ test_findRolesById()	29 ms
✓ test_update()	20 ms
✓ test_userHasRole()	22 ms
✓ test_saveUser()	13 ms
✓ test_findByUsername_shouldThrowWhenNoUserFound()	8 ms
✓ MediaServiceTests	73 ms
✓ test_uploadFile()	55 ms
✓ test_deleteFile()	18 ms

Posle izvršenja svih testova JVM gasi kontejner i svi podaci su obrisani.

Zaključak

U ovom projektu smo prikazali samo jedan deo testiranja koja se mogu izvršiti koristeći kontejnere. Pored integracionog testiranja baze moguće je vršiti integraciona testiranja više različitih servisa, GUI testiranja koristeći Selenium i još mnogo toga. Kontejnerizacija je jedna od tehnologija koja je najviše pospešila produktivnost u različitim sferama IT industrije u sada u prehodnoj deceniji. Jedna od tehnologija gde se konstantno otkrivaju nove primene i definitivno jedna od najzanimljivijih za proučavanja bilo sa tačke gledišta upotrebe ili daljeg ravijanja.