# Automatizirano testiranje internet servisa

**Ušković, Luka**

**Undergraduate thesis / Završni rad**

**2020**

*Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj:* **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

*Permanent link / Trajna poveznica:* https://urn.nsk.hr/urn:nbn:hr:200:252360

*Rights / Prava:* In copyright / Zaštićeno autorskim pravom.

*Download date / Datum preuzimanja:* **2024-04-26**

*Repository / Repozitorij:*

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

**SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU**

**FAKULTET ELEKTROTEHNIKE, RAČUNALSTVA I INFORMACIJSKIH TEHNOLOGIJA**

**Stručni studij**

# AUTOMATIZIRANO TESTIRANJE REST API WEB SERVISA

**Završni Rad**

**Luka Ušković**

**Osijek, 2019**

# SADRŽAJ

*1. UVOD* ..... *1*

*2. PREGLED KORIŠTENIH TEHNOLOGIJA* ..... *2*

**2.1. Java programski jezik** ..... **2**

**2.2. JSON** ..... **2**

**2.3. REST komunikacijski protokol** ..... **3**

**2.4. Platforma za testiranje** ..... **4**

**2.5. Java Biblioteka za slanje i ovjeru zahtjeva** ..... **6**

**2.6. Java biblioteka za generiranje izvješća** ..... **7**

*3. PLANIRANJE PRISTUPA I REALIZACIJE AUTOMATSKIH TESTOVA* ..... *9*

**3.1. Pogled na funkcionalnosti REST web servisa** ..... **9**

**3.2. Izrada testnih scenarija za automatizaciju** ..... **16**

*4. IMPLEMENTACIJA AUTOMATSKIH TESTOVA* ..... *19*

**4.1. Postavljanje strukture projekta** ..... **19**

**4.2. Definiranje konstanti** ..... **20**

**4.3. Definiranje modela** ..... **21**

**4.4. Definiranje automatskih testova** ..... **23**

**4.5. Pokretanje testova i pregled testnog izvješća** ..... **30**

*5. ZAKLJUČAK* ..... *33*

*LITERATURA* ..... *34*

*SAŽETAK* ..... *35*

*ABSTRACT* ..... *35*

*ŽIVOTOPIS* ..... *36*

# 1. UVOD

U modernom razvoju softvera, testiranje je postala vrlo značajna aktivnost. Greška na programu koji je u produkciji financijski može prouzročiti jako veliku štetu. Stoga, u razvoju je potreban proces koji će spriječiti nastanak takvih slučajeva i osigurati isporuku kvalitetnog i pouzdanog softvera.

Od početka razvoja do gotovog softvera, ali također i za vrijeme održavanja, neki testovi ponavljaju se više puta. Takvi testovi spadaju u skupinu regresijskih testova. Testovi u skupini regresijskih automatiziraju se radi smanjenja cijene testiranja. Automatizirati testove značilo bi izradu novog softvera koji će izvršavati testiranje i uspoređivati stvarne rezultate sa predviđenim rezultatima.

Sustav automatskih testova služit će za:

- Slanje predodređenih zahtjeva serveru (REST API web servisu)
- Provjeru i validaciju odgovora servera (engl. *Response)*
- Izradu izvješća o testnim slučajevima
- Testiranje naredbama iz terminala

Za razumijevanje automatskih testova potrebno je poznavanje REST API web servisa. Web servis s funkcionalnošću društvene mreže bit će kreiran u svrhu ovog završnog rada, a na njega će automatski testovi slati zahtjeve i uspoređivati rezultate sa predviđenim rezultatima. Cilj završnog rada je pokazati stečena znanja iz područja programskog inženjerstva, web programiranja i objektno orijentiranog programiranja. Tehnologije i jezici korišteni za izradu automatskih testova su: JAVA programski jezik, REST (engl. *Respresentational state transfer*) koncept web servisa, JSON (engl. *JavaScript Object Notation*). Sami testovi implementiraju se na TestNG platformi na kojoj se koriste i *RestAssured* kao biblioteka za slanje zahtjeva i *SureFire* Report kao biblioteka za Generiranje izvješća.

# 2. PREGLED KORIŠTENIH TEHNOLOGIJA

## 2.1. Java programski jezik

Java je objektno orijentirani programski jezik koji se prvi put pojavljuje 1995 godine, a zasnovan je na programskom jeziku C++. Java sadrži primitivne tipove (*Integer, Boolean, Float, Char*), a dizajnirana je tako da nije ovisna o operacijskom sustavu na kojem se koristi, tj. može se koristiti na bilo kojem operacijskom sustavu koji podržava Javu. Upravo ta svestranost Javu čini najrasprostranjenijim programskim jezikom. Sama sintaksa Jave je slična C++, ali ima puno manje objekata niže razine (engl. *low-level*) što ju čini jednostavnijom u tom smislu. Pet primarnih ciljeva kojih su se držali Java kreatori [1]:

- Jednostavan objektno orijentirani programski jezik
- Robustan i siguran
- Neutralan u arhitekturi i prenosiv
- Izvršava se s visokim performansama
- Dinamičnost

JVM (engl. *Java Virtual Machine*) sustav je koji rješava problem „prenosivosti“ Java programskog jezika. Kada kompajler završi sa procesuiranjem Java koda, on umjesto izvršnog koda generira *Bytecode*. *Bytecode* je set instrukcija koji se može izvršavati na bilo kojem operacijskom sustavu na kojem postoji JVM [2]. Java dolazi skupa sa velikim brojem ugrađenih API-a (engl. *application programming interface*) koji sa sobom donose puno već gotovih funkcionalnosti. Gotovi programi olakšavaju programeru posao i štede mu vrijeme, što je još jedan od razloga zašto je Java zauzela visoku poziciju među programskim jezicima.

## 2.2. JSON

JSON je format za razmjenu podataka. Kako bi komunikacija između servera i klijenta bila uspješna, potreban je format koji će predstavljati podatke i biti lak za raščlanjivanje podataka. Iako je potpuno neovisan o programskom jeziku, nastao je iz jezika Java Script 1999. godine. Ovaj tekstualni format idealan je za razmjenu podataka jer koristi konvencije iz programskih jezika, neki od jezika su (*C, C++, Java, Java Script, Perl, Python,* itd.) [3].

Podatke koje prikazuje JSON format će poslagati hijerarhijski po objektima i objektnim varijablama. Kao što je vidljivo iz programskog koda 2.1, prikazan je objekt korisnika u JSON formatu, a u objektu se nalaze različite varijable (elektronička pošta, korisničko ime, pratitelji,

itd.). Ovaj format je nastao je kao „lakša“ alternativa XML formatu koji svojom robusnošću daje nešto lošije rezultate samog raščlanjivanja i procesuiranja teksta u programske varijable.

```
{
        "firstName": "John",
        "lastName": "Smith",
        "isAlive": true,
        "age": 27,
        "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": "10021-3100"
        }
}
```

***Programski kod 2.1*** *– Primjer Json tekstualnog formata*

Osnovne karakteristike JSON formata:

- Minimalna veličina poruke
- Podržava objekte, polja i varijable po konvencijama svih programskih jezika
- Čitljiv ljudima
- Brz za raščlanjivanje zbog strukture poruke

## 2.3. REST komunikacijski protokol

REST protokol je set pravila koja definiraju način komunikacije između klijenta i servera tj. definira na koji će način programer implementirati primanje zahtjeva na serveru, kao i slanje zahtjeva sa klijentskog uređaja. Neka od glavnih pravila kojih se pridržava REST koncept su:

- Predmemorija servera ne koristi se za čuvanje stanja podataka
- Uniformno sučelje
- Klijent - server odnos u komunikaciji
- Korištenje isključivo HTTP metoda

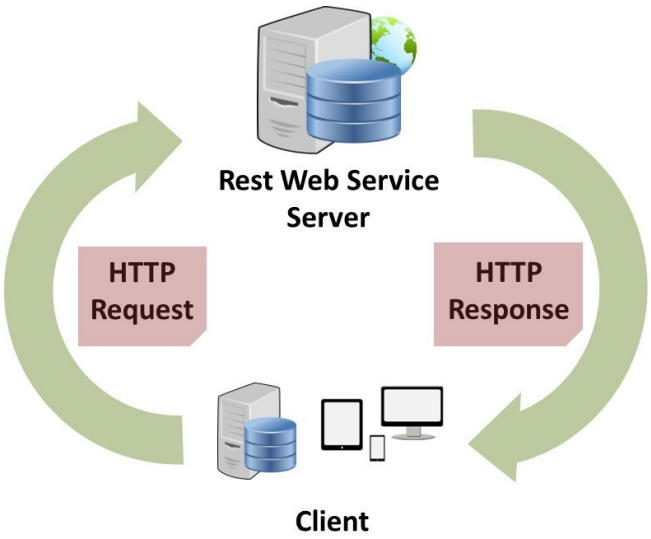
REST je nastao kao alternativa za robusniji SOAP (engl. *Simple Object Access Protocol*) zbog svoje manje propusnosti. REST API web servis koristi komunikacijske metode HTTP protokola.

