Java testiranje

Autori:
Goran Savić
Milan Segedinac

1. Testiranje

Standardan deo životnog ciklusa razvoja i upotrebe softvera je i provera ispravnosti softvera. Postupci koji se u ovu svrhu sprovode se objedinjeno nazivaju obezbeđenje kvaliteta (eng. *quality assurance* – QA). Važan deo obezbeđenja kvaliteta je testiranje softvera. Testiranje je proces koji se sastoji od svih aktivnosti u životnom ciklusu softvera vezanih za planiranje, pripremu i izvršavanje zadataka koji treba da pokažu da li softver zadovoljava specificirane zahteve i ispunjava namenu, kao i da detektuju greške u softveru.

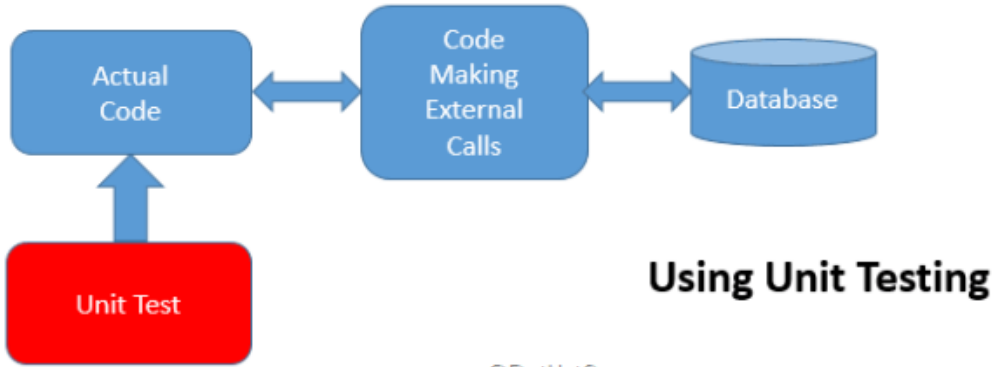
Postoje različiti tipovi testiranja. Prema jednom kriterijumu klasifikacije, testiranje možemo podeliti na **statičko** i **dinamičko**. Statičko testiranje podrazumeva procenu kvaliteta softvera analizom programskog koda i pratećih dokumenata (bez izvršavanja softvera). Dinamičko testiranje se zasniva na izvršavanju programskog koda i poređenju da li se stvarno ponašanje softvera poklapa sa očekivanim ponašanjem.

Prema drugom kriterijumu klasifikacije, testiranje delimo na **manuelno** i **automatsko**. Kod manuelnog testiranja, akcije koje se odnose na verifikaciju kvaliteta softvera izvršava čovek (npr. ručno unosi podatke u aplikaciju, koristi aplikaciju da bi proverio funkcionalnosti, analizira programski kod, ...). Sa druge strane, kod dinamičkog testiranja ove akcije izvršava računarski program (npr. kompajler vrši pronalaženje sintaksnih grešaka u kodu, specijalizovani programski kod simulira ponašanje aplikacije od strane korisnika itd).

Treći kriterijum klasifikacije testiranja koji ćemo ovde navesti se odnosi na količinu dokumenata, koda ili funkcionalnosti koja je pokrivena testom. U tom smislu razlikujemo jedinično testiranje, integraciono testiranje i *end-to-end* testiranje. Kod jediničnog testiranja, test provera kvalitet jedne jedinice softvera (npr. jedne klase ili metode u programskom kodu). Integracioni test se odnosi na proveru ispravnosti dela softvera koji uključuje komunikaciju više komponenti (npr. testiranje poslovne logike zajedno sa perzistencijom podataka). Konačno, *end-to-end* testiranje je testiranje kompletno integrisanog sistema da bi se utvrdilo da li sistem u celini zadovoljava zahteve (npr. provera da li korisnička akcija za dodavanjem entiteta rezultira novim entitetom u bazi podataka).

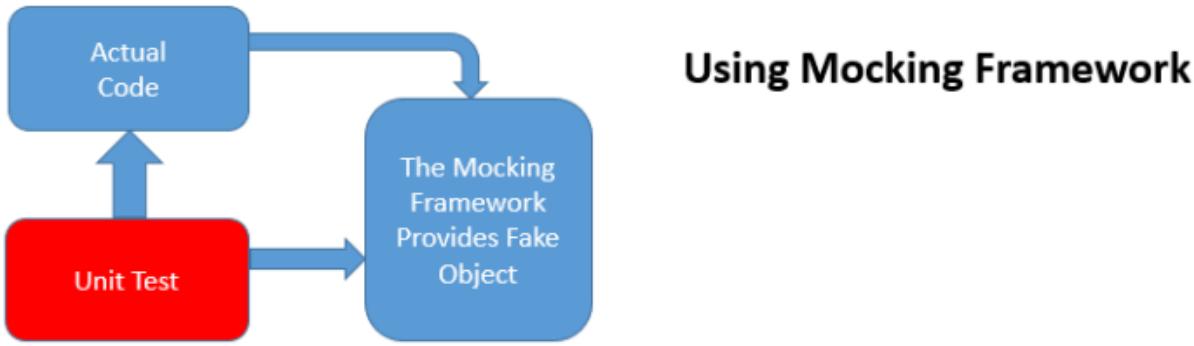
Test dvojnici

Važna karakteristika jediničnih testova je da moraju biti izolovani. To znači da na uspešnost testa sme da utiče samo ispravnost koda u jedinici koja se testira. Ako jedinica koju testiramo pri izvršavanju poziva kod iz drugog objekta, tada u slučaju neuspešnog testa ne možemo znati da li je greška uzrokovana od strane objekta koji testiramo ili tog drugog objekta. Ovo je ilustrovano na slici.



* preuzeto sa www.dotnetcurry.com

Iz tog razloga se pri jediničnom testiranju koriste test dvojnici (eng. *test double*). Umesto pravih objekata koje klasa referencira postavljaju se objekti dvojnici koji pojednostavljuju ili simuliraju ponašanje referenciranih objekata. Obzirom da je kod koji sadrže trivijalan i ne dovodi do greške, test dvojnici omogućuju da se test izvrši, a da bude fokusiran samo na objekat koji testira. Princip testiranja korišćenjem test dvojnika prikazan je na narednoj slici.



* preuzeto sa www.dotnetcurry.com

Postoji više tipova test dvojnika:

- *fake* objekat – objekat koji određenu funkcionalnost implementira na jednostavniji način za potrebe testa (npr. *fake* objekat perzistira podatke u memoriji, za razliku od stvarnog objekta koji podatke perzistira u bazi podataka)
- *stub* objekat – simulira izlaze stvarnog objekta tako što na predefinisane ulaze vraća predefinisane izlaze
- *mock* objekat – simulira i ponašanje stvarnog objekta. Ako stvarni kod sadrži sekvencu poziva određenih metoda, *mock* objekat ponavlja ovu sekvencu, ali sami rezultati poziva su simulirani (predefinisani)
- *spy* objekat – predstavlja modifikovani stvarni objekat. Deo ponašanja je stvarno ponašanje, a deo ponašanja je simuliran zbog potreba testiranja. Korišćenjem *spy* objekata test može da dobija stvarno ponašanje jednog dela, a simulirano ponašanje drugog dela funkcionalnosti

2. Java jedinično testiranje

Dinamičko jedinično testiranje Java aplikacija se vrši pisanjem Java programskog koda, koji poziva funkcionalnosti objekta čije ponašanje testiramo. Dobijeni rezultat se upoređuje sa očekivanim rezultatom. Ukoliko se ova dva rezultata poklapaju, test je prošao uspešno. Postoje razvijeni radni okviri koji pružaju pomoć u pisanju koda za testiranje, tako što sadrže specijalizovane biblioteke za testiranje. Takođe, ovi radni okviri omogućuju izvršavanje testova i dobijanje izveštaja o uspešnosti izvršenih testova. Najčešće korišćeni radni okvir za testiranje u Java aplikacijama je JUnit.

Najvažniji deo testa je verifikacija dobijenog rezultata, što se sprovodi poređenjem očekivane i stvarne vrednosti. Ovo se sintaksno vrši korišćenjem tvrdnji (eng. *assertions*). *Assert* izraz je deo koda u kojem se zahteva da određeni izraz bude istinit. Ako to nije slučaj, program prestaje da radi ili se dešava izuzetak. Ovo se koristi u testiranju pri utvrđivanju da li stvarna vrednost odgovara očekivanoj. Ako to nije slučaj, test nije uspešan.