Kako bi klijent dobio podatke sa servera to će učiniti pozivanjem GET metode itd. Najkorištenije HTTP metode su [4]:

- GET – koristi se za dohvaćanje podataka sa servera
- POST – koristi se za kreiranje podataka na serveru
- PUT – koristi se za ispravljanje podataka na server
- DELETE – koristi se za brisanje podataka sa servera

Još jedna od prednosti REST protokola je što možemo birati između korištenja više tekstualnih formata za komunikaciju. Tako da će, ovisno o potrebama aplikacije korisnik moći birati između JSON ili XML formata, dok SOAP dopušta korištenje samo XML formata. REST trenutno koriste neke od najvećih kompanija kao što su Amazon, Google, LinkedIn i Twitter. Pri odgovoru servera (engl. *Response*) REST API vraća poruku uz Standardni HTTP kod statusa. Načinom REST komunikacije koriste se mnogi serveri, a shemu sustava komunikacije servera i klijentskog uređaja prikazuje Slika 2.2. Najpoznatiji status kodovi su:

- 2XX – zahtjev je uspješno primljen i procesuiran
- 4XX – zahtjev je defektan, nije dobro strukturiran i ne može se ispuniti
- 5XX – server nije u mogućnosti ispuniti validan zahtjev




***Slika 2.2.*** *– Izgled komunikacije između dva sustava*

## 2.4. Platforma za testiranje

*TestNG* (engl. *Test New Generation*) je platforma koja služi za testiranje u Java programskom jeziku. Tvorac ove platforme bio je inspiriran platformom *JUnit* koju još uvijek koriste mnogi Java

programeri, ali *TestNG* teži ka pokrivanju više kategorija testnih scenarija kao što su funkcionalno testiranje i integracijsko testiranje, a uz to posjeduje i niz funkcionalnosti koje olakšavaju svakodnevno pisanje automatskih testova. Neke od glavnih značajki *TestNG* platfome su:

- Podrška anotacija u kodu
- Testiranje na temelju podataka (engl. *data-driven testing*)
- Mogućnost strukturiranja i prioritizacije Testnih scenarija
- Testovi se pokreću na više niti (engl. *threads*).

Za automatiziranje testova na velikim projektima, mogućnost platforme da se testni scenariji grupiraju jako je bitna. Ta se opcija na *TestNG* platformi ostvaruje pomoću paketa za testiranje (engl. *Test Suite*). Pakete za testiranje predstavljaju XML datoteke sa prvom oznakom <*suite*>, koji u nastavku sadržava više <*test*> oznaka, a svaka <*test*> oznaka može sadržavati više <*classes*>, <*packages*> i <*group*> oznaka koje predstavljaju jednu ili više testnih klasa u koje su implementirani automatski testovi. Ovim načinom mogu se posebno pokrenuti testovi iz skupine regresijskih testova, funkcionalni testovi i sl. Izgled paketa za testiranje prikazan je u programskom kodu 2.3.

```
<suite name="fultests">
  <test name="frontend">
    <classes>
      <class name="com.javatest.testng.TestUI" />
    </classes>
  </test>
  <test name="backend">
    <classes>
      <class name="com.javatest.testng.TestSecurity" />
      <class name="com.javatest.testng.TestDataBase" />
    </classes>
  </test>
</suite>
```

***Programski kod 2.3.*** *– Paket za testiranje*

Još jedan detalj važan za razumijevanje *TestNG* platforme su anotacije. One se pišu ispred klase sa znakom @ i služe za identifikaciju metode tj. označavaju čemu metoda služi i kad će se pokrenuti. Anotacije su uvedene u Java programski jezik u petoj inačici JDK (engl. *Java*

*Development Kit*) i njihova funkcionalnost je iskorištena pri izradi *TestNG* platforme, pa pri kreiranju neke od metoda možemo jednostavno označiti kada će se ona pokrenuti. Neke od *TestNG* anotacija su [5]:

- *@BeforeSuite* – označena metoda će se pokrenuti jednom prije pokretanja testova u paketu za testiranje
- *@AfterClass* – označena metoda će se pokrenuti nakon završetka svih testova u toj klasi
- *@DataProvider* – označena metoda će pribaviti podatke za neku testnu metodu. Ova metoda vraća objekt čije se vrijednosti pripisuju parametrima testne metode.
- *@Test* – označava metodu kao testni scenarij

Pokretanje automatskih testova na ovoj platformi može se inicirati na više načina, npr. testove možemo pokrenuti u integriranom *Eclipse* i *IntelliJ* programskom okruženju, kao i direktno iz terminala.

## 2.5. Java Biblioteka za slanje i ovjeru zahtjeva

Rest Assured Java je biblioteka koja olakšava slanje i validaciju HTTP zahtjeva i odziva servera. Ova biblioteka lako se integrira na bilo koju testnu platformu u javi, kao što su *JUnit* i *TestNG*. Sa Rest Assured bibliotekom moguć je poziv bilo koje *HTTP* metode ali također i validacija onoga što nam server vraća, a to su kod statusa i podatci u tijelu odziva servera. Ovu biblioteku moguće je uključiti u projekt pomoću *maven* (programski kod 2.4. [6]) ili *gradle* alata za izgradnju projekta.

```
<dependency>
  <groupId>io.rest-assured</groupId>
  <artifactId>rest-assured</artifactId>
  <version>3.0.6</version>
  <scope>test</scope>
</dependency>
```

***Programski kod 2.4.*** *– Maven*

Svaki test pisan ovom bibliotekom počinje metodom *given*() nakon koje pišemo postavke zahtjeva kao što su zaglavlja, autorizacijski ključevi i sl. Nakon postavki slijedi metoda *when*() u kojoj definiramo *HTTP* metodu, rutu na koju šaljemo zahtjev i tijelo zahtjeva. Na završetak slijeda

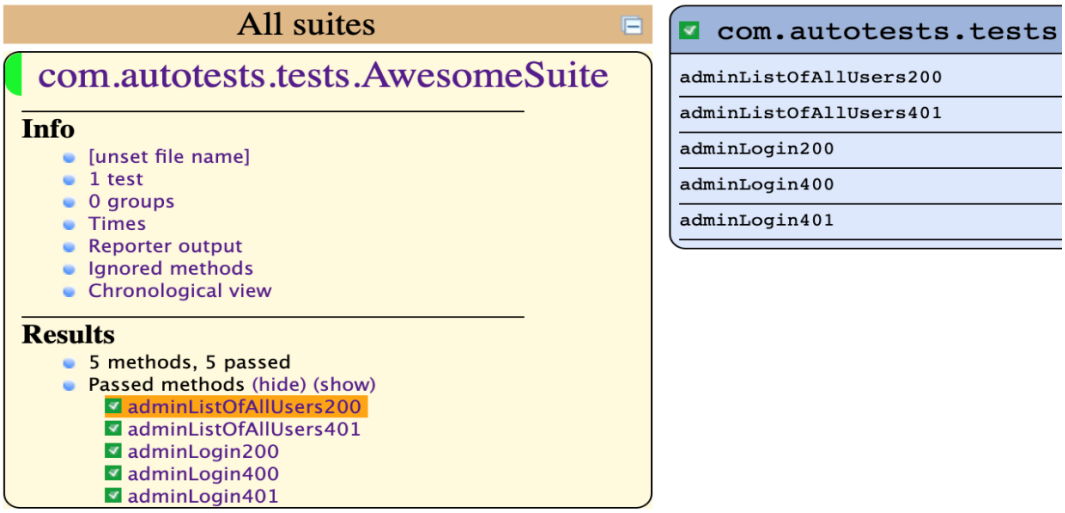
serijskih poziva metoda dodajemo metodu *then*() nakon koje postavljamo metode koje provjeravaju dobivene podatke kao što su kod statusa i tijelo odziva servera. Programski kod 2.5 prikazuje klasični testni scenarij pisan bibliotekom Rest Assured.

```
given()
        .contentType(ContentType.JSON)
        .pathParam("id", "someParam")
.when()
        .get("/examplepath/{id}")
.then()
        .statusCode(200)
        .body("firstName", equalTo("John"))
        .body("lastName", equalTo("Doe"));
```

***Programski kod 2.5.*** *– Primjer Testnog Scenarija*

## 2.6. Java biblioteka za generiranje izvješća

*Surefire* report Java je biblioteka za generiranje testnih izvješća nakon izvršenog testiranja. Po završetku testiranja ova biblioteka će napraviti više izvješća za više različitih platformi npr. izvješće za elektroničku poštu i web izvješće (Slika 2.6). Svako izvješće moguće je posebno urediti, jer *Surefire report* biblioteka generira *html* i *css* datoteke.




***Slika 2.6****. – Primjer web izvješća*

Za uključivanje ove biblioteke u projekt prvo je potrebno definirati <*plugin*> oznaku u *pom.xml* datoteci *maven* alata za izgradnju projekta (Programski kod 2.7). Zatim je potrebno i kreirati Java klasu „*Listener*“ koja ima za zadatak pratiti pokretanje testova i pamćenje njihovih

rezultata. U <*plugin*> oznaci *pom.xml* datoteke također je potrebno navesti paket za testiranje za koji radimo izvješća[7].

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-surefire-plugin</artifactId>
   <version>2.20.1</version>
   <configuration>
      <suiteXmlFiles>
         <suiteXmlFile>src/suites/testsuite.xml</suiteXmlFile>
      </suiteXmlFiles>
      <properties>
         <!--Set usedefaultlisteners to FALSE to start using custom listener-->
         <property>
            <name>usedefaultlisteners</name>
            <value>false</value>
         </property>
         <property>
            <name>listener</name>
            <value>
               com.rest.autotests.utils.reporting.MyReporterListener
            </value>
         </property>
      </properties>
   </configuration>
</plugin>
```

***Programski kod 2.7.*** *– Primjer pozivanja surefire report biblioteke u projekt*

# 3. PLANIRANJE PRISTUPA I REALIZACIJE AUTOMATSKIH TESTOVA

## 3.1. Pogled na funkcionalnosti REST web servisa

Web servis koji će biti podvrgnut automatskim testovima platforma je na koju se korisnici mogu registrirati i objavljivati fotografije. Posjeduje rute za ovjeru korisnika tj. rutu za registraciju, rutu za prijavu i rutu za promjenu lozinke. Zbog zaštite korisnikovog profila, prilikom uspješne ovjere korisnika server generira pristupni žeton (engl. *access token*)[8] koji se u svakom slijedećem zahtjevu mora se staviti u zaglavlje zahtjeva. Na ovaj način korisnik će svoju elektroničku poštu i lozinku unijeti samo jedanput, a pristupni žeton služit će kao njegova potvrda o autentičnosti.

### 3.1.1. RUTA ZA REGISTRACIJU

- URL: https://imgybackend.herokuapp.com/auth/sign-up
- METODA: POST
- ZAGLAVLJA – *Authorization-Static* – statični žeton („*imgybackend*“)
- TIJELO ZAHTJEVA (Format *FORM-DATA*) – programski kod 3.1

```
{
   "email": "example@email.com",
   "password": "examplePassword",
   "username": "exampleUsername"
}
```

***Programski kod 3.1*** *– Primjer Tijela registracije u „data“ polju*

Ako je zahtjev validan, uspješan odgovor servera sa statusnim kodom 200 u tijelu odgovora servera imat će pristupni žeton, ID korisnika i poruku o uspješnoj registraciji (Programski kod 3.2). U slučaju defektnog zahtjeva, dobit ćemo jedan od slijedećih odgovora:

- ERROR 409 – Korisnik već postoji
- ERROR 400 – Defektan zahtjev
- ERROR 500 – Problem sa serverom

```
{
   "token":
"eUzI1NiJ9.eyJpZCI6IjVkMTdkODAxMzRiZjQxMDdavkK9nP6Spaso61zDbZ--4",
   "_id": "5d17d80134bf4100048ffdd4",
   "message": "Successfully registered! Please proceed with login."
```

***Programski kod 3.2.*** *– Uspješan odziv servera*

### 3.1.2. RUTA ZA PRIJAVU

- URL: https://imgybackend.herokuapp.com/auth/sign-in
- METODA: POST
- ZAGLAVLJA – *Authorization-Static* – statični žeton („*imgybackend*“)
- TIJELO ZAHTJEVA (Format *RAW*) – elektronička pošta i lozinka

Slično kao i na ruti za registraciju, uspješan odgovor servera sa statusnim kodom 200 u tijelu odgovora servera imat će pristupni žeton i ID korisnika, a u slučaju defektnog zahtjeva predstavit će se jedan od slijedećih odgovora:

- ERROR 400 – Defektan zahtjev
- ERROR 401 – Loša autorizacija
- ERROR 409 – Neispravna lozinka

### 3.1.3. RUTA ZA PROMJENU LOZINKE

- URL: https://imgybackend.herokuapp.com/auth/change-password
- METODA: POST
- ZAGLAVLJA – *Authorization* – pristupni žeton
- TIJELO ZAHTJEVA (Format *RAW*) – programski kod 3.3

```
{
   "oldPassword": "example",
   "newPassword": "newExample",
}
```

***Programski kod 3.3***

Kao što je vidljivo u specifikacijama u ovom zahtjevu potrebno je u zaglavlju poslati pristupni žeton, procesuiranjem pristupnog žetona na serveru zna se o kojem se korisničkom profilu radi pa

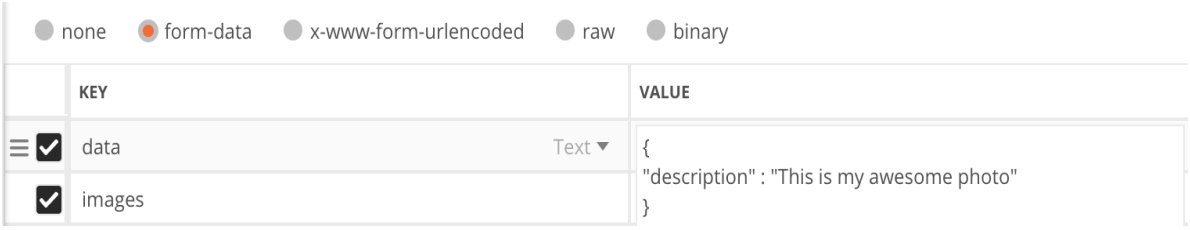
je u tijelu zahtjeva potrebno poslati samo novu i staru lozinku. U uspješnom zahtjevu server vraća poruku o uspješnoj promjeni lozinke, a zahtjevi sa greškom imaju tri slučaja:

- ERROR 400 – Defektan zahtjev
- ERROR 401 – Pošiljatelj zahtjeva nije autoriziran
- ERROR 409 – Lozinke se ne poklapaju
- ERROR 500 – Problem sa serverom

Nakon ruta za autentikaciju slijede rute za dohvaćanje fotografija kao i objavljivanje fotografija. Fotografije kao objekti imaju svoj opis, ali drugi korisnici ih mogu označiti oznakom „sviđa mi se“ (engl. *like*). Kada korisnik objavi fotografiju ona je dostupna ostalim korisnicima na ruti za dohvaćanje svih fotografija.

### 3.1.4. RUTA ZA OBJAVU FOTOGRAFIJE

- URL: https://imgybackend.herokuapp.com/feed
- METODA: POST
- ZAGLAVLJA – *Authorization* – pristupni žeton
- TIJELO ZAHTJEVA (Format *FORM-DATA*) – slika 3.4.




***Slika 3.4.*** *– Primjer tijela zahtjeva*

Kao što je vidljivo na slici Objava fotografije 3.4, zahtjev se sastoji od dva polja forme. Prvo polje naziva se „data“ i u njega se piše samo jedan atribut a to je opis fotografije odnosno „*description*“. Sadržaj polja „data“ treba se poslati u JSON tekstualnom formatu. Drugo polje služi za učitavanje slike koja može biti u *.jpg* ili *.png* formatu, a naziva se „*images*“. Ako je slika sa opisom uspješno pohranjena na server on će odgovoriti sa statusnim kodom 200 i u tijelu odziva bit će ispisana poruka o uspješnom kreiranju nove fotografije. U suprotnom server će vratiti jednu od slijedećih poruka:

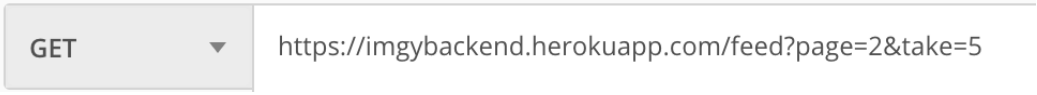
- ERROR 400 – Defektan zahtjev
- ERROR 401 – Pošiljatelj zahtjeva nije autoriziran
- ERROR 415 – Pogrešan format slike
- ERROR 500 – Problem sa serverom

Sve dosad objašnjene rute koriste HTTP POST metodu jer predaju forme serveru koje on dalje procesuira. Slijedeća ruta koristi HTTP GET metodu, a služi za dohvaćanje svih fotografija koje su korisnici objavili. Kod HTTP GET metode serveru se ne šalje tijelo zahtjeva, već se unose neobvezni URL (engl. *Uniform Resource Locator*) parametri direktno u URL zahtjeva.

### 3.1.5. RUTA ZA DOHVAĆANJE SVIH FOTOGRAFIJA

- URL: https://imgybackend.herokuapp.com/feed
- METODA: GET
- ZAGLAVLJA – *Authorization* – pristupni žeton
- URL PARAMETRI – „*page*“ – stranica, „*take*“ – količina fotografija po stranici

Zbog prevencije problema koji može nastati učitavanjem predugačke liste fotografija, ovoj ruti je dodana funkcionalnost paginacije. Prosljeđivanjem URL parametara „*page*“ i „*take*“ koji predstavljaju stranicu i količinu artikala po stranici moguće je odabrati optimalan broj ovisno o platformi koja šalje ovaj zahtjev. Tako će se za npr. drugih pet artikala poslati zahtjev sa parametrima „*page*=2“ i „*take*=5“ (slika 3.5).




***Slika 3.5.*** *– izgled zahtjeva*

Ako je korisnički pristupni žeton valjan, server će poslati listu artikala koji sadrže fotografije i njihove opise. Svaki artikl sastoji se od više atributa kao što su ID fotografije, opis, broj oznaka sviđa mi se itd. (programski kod 3.6)