JUnit sadrži sledeće *assert* izraze:

- `assertTrue()` - istinito ako je prosleđeni parametar *true*
- `assertFalse()` - istinito ako je prosleđeni parametar *false*
- `assertEquals()` - istinito ako su dva prosleđena objekta jednaka (poređenje se vrši pozivom metode *equals*)
- `assertArrayEquals()` - istinito ako su dva prosleđena niza jednaka (imaju sve elemente jednake)
- `assertNull()` - istinito ako je prosleđeni parametar *null*
- `asertNotNull()` - istinito ako prosleđeni parametar nije null
- `assertSame()` - istinito ako dve prosleđene reference predstavljaju reference na isti objekat
- `assertNotSame()` - istinito ako dve prosleđene reference ne referenciraju isti objekat

JUnit predviđa pisanje testova u proizvoljnim metodama anotiranim @Test anotacijom. JUnit će omogućiti izvršavanje svih ovih metoda i prikaz izveštaja o uspešnosti testova. Ukoliko je potrebno izvršiti pripremu određenih objekata za testiranje, moguće je napisati dodatni kod koji će se izvršavati pre testova. Ovo se vrši u proizvoljnoj metodi anotiranoj @Before anotacijom. Slično, postoji i @After anotacija za kod koji se izvršava nakon testa.

Pogledajmo primer provere ispravnosti rada metode korišćenjem JUnit biblioteke, za metodu *concatenate* koja prima dva stringa kao parametar i čiji je zadatak da vrati novi string koji predstavlja spoj dva prosleđena stringa.

```
@Test
public void testConcatenate() {
      ExampleClass example = new ExampleClass();
      String result = example.concatenate("one", "two");
      assertEquals("onetwo", result);
}
```

U prikazanom primeru se najpre poziva metoda *concatenate* da bismo dobili njen stvarni rezultat. Nakon toga se ovaj rezultat poredi sa očekivanim rezultatom. *Assert* izraz će zaustaviti test i označiti ga kao neuspešan ako ove dve vrednosti nisu jednake.

3. Testiranje Spring Boot veb aplikacije

U ovom poglavlju ćemo objasniti kako možemo testirati Spring Boot višeslojnu veb aplikaciju kakva je predstavljena u prethodnim lekcijama. Ovakva aplikacija ima tri sloja - sloj za upravljanje podacima (Spring Data JPA repozitorijumi), sloj za poslovnu logiku (Spring servisi), i sloj za komunikaciju sa klijentom (Spring REST kontroleri).

Ako sloj testiramo izolovano putem jediničnog testa, tada je neophodno postaviti test dvojnike na mesto slojeva sa kojima testirani sloj komunicira. Ako realizujemo integracioni test, tada testirani sloj zaista komunicira sa drugim slojevima i time testiramo rad više slojeva odjednom.

Spring ima podršku za testiranje aplikacije kroz automatsko obezbeđivanje Spring Bean objekata koje test koristi, kao i kreiranje test dvojnika koji simuliraju ponašanje određenog Spring Bean objekta. Pri testiranju Spring aplikacije standardno se koristi JUnit radni okvir. Samo izvršavanje testova zadatak je JUnit komponente koja se naziva *test runner*. Ova komponenta vrši instanciranje objekta koji sadrži *@Test* metode, izvršava test metode i kreira izveštaj o testu. Postoje različite implementacije *test runner* komponente. Na primer, JUnit ima ugrađeni konzolni *test runner,* dok Eclipse sadrži svoj grafički *test runner*. Moguće je koristiti proizvoljan *test runner*, pa se tako za testiranje delova Spring aplikacije koristi *test runner* definisan u klasi *SpringRunner*.

Da bi testovi mogli da pozivaju funkcionalnosti Spring Bean objekata, neophodno je učitati ove objekte. Anotacijom @SpringBootTest vrši se inicijalizacija Spring kontejnera pri pokretanju testova, kako bi bili dostupni svi objekti kojima Spring aplikacija upravlja.

Pogledajmo detalje testiranja svakog od pomenutih slojeva aplikacije.

Testiranje JPA sloja

Upravljanje podacima je definisano u Spring Data JPA repozitorijumima. Za testiranje njihovih funkcionalnosti, potrebno je da nad instancom repozitorijuma pozivamo željenu metodu i proverimo rezultat koji vraća. U nastavku je dat primer testa koji verifikuje da li sloj za upravljanje podacima ispravno preuzima države na osnovu broja stanovnika.