```
{
        "_id": "5d14f3021243ea0004a8cae0",
        "description": "This is my awesome photo",
        "user": {
          "_id": "5d14bf6130527a0004f31c31",
          "username": "example"
        },
        "likesCount": 1,
        "createdAt": 1561654018228,
        "images": ["https://res.cloudinary.com/djul74lwz/image"],
        "imageIds": ["ixrm7kme9b9hovlf8mnn"],
        "likes": ["5c9e9e7c918d9f00045831fe"]
}
```

***Programski kod 3.6.*** *– Izgled jednog artikla u listi*

### 3.1.6. RUTA ZA STAVLJANJE OZNAKE „SVIĐA MI SE“

- URL: https://imgybackend.herokuapp.com/feed/item/like
- METODA: PUT
- ZAGLAVLJA – Authorization – pristupni žeton
- URL PARAMETRI – „*feedItemId*“ – ID fotografije koju označavamo

Za ovu rutu potrebno je posjedovati ID fotografije koju želimo označiti i pristupni žeton. Funkcionalnost ove rute radi na principu sklopke (engl. *toggle*). To znači da ako smo fotografiju prethodno označili, ponovno pozivanje ovog zahtjeva uklonit će korisnika iz liste ljudi koji su označili fotografiju. Ako je ovaj zahtjev uspješno ispunjen server će u odgovoru vratiti ažuriran artikl koji smo označili, a u suprotnom server će vratiti jedan od slijedećih statusnih kodova:

- ERROR 400 – Defektan zahtjev
- ERROR 401 – Pošiljatelj zahtjeva nije autoriziran
- ERROR 409 – Artikl nije pronađen
- ERROR 500 – Problem sa serverom

Zadnja preostala grupa ruta su korisničke rute, a njima možemo dohvatiti svoj profil, ili tuđi profil. Objekt korisnika sadrži profilnu sliku, korisničko ime, pratitelje itd. Za učitavanje odnosno zamjenu profilne slike služi ruta za uređenje profila. A tu je još i ruta koja je implementirana za funkcionalnost praćenja (engl. *follow*) drugog profila.

### 3.1.7. RUTA ZA DOHVAĆANJE SVOG PROFILA

- URL: https://imgybackend.herokuapp.com/user/me
- METODA: GET
- ZAGLAVLJA – Authorization – pristupni žeton
- URL PARAMETRI – „*page*“ – stranica, „*take*“ – količina fotografija po stranici

Za uspješno procesuiran zahtjev ove rute server će vratiti objekt korisnika tj. atribute koje predstavljaju njegov profil. Skupa sa korisničkim profilom dolazi i lista svih fotografija koje je korisnik objavio pa zbog toga i ova ruta ima funkcionalnost već objašnjene paginacije.
Programski kod 3.7. prikazuje korisnički model koji je vraćen u tijelu uspješnog odziva servera. U slučaju neuspješnog procesuiranja zahtjeva moguće su dvije greške, a to su da korisnik nije autoriziran ili je došlo do pogreške na serveru.