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class CountryRepositoryIntegrationTest {
  @Autowired
  CountryRepository countryRepository;

  @Test
  public void testFindByPopulation() {
    List<Country> countries = countryRepository.findByPopulation(60800000);

    assertEquals(1, countries.size());
    assertEquals(Long.valueOf(3), countries.get(0).getId());
    assertEquals("Italy", countries.get(0).getName());
    assertEquals(60800000, countries.get(0).getPopulation());
  }
}
```

Vidimo da se na početku koda definiše *test runner*, kao i učitavanje Spring Boot okruženja. Pri pokretanju testa, izvršiće se pokretanje Spring servera na proizvoljnom portu koji će Spring automatski odrediti. Klasa čiju funkcionalnost testiramo je *CountryRepository* i

jedan njen objekat će u test biti dobavljen automatski kroz injekciju zavisnosti definisanu anotacijom @Autowired.

Sam test najpre vrši poziv metode repozitorijuma čiju ispravnost želi da proveri. Nakon toga, proverava se da li se dobijeni rezultat poklapa sa očekivanim rezultatom. *Assert* izrazi redom proveravaju da li je dobijena tačno jedna država kao rezultat, da li je njen identifikator 3, ime Italija, a broj stanovnika 60800000. Ako bilo koja od provera utvrdi odstupanje stvarne od očekivane vrednosti, test će završiti neuspešno.

Prikazani test se može smatrati integracionim testom, obzirom da se kod u repozitorijumu obraća bazi podataka, pa ispravnost testa zavisi kako od koda u repozitorijumu, tako i od baze podataka. Ako želimo da implementiramo jedinični test, morali bismo simulirati ponašanje baze podataka, tako da njen rad ne može da ugrozi ispravnost testa. Spring sadrži podršku za izolovano testiranje JPA repozitorijuma, tako da se pri izvršavanju testa podaci skladište u privremenoj bazi podataka u memoriji (eng. *in-memory database*). U nastavku je prikazan prethodni test izmenjen tako da bude jedinični kroz korišćenje *in-memory* baze podataka.

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class CountryRepositoryUnitTest {
  @Autowired
  private TestEntityManager entityManager;

  @Autowired
  CountryRepository countryRepository;

  @Test
  public void testFindByPopulation() {
    entityManager.persist(new Country("Serbia", 7098000));
    entityManager.persist(new Country("France", 66810000));
    entityManager.persist(new Country("Italy", 60800000));

    List<Country> countries = countryRepository.findByPopulation(60800000);

    assertEquals(1, countries.size());
    assertEquals(Long.valueOf(3), countries.get(0).getId());
    assertEquals("Italy", countries.get(0).getName());
    assertEquals(60800000, countries.get(0).getPopulation());
  }
}
```

Korišćenje privremene baze podataka specificirano je anotacijom @DataJpaTest. Koja konkretno baza podataka će biti korišćena zavisi od biblioteke koja je dodata u CLASSPATH aplikacije. Ovo se specificira u spisku zavisnosti aplikacije u alatu za izgradnju projekta (npr. Maven pom.xml fajl). Komunikacija sa privremenom bazom podataka se vrši putem injektovanog *entityManager* objekta specijalizovanog za korišćenje u testovima i predstavljenog klasom *TestEntityManager*.

Vidimo da test metoda na svom početku sada ubacuje u privremenu bazu 3 države. Poziv metode *findByPopulation* će se obratiti ovoj privremenoj bazi podataka. Ostatak koda proverava da li su dobijeni rezultati očekivani i isti je kao u prethodnom primeru.

Testiranje servisa

Princip testiranja servisa je isti kao i za repozitorijume. Neophodno je pozvati metodu servisa koja se testira i proveriti dobijeni rezultat. Pri tome, Spring će dobaviti objekte kojima test pristupa.

Pogledajmo primer testiranja jedne metode poslovne logike, koja treba da izvrši klasifikaciju države na osnovu njenog broja stanovnika. Metoda vraća rezultat 1, 2 ili 3 u zavisnosti od toga da li država ima više od 10 miliona stanovnika, između 1 i 10 miliona stanovnika ili manje od milion stanovnika. U nastavku je dat kod testa za proveru ispravnosti ove funkcionalnosti.