```
{
        "_id": "5c9e9e7c918d9f00045831fe",
        "email": "john@doe.com",
        "username": "john@doe.com",
        "isRegularUser": true,
        "followersCount": 2,
        "followingCount": 1,
        "profileImage":
        "https://res.cloudinary.com/djul74lwz/image/upload/v1555584552/rddb9rccaafw
        iozv0oed.jpg",
        "socialPlatforms": [],
        "followers": [
        "5d0aa6cb777bcb0004567c14",
        "5d14bf6130527a0004f31c31"
        ],
        "following": [
        "5d0aa6cb777bcb0004567c14"
        ]
}
```


***Programski kod 3.7.*** *–Izgled korisničkog modela*

### 3.1.8. RUTA ZA DOHVAĆANJE TUĐEG PROFILA

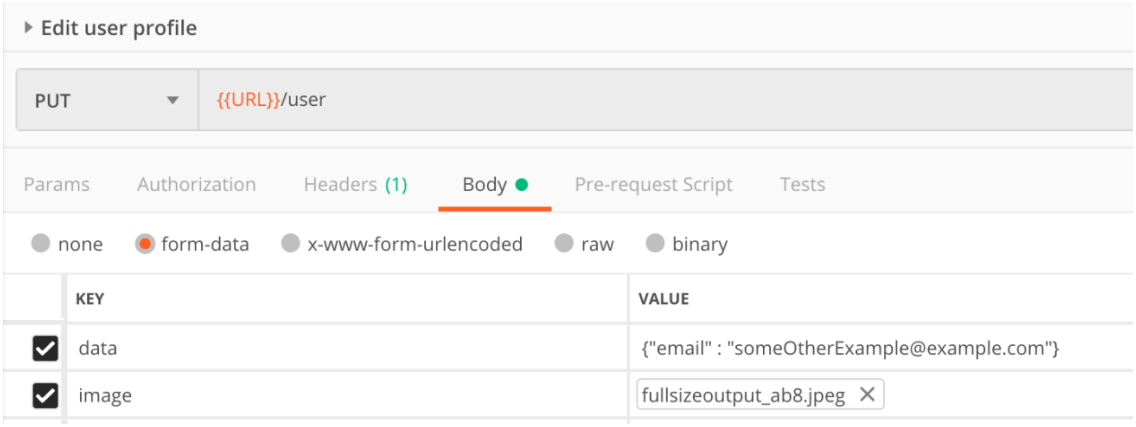
- URL: https://imgybackend.herokuapp.com/user
- METODA: GET
- ZAGLAVLJA – Authorization – pristupni žeton
- URL PARAMETRI – „*userId*“- ID traženog korisnika, „*page*“ – stranica. „*take*“ – količina fotografija po stranici

Uz parametre paginacije koji nisu obvezni, uz ovaj zahtjev potrebno je poslati i obvezni parametar „*userId*“ koji predstavlja ID korisnika kojeg tražimo. Odgovor servera razlikuje se od prethodno objašnjene rute u tome što se na ovoj ruti neće vratiti korisnikova adresa elektroničke pošte, koja je zaštićena i dostupna samo u okvirima korisnikovog pristupnog žetona.

### 3.1.9. RUTA ZA UREĐIVANJE PROFILA

- URL: https://imgybackend.herokuapp.com/user
- METODA: PUT
- ZAGLAVLJA – *Authorization* – pristupni žeton
- TIJELO ZAHTJEVA (Format *FORM-DATA*) – slika 3.8

Korisnik ovom rutom može mijenjati dva parametra na svom profilu a to su elektronička pošta i profilna slika. Tijelo zahtjeva moramo poslati u formatu forme sa dva polja, a to su „*data*“ polje u koje upisujemo elektroničku poštu u JSON tekstualnom formatu i „*image*“ polje u koje učitavamo sliku u *.jpg* ili *.png* formatu.




***Slika 3.8.*** *– Primjer zahtjeva za uređivanje profila*

Polja forme nisu obvezna i moguće je poslati samo jedno od njih. Za uspješno obavljen zahtjev server će vratiti ažuriran korisnikov model profila, a u slučaju neuspjeha postoje slijedeći statusni kodovi:

- ERROR 400 – Defektan zahtjev
- ERROR 401 – Pošiljatelj zahtjeva nije autoriziran
- ERROR 415 – Pogrešan format slike
- ERROR 500 – Problem sa serverom

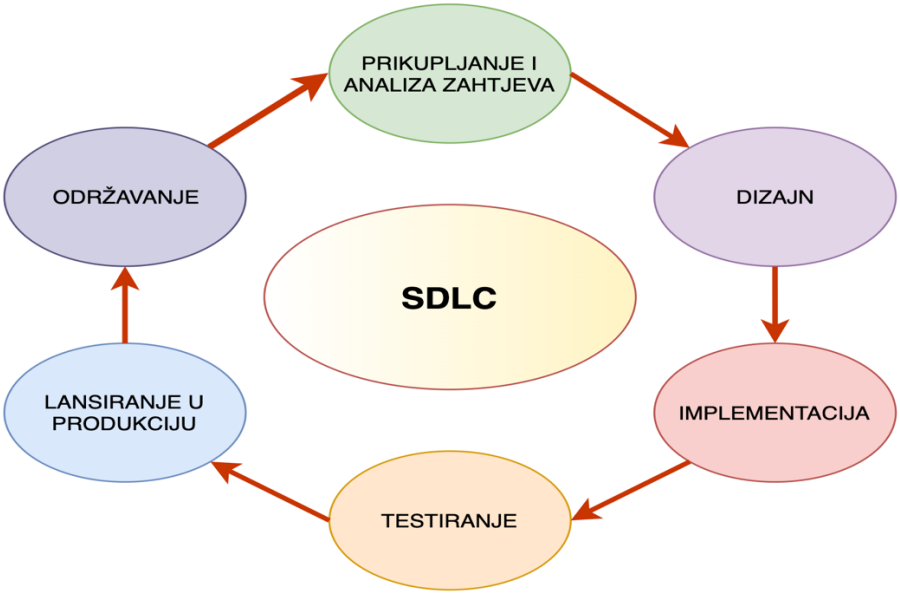
### 3.1.10. RUTA ZA PRAĆENJE PROFILA

- URL: https://imgybackend.herokuapp.com/user/follow
- METODA: PUT
- ZAGLAVLJA – *Authorization* – pristupni žeton
- URL PARAMETRI – „*userId*“- ID korisnika za praćenje

Ruta za praćenje korisnika posljednja je ruta ovog REST web servisa. Kao URL parametar potrebno je poslati ID korisnika koji će se pratiti. Funkcionalnost ove rute funkcionira kao sklopka To znači da ako se ovaj zahtjev primjeni na već praćenog korisnika, taj korisnik se prestaje pratiti.

## 3.2. Izrada testnih scenarija za automatizaciju

Pri generiranju nove verzije ovog web servisa sa proširenim funkcionalnostima cijeli web servis mora proći kroz fazu testiranja. Ponavljanjem ovog ciklusa (Slika 3.9) obujam testiranja sve se više povećava. Zbog uštede vremena i novca, testove za funkcionalnosti postojećih ruta automatizirati će se, a za to je potrebno napraviti strategiju tj. plan testiranja.




***Slika 3.9*** *SDLC(engl. „Software development life cycle“)*

### 3.2.1. TESTIRANJE RUTA ZA OVJERU KORISNIKA

Plan počinje tamo gdje korisnik ima prvi kontakt sa servisom a to su rute za ovjeru korisnika odnosno rute za registraciju, prijavu i promjenu lozinke. Za prvi testni slučaj poslužit će funkcionalnost rute za registraciju. Cilj je provjeriti hoće li za ispravno unesene podatke u zahtjevu server vratiti očekivan rezultat, a to je pristupni žeton. Isti slučaj treba provjeriti i na ruti za prijavu. Dakle, prva dva testna slučaja izgledat će ovako:

- TC-1 (engl. *Test Case*) – Registracija novog korisnika i provjera statusnog koda i pristupnog žetona
- TC-2 – Prijava Postojećeg korisnika i provjera statusnog koda i pristupnog žetona

Provjerit će se i neki statusni kodovi koji nisu uspješni. Prosljeđivanjem postojeće elektroničke pošte na rutu za registraciju može se provjeriti radi li Error-409 ispravno, a ruta za prijavu trebala bi vratiti Error-401 ako joj se u zahtjevu ne pošalje statični žeton.

- TC-3 – Provjera koda greške 409 na ruti za registraciju, slanjem elektroničke pošte već postojećeg korisnika
- TC-4 – Provjera koda greške 401 na ruti za prijavu, slanjem zahtjeva bez statičnog žetona

Također je potrebno poslati zahtjev na rutu za promjenu lozinke, a potom provjeriti je li nova lozinka u funkciji. Ovaj scenarij možemo raspisati u dva testna slučaja:

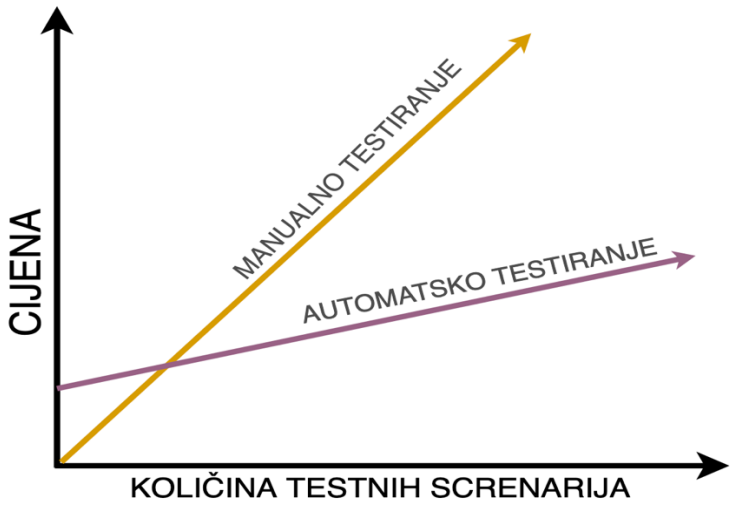
- TC-5 – Provjera statusnog koda 200 na ruti za promjenu lozinke
- TC-6 – Provjera rute za prijavu nakon uspješno promijenjene lozinke

### 3.2.2. TESTIRANJE RUTA ZA INTERAKCIJU SA FOTOGRAFIJAMA

Na rutu za postavljanje fotografije potrebno je poslati testnu sliku i opis pa zatim provjeriti je li slika uspješno učitana. Nakon toga ispitat će se funkcionalnost rute za dohvaćanje svih fotografija. Također provjeriti će se radi li ruta za stavljanje oznake „sviđa mi se“ pa nastaju slijedeći testni slučajevi:

- TC-7 – Provjera rute za postavljanje fotografije, šalju se važeći testni podatci i provjerava statusni kod 200
- TC-8 – Provjera rute za dohvaćanje svih fotografija za statusni kod 200
- TC-9 – Stavljanje oznake „sviđa mi se“ na neku od fotografija i provjera nalazi li se ID korisnika u polju *likes*[]

Ovim testovima pokrili smo osnovne funkcionalnosti ruta za interakciju s korisnicima. Najbitniji dio ovih testova je provjeriti radi li ruta za uspješno procesuiran zahtjev sa statusnim kodom 200. Razlog tome je to što je ruta napravljena da služi sa statusom 200 i prilikom procesuiranja zahtjeva s validnim informacijama pokreće se sloj programske logike za koju ta ruta i služi. To znači da je, pri nadograđivanju web servisa, najveća vjerojatnost da će se dogoditi greška upravo u uspješnom zahtjevu. Na slici 3.10 iz grafa je vidljivo kako sa većim brojem testnih slučajeva automatski testovi opravdavaju svoju egzistenciju po cijeni samog testiranja.




***Slika 3.10.*** *– Usporedba manualnog u odnosu na automatsko testiranje[9]*

### 3.2.3. TESTIRANJE KORISNIČKIH RUTA

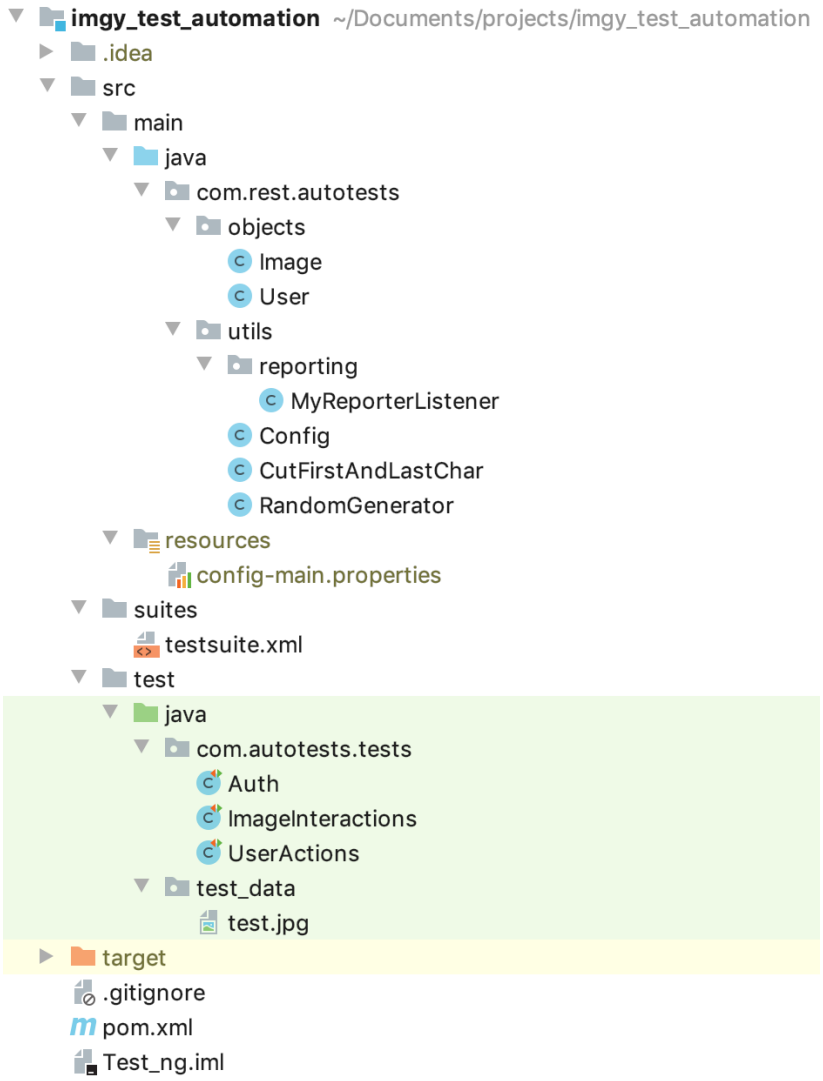
Posljednjih nekoliko testnih slučajeva odnosit će se na korisničke rute. Na ruti za dohvaćanje svog profila dovoljan je samo pristupni žeton kako bi zahtjev bio uspješno odrađen, a na ruti za dohvaćanje tuđeg profila prosljeđuje se ID korisnika. U oba slučaja provjeravat će se uspješno procesuiran zahtjev. Ruta za uređivanje profila krije dva testna slučaja, a to su promjena elektroničke pošte i učitavanje profilne slike. Seriju testova završit će testni slučaj za funkcionalnost praćenja koji će provjeriti je li ID korisnika u *followers*[] polju.

- TC-10 – Dohvaćanje svog profila, provjera odziva servera
- TC-11 – Dohvaćanje rute tuđeg profila, provjera odziva servera
- TC-12 – Uređivanje profila, promjena elektroničke pošte
- TC-13 – Uređivanje profila, učitavanje profilne slike
- TC-14 – Provjera rute za praćenje profila, ispitati nalazi li se korisnikov ID u polju *followers*[]

# 4. IMPLEMENTACIJA AUTOMATSKIH TESTOVA

## 4.1. Postavljanje strukture projekta

Struktura projekta ima dvije glavne grane. Prva grana pokriva definicije modela, funkcija i konfiguracije koje će se implementirati u automatske testove, a u druga grana služi za pisanje automatskih testova, a u njoj se pozivaju modeli i funkcije definirani u prvoj grani. Struktura projekta prikazana je na slici 4.1




***Slika 4.1.*** *– Izgled strukture projekta za automatske testove*

U *pom.xml* datoteci u projekt se uključuju sve vanjske biblioteke pomoću *maven* alata za gradnju projekta. U ovaj projekt uključene su sve biblioteke opisane u drugom poglavlju – pregled korištenih tehnologija. Primjer sintakse *pom.xml* datoteke moguće je vidjeti na slici 4.1.

## 4.2. Definiranje konstanti

Konstante se u projektu definiraju za sve vrijednosti koje se više puta ponavljaju a uvijek ostaju iste. U ovom projektu konstante se definiraju pomoću *Java properties* biblioteke koja predstavlja Java klasu. U toj klasi se konstante povlače iz zasebnog *config-main.properties* dokumenta (Programski kod 4.2). Ovakvim pristupom definiranja konstanti one se nikada ne moraju pisati ili mijenjati u kodu.