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class CountryServiceIntegrationTest {
	@Autowired
	CountryService countryService;

	@Test
	public void testClassifyCountry() {
		int result = countryService.classifyCountry(1L);
		assertEquals(2, result);
	}
}
```

Vidimo da se automatski injektuje objekat klase *CountryService*, čija se funkcionalnost testira. Metodi *classifyCountry* se prosleđuje identifikator države Srbije, pa je očekivani rezultat metode 2, obzirom da Srbija ima između 1 i 10 miliona stanovnika. Prikazani *assert* izraz vrši poređenje očekivanog i dobijenog rezultata.

Slično primeru sa testiranjem repozitorijuma, i upravo prikazani test je integracioni, obzirom da će se metoda *classifyCountry* obratiti repozitorijumu za dobavljanje države. U slučaju greške, postoji mogućnost da je grešku izazvao kod u repozitorijumu, a ne u metodi *classifyCountry* koju testiramo. Za izolovano testiranje servisa neophodno je na mesto repozitorijuma sa kojim servis komunicira postaviti test dvojnika koji će simulirati ponašanje repozitorijuma. Pogledajmo detalje na sledećem primeru. Ovaj primer predstavlja izmenu prethodnog primera tako da se umesto repozitorijuma koristi njegov test dvojnik.

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class CountryServiceUnitTest {

	@Autowired
	CountryService countryService;

	@MockBean
	CountryRepository countryRepository;

	@Before
	public void setup() {
		given(
			countryRepository.findOne(1L)
		).willReturn(
			new Country(1L, "Serbia", 7098000)
		);
	}
```

```
	@Test
	public void testClassifyCountry() {
		int result = countryService.classifyCountry(1L);
		assertEquals(2, result);
	}
}
```

Spring može automatski kreira test dvojnika za određenu klasu putem anotacije @MockBean, što je u prikazanom primeru i učinjeno za *CountryRepository* Bean. Ova anotacija će kreirati objekat koji ima isti interfejs kao i originalni objekat, ali će ponašanje biti simulirano kroz vraćanje predefinisanih izlaza na predefinisane ulaze. Ovo predefinisano ponašanje test dvojnika je definisano u metodi *setup* koja će se pozvati pre izvršavanja testa. Za definisanje ponašanja test dvojnika, Spring koristi Mockito biblioteku, koja je najpopularnija Java biblioteka ovog tipa. Metoda *given* definiše predefinisani ulaz, a metoda *willReturn* predefinisani izlaz. U konkretnom primeru vidimo da će poziv metode *findOne* repozitorijuma (što je poziv koji vrši testirana metoda *classifyCountry* iz servisa), vratiti državu Srbiju kao rezultat. Dakle, neće se vršiti preuzimanje države iz baze, nego će odmah biti vraćena instanca klase *Country* inicijalizovana podacima koji opisuju Srbiju. Na ovaj način se test neće obraćati repozitorijumu, pa samim tim ispravnost repozitorijuma ne utiče na ispravnost testa. Time izolovano testiramo ponašanje servisa, jer samo kod napisan u servisu utiče na ispravnost testa.

Sama metoda za testiranje je ista kao u prethodnom primeru, s tim da metoda *classifyCountry* neće zaista komunicirati sa repozitorijumom, nego samo sa njegovim test dvojnikom.

Testiranje kontrolera

Za testiranje Spring REST kontrolera, potrebno je uputiti HTTP zahtev ka odgovarajućem URL-u. Spring pruža podršku za izvršavanja ovakvih zahteva u testovima putem klase *TestRestTemplate*, koja može biti injektovana u test. Analizom rezultata odgovora koji je REST kontroler vratio, utvrđuje se da li je test prošao. Pogledajmo primer testiranja REST kontrolera za preuzimanje država.

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class CountryControllerTest {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void testGetCountries() {
        ResponseEntity<CountryDTO[]> responseEntity =
           restTemplate.getForEntity("/api/countries", CountryDTO[].class);

        CountryDTO[] countries = responseEntity.getBody();

        assertEquals(HttpStatus.OK, responseEntity.getStatusCode());
        assertEquals(3, countries.length);
        assertEquals(Long.valueOf(3), countries[2].getId());
        assertEquals("Italy", countries[2].getName());
        assertEquals(60800000, countries[2].getPopulation());
    }
}
```

Metodom *getForEntity* klase *TestRestTemplate* upućuje se zahtev ka URL-u koji je specificiran u prvom parametru metode. Drugi parametar metode specificira tip odgovora. Spring će automatski izvršiti deserijalizaciju vraćenog JSON dokumenta u objekat navedenog tipa. U konkretnom primeru, dobija se odgovor (predstavljen klasom *ResponseEntity*) koji u telu sadrži niz objekata klase *CountryDTO* koja reprezentuje državu.

Za proveru ispravnosti dobijenog rezultata, najpre se proverava da li HTTP odgovor ima status OK – 200. Nakon toga, proverava se da li su ukupno vraćene 3 države, kao i da li treća država u nizu (država sa indeksom 2) ima identifikator 3, naziv Italy i broj stanovnika 60800000.

Testiranje kontrolera sa kontrolom prava pristupa

Prethodni test kontrolera podrazumeva da aplikacija dozvoljava pristup API-ju bez provere prava pristupa. Ako aplikacija sprovodi kontrolu pristupa, prethodno prikazani test neće biti uspešan, jer u HTTP zahtevu nisu navedene informacije neophodne da server izvrši autorizaciju. Pogledajmo kako možemo testirati REST kontrolere u Spring aplikaciji koja koristi kontrolu pristupa baziranu na JWT tokenima.

Kao što smo ranije objasnili, kod ovog tipa autentikacije korisnik pri prijavi dobija token koji kasnije šalje u okviru HTTP zahteva da bi imao dozvolu da izvrši traženu funkcionalnost. Iz tog razloga, pre izvršavanja testa, pozvaćemo REST servis za prijavu na sistem kako bismo dobili token koji ćemo kasnije koristiti za pristup pri drugim REST servisima pri njihovom testiranju. Metoda koja vrši dobavljanje tokena je prikazana u nastavku.

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class CountryControllerTest {

  @Autowired
  private TestRestTemplate restTemplate;

  private String accessToken;

  @Before
  public void login() {
  ResponseEntity<String> responseEntity =
    restTemplate.postForEntity("/api/login",
                               new LoginDTO("admin", "12345"),
                               String.class);
    accessToken = responseEntity.getBody();
  }
  ...
}
```

Atribut *accessToken* predstavlja JWT token koji se dobija pri prijavi. Prijava na sistem će biti obavljena pre izvršavanja testa, što je definisano anotacijom @Before. Kao i u prethodnom primeru, za pristup REST servisu koristi se objekat klase *TestRestTemplate*. Vidimo da na REST servis za prijavu putem POST zahteva šaljemo korisničko ime i lozinku upakovane u LoginDTO objekat. Kao rezultat od servera očekujemo String koji predstavlja token za pristup. Vidimo da se po dobijanju odgovora, token preuzima iz tela odgovora i upisuje u atribut *accessToken* kako bi se mogao koristiti u testovima.

Kada smo dobili token za pristup, može se napisati kod koji testira željeni REST servis. Pri pristupu REST servisu, potrebno je u zaglavlju zahteva poslati i token za pristup. Pogledajmo ponovo testiranje REST servisa za preuzimanje država, s tim da je ovog puta pristup dozvoljen samo ako HTTP zahtev sadrži ispravan JWT token.

```
@Test
public void testGetCountries() {
  HttpHeaders headers = new HttpHeaders();
  headers.add("X-Auth-Token", accessToken);

  HttpEntity<Object> httpEntity = new HttpEntity<Object>(headers);

  ResponseEntity<CountryDTO[]> responseEntity =
    restTemplate.exchange("/api/countries", HttpMethod.GET, httpEntity,
                    CountryDTO[].class);

  CountryDTO[] countries = responseEntity.getBody();

  assertEquals(HttpStatus.OK, responseEntity.getStatusCode());
  assertEquals(3, countries.length);
  assertEquals(Long.valueOf(3), countries[2].getId());
  assertEquals("Italy", countries[2].getName());
  assertEquals(60800000, countries[2].getPopulation());
}
```

Vidimo da se na početku testa definiše zaglavlje HTTP zahteva. U zaglavlje se postavlja token u polje *X-Auth-Field*, obzirom da je to polje zaglavlja iz kojeg server preuzima JWT token. Zahtev je predstavljen objektom klase HttpEntity, pri čemu se pri instanciranju ovog objekta postavlja definisano zaglavlje. Obzirom da ćemo vršiti GET metodu ka serveru, ovaj zahtev nema telo, već samo zaglavlje.

Da bismo definisani zahtev sa zaglavljem poslali REST servisu, koristimo metodu *exchange* klase TestRestTemplate. Obzirom da zaglavlje sadrži ispravan token, ako metoda ispravno obavlja posao, dobićemo u odgovoru niz država predstavljenih objektima klase CountryDTO. Ostatak koda koji proverava ispravnost dobijenog niza država je isti kao i u ranijem primeru koji nije imao kontrolu pristupa.