```
base.url = https://imgybackend.herokuapp.com
user.email = john@doe.com
user.password = johndoe
user.username = johndoe
static.token = imgybackend
```

***Programski kod 4.2.*** *– Config-main.properties datoteka*

Kao konstante u projektu navode se *url* web servisa kojeg testiramo, jedan već postojeći korisnik i statični žeton web servisa. Funkcionalnost *Config* klase koja povlači konstante vidljiva je na programskom kodu 4.3. Iz ove klase konstante se pozivaju u ostatak projekta.

```
static {
   try {
      PROPERTIES.load(Config.class.getResourceAsStream("/config-main.properties"));
      String configName = System.getProperty("configName");
       } catch (IOException e) {
      throw new IllegalStateException(e);
   }
}
public static final String USER_EMAIL = get("user.email");
public static final String USER_PASSWORD = get("user.password");
public static final String USER_USERNAME = get("user.username");
public static final String BASE_URL = get("base.url");
public static final String STATIC_TOKEN = get("static.token");
```

***Programski kod 4.3. -*** *Funkcionalnost Config klase*

## 4.3. Definiranje modela

U pogledu na funkcionalnost web servisa uočljivo je da se logika uglavnom kreće oko modela korisnika i modela slike. Radi lakšeg raščlanjivanja varijabli u odzivu servisa potrebno je kreirati ta dva modala. Model korisnika mora sadržavati varijable iste kao i model korisnika na odzivima servera, a to su elektronička pošta, lozinka, korisničko ime, autorizacijski žeton i ID. Model korisnika prikazan je na programskom kodu 4.4.

```
public class User {
    private String email;
    private String password;
    private String username;
    private String token;
    private String id;

    public User(String email, String password, String username) {
        this.email = email;
        this.password = password;
        this.username = username;
    }
public String getEmail(){return email;}
public void setEmail(String email){this.email=email;}
public String getPassword(){return password;}
public void setPassword(String password){this.password=password;}
public String getToken(){return token;}
public void setToken(String token){this.token=token;}
public String getId(){return id;}
public void setId(String id){this.id=id;}
public String getUsername(){return username;}
public void setUsername(String username){this.username=username;}}
```

***Programski kod 4.4.*** *– Izgled korisničkog modela u Java programskom jeziku*

Klasa sadrži konstruktor i *get* funkcije za pozivanje varijabli kao i *set* funkcije za postavljanje varijabli. Nakon korisničkog, kreira se model slike koji će sadržavati varijable identične kao i u odziv servera na rutama za slike. Model slike prikazan je na programskom kodu 4.5, a u njemu su definirana dva konstruktora zbog dvije različite inicijalizacije u automatskim testovima.

```
public class Image {
   private String id;
   private String description;
   private String userId;
   private int likesCount;
   private String imageUrl;
   private String path;

   public Image(String description, String path) {
      this.description = description;
      this.path = path;
   }

   public Image(String id, String description, String userId, int likesCount, String imageUrl) {
      this.id = id;
      this.description = description;
      this.userId = userId;
      this.likesCount = likesCount;
      this.imageUrl = imageUrl;
   }

public String getId(){return id;}
public void setId(String id){this.id=id;}
public String getDescription(){return description;}
public void setDescription(String description){this.description=description;}
public String getUserId(){return userId;}
public void setUserId(String userId){this.userId=userId;}
public int getLikesCount(){return likesCount;}
public void setLikesCount(int likesCount){this.likesCount=likesCount;}
public String getImageUrl(){return imageUrl;}
public void setImageUrl(String imageUrl){this.imageUrl=imageUrl;}
public String getPath(){return path;}
public void setPath(String path){this.path=path;}
```

***Programski kod 4.5..*** *– Izgled modela slike u Java programskom jeziku*

## 4.4. Definiranje automatskih testova

### 4.4.1. TESTOVI ZA OVJERU KORISNIKA

U prvoj testnoj klasi testirat će se rute za ovjeru korisnika, a testni scenariji izrađeni su u prošlom poglavlju. U začelju klase inicijaliziraju se dva objekta korisničkog modela klase *User* (Programski kod 4.5.), kao i dva pripadajuća objekta tipa *Json* koji će se slati u tijelu zahtjeva. Prvi korisnički objekt i pripadajući *json* sadržavaju već postojeću elektroničku poštu i lozinku, a u drugi korisnički objekt i pripadajući *json* kreiraju se novi nasumično odabrani elektronička pošta, lozinka i korisničko ime, te služi za registraciju korisnika.

```
public class Auth {

   static User user = new User(USER_EMAIL, USER_PASSWORD, USER_USERNAME);
   static User registrationUser = new User(randMail(), randPass(), randUsername());
   private static JSONObject loginBody = new JSONObject().put("email",
user.getEmail()).put("password", user.getPassword());
   private static JSONObject registrationBody = new JSONObject().
         put("email", registrationUser.getEmail()).
         put("password", registrationUser.getPassword()).
         put("username", registrationUser.getUsername());

   @BeforeTest
   public static void takeToken() {
      Response response = given().
            header("authorization-static", STATIC_TOKEN).
            header("Content-Type", "application/json").
            body(loginBody.toString()).
            post(BASE_URL + "/auth/sign-in");
      String token = response.then().extract().jsonPath().getString("token");
      String id = response.then().extract().jsonPath().getString("_id");
      user.setToken(token);
      user.setId(id);
   }
```

***Programski kod 4.5.*** *– Inicijalizacija objekata i metoda za uzimanje pristupnog žetona sa web servisa*

U nastavku na programskom kodu 4.5. definirana je metoda sa *TestNG* anotacijom *@BeforeTest*, koja se izvršava prije svih testova. Ova metoda sa bibliotekom *rest assured* šalje zahtjev na rutu za prijavu i iz odziva razlaže i sprema pristupni žeton, koji se sa set funkcijom sprema u prvi objekt korisnika koji je kreiran sa podatcima iz konstanti.

U nastavku prve testne klase definiraju se testovi TC-2 za provjeru rute za prijavu sa statusnim kodom 200 i TC-4 za provjeru za odziv 401 (Programski kod 4.6.). Metode počinju definiranjem *Response* objekta *TestNG* klase. U objektima se definira zaglavlje tipa *json* i statični žeton za pristup web servisu. U testu za provjeru rute za prijavu kao tijelo zahtjeva šalje se već ranije definiran *json* sa elektroničkom poštom i lozinkom, a u drugom testu šaljemo identične podatke bez statičnog žetona. U prvom testu kao validaciju odziva servera navodimo status kod 200 i prisutnost korisničkog autorizacijskog žetona, a u drugom testu kao odziv servera očekujemo statusni kod 401.

```
@Test
public static void userLogin200() {
   Response response = given().
         header("Content-Type", "application/json").
         header("authorization-static", STATIC_TOKEN).
         body(loginBody.toString()).
         post(BASE_URL + "/auth/sign-in");
   response.then().statusCode(200);
   response.then().body("token", notNullValue());
}

@Test
public static void userLogin401() {
   Response response = given().
         header("Content-Type", "application/json").
         body(loginBody.toString()).
         post(BASE_URL + "/auth/sign-in");
   response.then().statusCode(401);
}
```

***Programski kod 4.6.*** *– Definiranje TC-2 i TC-4*

Na programskom kodu 4.7 prikazani su testovi TC-1 i TC-4 koji provjeravaju rutu za registraciju za statusne kodove 200 i 409. U prvom testu, definiraju se zaglavlja za formu i statični žeton, a kao što je vidljivo na slici, u polju „data“ šalje se tijelo registracijskog zahtjeva definirano na početku klase. Provjerava se ima li odziv servera statusni kod 200 i postoji li žeton u tijelu odziva. Nakon provjere u prvom testu se u objekt korisnika za registraciju sprema autorizacijski žeton i ID koji su razloženi iz odziva servera. U drugom testu šalje se identičan zahtjev sa podatcima već postojećeg korisnika i provjerava se ima li odziv servera statusni kod 409 koji predstavlja grešku zbog registracije već postojećeg korisnika.

```
@Test
public static void userRegistration200() {
   Response response = given().
         header("Content-Type", "multipart/form-data").
         header("authorization-static", STATIC_TOKEN).
         multiPart("data", registrationBody.toString()).
         post(BASE_URL + "/auth/sign-up");
   response.then().statusCode(200);
   response.then().body("token", notNullValue());
   registrationUser.setToken(response.then().extract().path("token").toString());
   registrationUser.setId(response.then().extract().path("_id").toString());
}

@Test
public static void userRegistration409() {
   Response response = given().
         header("Content-Type", "multipart/form-data").
         header("authorization-static", STATIC_TOKEN).
         multiPart("data", registrationBody.toString()).
         post(BASE_URL + "/auth/sign-up");
   response.then().statusCode(409);
}
```

***Programski kod 4.7.*** *– TC-2 i TC-4*

Posljednja dva testa odnose se na rutu za promjenu lozinke. U prvom testu u parametrima anotacije @Test navodi se da ovaj test ovisi o testu za registraciju korisnika. U testu se definira

json sa poljima stara lozinka i nova lozinka. Šalje se korisnički autorizacijski žeton i provjerava se statusni kod 200. Nakon odrađenog prvog testa pokreće se drugi test TC-6 u čijoj se anotaciji navodi da ovisi o testu promjene lozinke. Test TC-6 poslat će jednostavan zahtjev za prijavu nakon promjene lozinke i očekuje da se korisnik može prijaviti sa novom lozinkom. Dakle očekuje se statusni kod 200. Kod ova dva testa vidljiv je na programskom kodu 4.8.

```
@Test(dependsOnMethods = {"userRegistration200"})
   public static void changePassword200() {
      JSONObject changePasswordBody = new JSONObject().
            put("oldPassword", registrationUser.getPassword());
      registrationUser.setPassword(randPass());
      changePasswordBody.put("newPassword", registrationUser.getPassword());
      Response response = given().
            header("Content-Type", "application/json").
            header("Authorization", registrationUser.getToken()).
            body(changePasswordBody.toString()).
            post(BASE_URL + "/auth/change-password");
      response.then().statusCode(200);
   }

   @Test(dependsOnMethods = {"changePassword200"})
   public static void userLoginAfterPasswordChange200() {
      Response response = given().
            header("Content-Type", "application/json").
            header("Authorization-static", STATIC_TOKEN).
            body(loginBody.toString()).
            post(BASE_URL + "/auth/sign-in");
      response.then().statusCode(200);
   }
}
```

***Programski kod 4.8.*** *– TC-5 i TC-6*

### 4.4.2. TESTOVI ZA INTERAKCIJU S FOTOGRAFIJAMA

Druga testna klasa testirat će rute za interakciju s fotografijama. U začelju klase definira se objekt modela slike koji sadrži opis i sliku, odnosno put do direktorija u kojem se slika nalazi.

Nakon toga definira se *json* koje sadrži testni opis slike. Prvi test TC-7 provjerava rutu za postavljanje slike. U tijelu testne funkcije definira se zahtjev sa autorizacijskim žetonom u formatom tijela u zaglavlju. U prvom polju zahtjeva šalje se prethodno kreirani *json* sa opisom slike, a u drugom polje šalje se *jpg* file. Provjerava se je li statusni kod odziva 200 i iz tijela odziva *parsira* se ID i stavlja u objekt slike (Programski kod 4.9.).

```
public class ImageInteractions {

   static Image image = new Image("testDescription", System.getProperty("user.dir") +
"/src/test/java/test_data/test.jpg");
   private static JSONObject imageBody = new JSONObject().put("description",
image.getDescription());

   @Test
   public static void addFeedItem200() {
      Response response = given().
            header("Authorization", registrationUser.getToken()).
            header("Content-Type", "multipart/form-data").
            multiPart("data", imageBody.toString()).
            multiPart("images", new File(image.getPath()), "image/jpg").
            post(BASE_URL + "/feed");
      response.then().statusCode(200);
      image.setId(response.jsonPath().getString("newItem._id"));
   }
```

***Programski kod 4.9.*** *– TC-7*

U nastavku testne klase nalaze se dva testa TC-8 i TC-9 (Programski kod 4.10). U prvom testu koji provjerava rutu za dohvaćanje svih fotografija definira se zaglavlje sa statičnim žetonom i šalje se http *get* zahtjev na web server sa tim podatcima. Kada se taj zahtjev izvrši pregledava se ima li odziv statusni kod 200 i postoji li u tijelu odziva varijabla „*feedItemCount*“ koja služi kao broj trenutno objavljenih fotografija. U drugom zahtjevu sa korisničkim autorizacijskim žetonom šalje se zahtjev sa ID parametrom u ruti zahtjeva koji predstavlja *id* slike. Po gotovom zahtjevu pregledava se statusni kod 200 i sadržava li like polje u odzivu zahtjeva korisnički *id*.

```
@Test
    public static void getAllItems200() {
        Response response = given().
                header("Authorization-static", STATIC_TOKEN).
                get(BASE_URL + "/feed");
        response.then().statusCode(200);
        response.then().body("feedItemCount", notNullValue());
    }

    @Test
    public static void likeImage200() {
        Response response = given().
                header("Authorization", user.getToken()).
                put(BASE_URL + "/feed/item/like?feedItemId=" + image.getId());
        response.then().statusCode(200);
        List<Object> likes = response.jsonPath().getList("likes");
        assertTrue(likes.contains(user.getId()));
    }
}
```

***Programski kod 4.10.*** *– TC-8 i TC-9*

### 4.4.3. TESTOVI KORISNIČKIH RUTA

Posljednja testna klasa testirat će korisničke rute čiji su testni scenariji izrađeni u prethodnom poglavlju. Na početku klase definira se *json* s poljem elektroničke pošte koji će se kasnije koristiti u testu za promjenu iste. U prvom testu TC-10, koji testira rutu za dohvaćanje profila, definira se zahtjev s autorizacijskim žetonom korisnika koji se šalje *get* metodom na rutu. Provjerava se statusni kod 200 i postoji li varijabla „*foundUser*“ koja predstavlja objekt korisnika u odzivu servera.

Drugi test u ovoj klasi testira rutu za dohvaćanje tuđeg profila. Postavke zahtjeva identične su kao i u prvom zahtjevu, s iznimkom rute zahtjeva. Prve dvije funkcije prikazane su na programskom kodu 4.11.

```
public class UserActions {

   private static JSONObject changeMailBody = new JSONObject().put("email", randMail());

   @Test
   public static void getMyProfile200() {
      Response response = given().
            header("Authorization", user.getToken()).
            get(BASE_URL + "/user/me");
      response.then().statusCode(200);
      response.then().body("foundUser", notNullValue());
   }

   @Test
   public static void getOtherUserProfile200() {
      Response response = given().
            header("Authorization", user.getToken()).
            get(BASE_URL + "/user?userId=" + user.getId());
      response.then().statusCode(200);
      response.then().body("foundUser", notNullValue());
   }
```

***Programski kod 4.11.*** *– TC-10 i TC-11*

U posljednja tri testa pregledavat će se funkcionalnost ruta za promjenu elektroničke pošte, promjenu profilne slike i praćenja profila (Slika 4.12.). U prvom testu TC-12 definira se zahtjev s korisničkim autorizacijskim žetonom u zaglavlju i format zahtjeva *multipart*. U tijelu zahtjeva šalje se već ranije definiran *json* za promjenu maila i provjerava se statusni kod 200.
Postavke zaglavlja u drugom testu TC-13 identične su prvom, s iznimkom tijela zahtjeva. U tijelo zahtjeva u *multipart* formatu šalje se *jpg* slika u polju „*image*“. U nastavku funkcije provjerava se statusni kod 200.

Posljednji test provjerava funkcionalnost rute za praćenje korisnika. Nakon definiranja zaglavlja, podešavamo put metodu zahtjeva s korisničkim ID atributom koji stavljamo u *url* zahtjeva. Nakon izvršenog zahtjeva provjerava se statusni kod 200 i sadrži li polje *likes* u korisničkom profilu ID korisnika koji je poslao zahtjev.

```
    @Test
    public static void changeMail200() {
        Response response = given().
                header("Authorization", registrationUser.getToken()).
                header("Content-Type", "multipart/form-data").
                multiPart("data", changeMailBody.toString()).
                put(BASE_URL + "/user");
        response.then().statusCode(200);
    }

    @Test
    public static void changeProfileImage200() {
        Response response = given().
                header("Authorization", registrationUser.getToken()).
                header("Content-Type", "multipart/form-data").
                multiPart("image", new File(ImageInteractions.image.getPath()), "image/jpg").
                put(BASE_URL + "/user");
        response.then().statusCode(200);
    }

    @Test
    public static void follow200() {
        Response response = given().
                header("Authorization", registrationUser.getToken()).
                put(BASE_URL + "/user/follow?userId=" + user.getId());
        response.then().statusCode(200);
        List<Object> followers = response.jsonPath().getList("followedUser.followers");
        assertTrue(followers.contains(registrationUser.getId()));
    }
}
```

***Programski kod 4.12.*** *– TC-12 , TC-13 i TC-14*

## 4.5. Pokretanje testova i pregled testnog izvješća

Kreirana datoteka testsuite.xml (Programski kod 4.13) grupira tri testne klase u jedan testni paket. Ovaj testni paket pokreće se iz terminala pomoću *maven* alata za gradnju projekta. Naredba

za pokretanje ima slijedeću sintaksu: „*mvn clean test -DsuiteXmlFile=testsuite.xml*“. Nakon pokretanja ove naredbe svi testovi iz testnih klasa izvršavaju se logičkim redoslijedom.

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="RESTSuite" verbose="1">

  <test name="REST API Tests" enabled="true">
    <classes>
      <class name="com.autotests.tests.Auth"/>
      <class name="com.autotests.tests.ImageInteractions"/>
      <class name="com.autotests.tests.UserActions"/>
    </classes>
  </test>
```

***Programski kod 4.13.*** *– Testsuite.xml*

Nakon izvršenog paketa za testiranje testni izvještaj moguće je pogledati odmah u terminalu, ili u naknadno generiranim datotekama koje predstavljaju testni izvještaj. Na slici 4.15. prikazan je testni izvještaj namijenjen slanju na željenu elektroničku poštu. Iz testnog izvještaja je vidljivo da su svi testovi uspješno potvrdili valjanost funkcionalnosti web servisa, osim testa za provjeru profilne slike. Daljnjom provjerom datoteka za izvještaje koje se također kreiraju pri izvršenim testovima (Slika 4.14) može se uočiti da test za promjenu profilne slike kao odziv servera umjesto statusnog koda 200 šalje statusni kod 400.

```
-------------------------------------------------------------------------------
Test set: TestSuite
-------------------------------------------------------------------------------
Tests run: 14, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 32.697 s <<<
FAILURE! - in TestSuite
changeProfileImage200(com.autotests.tests.UserActions)  Time elapsed: 1.177 s  <<<
FAILURE!
java.lang.AssertionError:
1 expectation failed.
Expected status code <200> but was <400>.
        at
com.autotests.tests.UserActions.changeProfileImage200(UserActions.java:58)
```

***Slika 4.14.*** *– tekstualni testni izvještaj*

| Test | Methods Passed | Scenarios Passed | # skipped | # failed | Total Time | Included Groups |
|---|---|---|---|---|---|---|
| REST API Tests | 13 | 13 | 0 | 1 | 32.4 seconds | |

| Class | Method | Name | Time (s) |
|---|---|---|---|
| **REST API Tests — failed** | | | |
| com.autotests.tests.UserActions | **changeProfileImage200** | null | 1 |
| **REST API Tests — passed** | | | |
| com.autotests.tests.Auth | **userLogin200** | null | 1 |
| | **userLogin401** | null | 0 |
| | **userRegistration200** | null | 0 |
| | **userRegistration409** | null | 0 |
| | **changePassword200** | null | 1 |
| | **userLoginAfterPasswordChange200** | null | 0 |
| com.autotests.tests.ImageInteractions | **addFeedItem200** | null | 2 |
| | **getAllItems200** | null | 1 |
| | **likeImage200** | null | 0 |
| com.autotests.tests.UserActions | **changeMail200** | null | 1 |
| | **follow200** | null | 0 |
| | **getMyProfile200** | null | 0 |
| | **getOtherUserProfile200** | null | 0 |

***Slika 4.15.*** *– testni izvještaj za elektroničku poštu*

# 5. ZAKLJUČAK

Automatizirano testiranje web servisa kroz razvoj i održavanje servisa uvelike olakšava proces testiranja regresijskih testova. Projekt automatskih testova spaja se u sustav kontinuirane integracije web servisa i pokreće se na svako ažuriranje koda, a potom obavještava programera o uspješnosti testova. Kako se obujam projekta povećava, tako automatski testovi sve više opravdavaju svoju egzistenciju.

Izrada praktičnog dijela završnog rada zahtijevala je poznavanje REST protokola i web alata za izradu takvog sustava, kao i poznavanje, istraživanje i planiranje sustava automatskih testova. U dijelu planiranja projekta osmišljeni su testni scenariji, odnosno plan testiranja koji razrađuje način validacije svake rute web servisa. Za izradu projekta automatskih testova korišteno je *IntelliJ* programsko okruženje, a za izradu samog web servisa korišteno je *WebStorm* programsko okruženje sa programskim jezikom *JavaScript* i *mongoDB* ne-relacijskom bazom podataka. Za prikaz izgleda zahtjeva serveru korišten je program *Postman*.

Najveća prednost sustava automatiziranih testova jest brzina testiranja kao i činjenica da su automatski testovi daleko pouzdaniji u rezultatima jer nema greške ljudskog faktora u procesu testiranja. Ovakvim sustavom možemo povećati i broj ciklusa testiranja što rezultira povećanjem efikasnosti samog testnog procesa i kvalitetnijom finalnom verzijom proizvoda. No uz sve prednosti automatiziranog testiranja ne treba podcijeniti važnost i potrebu za manualnim testiranjem.

# LITERATURA

[1] Java documentation, About the Java technology, s interneta: http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html, siječanj 2017

[2] J. Bloch, Effective Java, Pearson Education, New Jersey, siječanj 2018

[3] Json official, s interneta: https://www.json.org , siječanj 2018

[4] Wikipedia, REST, s interneta: https://en.wikipedia.org/wiki/Representational_state_transfer, rujan 2018

[5] TestNG dokumentacija, s interneta: https://testng.org/doc/documentation-main.html , travanj 2004

[6] Rest assured dokumentacija, s interneta: https://github.com/rest-assured/rest-assured/wiki/GettingStarted, svibanj 2016

[7] Surefire report dokumentacija, s interneta: https://maven.apache.org/surefire/maven-surefire-plugin/usage.html, prosinac 2018

[8] Auth, Access tokens, s interneta: https://auth0.com/docs/api-auth/why-use-access-tokens-to-secure-apis, svibanj 2018

[9] R. Black, E. Van Veendall, D. Graham, Foundations of software testing, Thomson Press, Delhi, Prosinac 2007

[10] Slika 2.2 - REST, s interneta: https://medium.com/@sagar.mane006/understanding-rest-representational-state-transfer-85256b9424aa, lipanj 2017

# SAŽETAK

U završnom radu razvijen je sustav automatskih testova namijenjen brzom pronalasku grešaka u razvoju web servisa. Ovakav sustav implementira se u razvoj web aplikacije i pokreće se na svako ažuriranje web servisa sa ciljem održavanja kvalitete sustava. U teorijskom dijelu rada opisane su tehnologije korištene u razvoju ovakvog sustava, funkcionalnosti web servisa koji se testira te planiranje testnih scenarija. Praktični dio rada bavi se izradom sustava automatskih testova u programskom jeziku Java na testnoj platformi *TestNG*. Nakon izrade sustava objašnjeno je i kako iz terminala pokrenuti automatske testove te pogledati njihove rezultate. Za generiranje testnog izvješća korištena je *SureFire report* java biblioteka.



Ključne riječi: automatizirano testiranje, testni scenarij, web servis



# ABSTRACT

# AUTOMATED TESTING OF REST WEB SERVICES

In this bachelor thesis, an automated testing system for fast application development bugs and errors discovering was developed. This type of system can be embedded in a continuous integration process and invoked on every application update with the purpose of maintaining quality of the application. The theoretical part describes technologies used during development of the automated tests, functionality of the web service and test plan that is going to be implemented. Practical part of the thesis is dealing with the development of the test automation system with Java programming language based on TestNG platform. After the system implementation it is explained how to run automated tests and check their outcome. Java SureFire library is used for generating test reports.



Keywords: automated testing, test case, web service

# ŽIVOTOPIS

Luka Ušković rođen je 23. veljače 1997. godine u Đakovu. Od 2003. do 2010. pohađa Osnovnu školu Ivan Goran Kovačić u Đakovu. Godine 2010 upisuje Elektrotehničku i prometnu školu Osijek koju završava 2015. obranom završnog rada i polaganjem ispita državne mature. Godine 2015. upisuje Fakultet elektrotehnike, računarstva i informacijskih tehnologija na Sveučilištu Josipa Juraja Strossmayera u Osijeku na stručni studij informatike.

Luka Ušković