Programski alat za automatiziranu izradu testnih izvještaja poslužitelja

Budoš, Ivan

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni studij

PROGRAMSKI ALAT ZA AUTOMATIZIRANU IZRADU TESTNIH IZVJEŠTAJA POSLUŽITELJA

Diplomski rad

Ivan Budoš

Osijek, 2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za ocjenu diplomskog rada na sveučilišnom diplomskom studiju

Ocjena diplomskog rada na sveučilišnom diplomskom studiju

Ime i prezime pristupnika:	Ivan Budoš
Studij, smjer:	Sveučilišni diplomski studij Računarstvo
Mat. br. pristupnika, god.	D-1108R, 22.10.2020.
JMBAG:	0165072138
Mentor:	izv. prof. dr. sc. Josip Balen
Sumentor:	Matej Arlović, univ. mag. ing. comp.
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	prof. dr. sc. Krešimir Nenadić
Član Povjerenstva 1:	izv. prof. dr. sc. Josip Balen
Član Povjerenstva 2:	Davor Damjanović, univ. mag. ing. comp.
Naslov diplomskog rada:	Programski alat za automatiziranu izradu testnih izvještaja poslužitelja
Znanstvena grana diplomskog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	U teorijskom dijelu rada potrebno je proučiti i opisati postupke testiranja programskih sustava te opisati alate za testiranje aplikacijskog programskog sučelja poslužitelja. U praktičnom dijelu rada potrebno je u programskom jeziku C# koristeći Visual Studio razvojno okruženje razviti alat koji omogućuje stvaranje i slanje automatiziranih zahtjeva na poslužitelj slijedeći osmišljene testne scenarije te nakon izvršavanja kreira izvještaj testiranja. Nadalje, za potrebe demonstracije alata potrebno je osmisliti i implementirati testne scenarije koji testiraju funkcionalnosti stvarnog poslužitelja.
Datum ocjene pismenog dijela diplomskog rada od strane mentora:	03.09.2024.
Ocjena pismenog dijela diplomskog rada od strane mentora:	Izvrstan (5)
Datum obrane diplomskog rada:	18.09.2024.
Ocjena usmenog dijela diplomskog rada (obrane):	Izvrstan (5)
Ukupna ocjena diplomskog rada:	Izvrstan (5)
Datum potvrde mentora o predaji konačne verzije diplomskog rada čime je pristupnik završio sveučilišni diplomski studij:	18.09.2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O IZVORNOSTI RADA

Osijek, 18.09.2024.

Ime i prezime Pristupnika:	Ivan Budoš
Studij:	Sveučilišni diplomski studij Računarstvo
Mat. br. Pristupnika, godina upisa:	D-1108R, 22.10.2020.
Turnitin podudaranje [%]:	9

Ovom izjavom izjavljujem da je rad pod nazivom**: Programski alat za automatiziranu izradu testnih izvještaja poslužitelja**

izrađen pod vodstvom mentora izv. prof. dr. sc. Josip Balen

i sumentora Matej Arlović, univ. mag. ing. comp.

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis pristupnika:

SADRŽAJ

1. UVOD

U digitaliziranom dobu današnjice spoj poslužitelja (engl. *server*) i korisničkih aplikacija postala je ključna stavka mnogih organizacija koje se trude pružati ugodno korisničko iskustvo i olakšati si upravljanje poslovnim procesima. Povećanjem broja platformi i uređaja korištenih kao pristup raznim uslugama i informacijama, sve važnija postaje potreba za sigurnom i skladnom integracijom tih sustava. Mobilne aplikacije i web aplikacije povezane su s poslužiteljima koji im na zahtjev pružaju podatke i tako stvaraju zaokruženu cjelinu. Stvaranje te cjeline podrazumijeva spajanje različitih platformi, tehnologija i protokola koji omogućuju izmjenu podataka, usklađivanje funkcionalnosti i korisničkih sučelja. Integracija poslužitelja u takav sustav uključuje tehničke i sigurnosne izazove koje je potrebno pomno planirati. Zbog zahtjeva za sve bržim razvojem sustava i neprekidnim nadogradnjama postojećih sustava nerijetko se pojavljuju greške, što zbog lošeg planiranja, što zbog ljudskog faktora. Sustav s opsežnim setom testova robustan je na greške te samim time dobiva na pouzdanosti. Pouzdanost je posebice bitna kod sustava kod kojih greške mogu dovesti do značajno loših posljedica po korisnike sustava, kao npr. bankovni sustavi, online trgovine i ostali sustavi s novčanim transakcijama, avionski sustavi, sustavi navigacije, sigurnosni sustavi itd.

U sklopu diplomskog rada istražuju se metodologije testiranja programskih sustava i kada se one primjenjuju u različitim životnim ciklusima razvoja programskih sustava (engl. *systems development life cycle*, SDLC). Potom se fokus stavlja na moderne metode i alate za integracijsko API (engl. a*pplication programming interface*) testiranje poslužitelja. Cilj rada je razviti alat kojim se automatiziraju i izvode testovi kojima se provjerava uspješna komunikacija s API-jem, učinkovito pronalaze greške te po završetku izručuje automatski generiran izvještaj provedenog testa. Alat će omogućiti automatsko izvođenje različitih tijekova rada (engl. *workflow*) od interesa te brzu validaciju rezultata izvođenja svakog koraka. Automatizacija izvođenja testnih scenarija osigurava da se testovi izvode na identičan način svaki put kada su pokrenuti tj. garantira izostanak grešaka izazvanih ljudskim faktorom. U radu će biti opisani funkcionalnost stvarnog poslužitelja za kojeg će biti implementirani testni scenariji koristeći razvijeni alat. Nakon demonstracije dodavanja i uređivanja testova te njihovog pokretanja, prikazat će se dobiveni rezultati i izvještaj testiranja.

1.1. Zadatak diplomskog rada

U teorijskom dijelu rada potrebno je proučiti i opisati postupke testiranja programskih sustava te opisati alate za testiranje aplikacijskog programskog sučelja poslužitelja. U praktičnom dijelu rada potrebno je u programskom jeziku *C#* koristeći Visual Studio razvojno okruženje razviti alat koji omogućuje stvaranje i slanje automatiziranih zahtjeva na poslužitelj slijedeći osmišljene testne scenarije te nakon izvršavanja kreira izvještaj testiranja. Nadalje, za potrebe demonstracije alata potrebno je osmisliti i implementirati testne scenarije koji testiraju funkcionalnosti stvarnog poslužitelja.

2. TESTIRANJE PROGRAMSKE PODRŠKE

Svaki računalni program koji se razvija za određenu svrhu ima svoj očekivani način rada. To podrazumijeva ispunjavanje specifičnih zahtijeva uz održavanje zadovoljavajućeg standarda kvalitete. Zahtjevi i standardi kvalitete definiraju se u ranoj fazi razvoja, odnosno u fazi planiranja, dok se kontrola kvalitete (engl. *quality assurance*, QA) različito primjenjuje ovisno o modelu razvoja programske podrške. Cilj QA-a je uvjeriti se kako program koji se isporučuje korisnicima radi ispravno unutar granica dogovorene tolerancije tako što se otkrivaju i uklanjaju svi nedostaci.

Neki programski sustavi zahtijevaju pomnije testiranje od drugih. Kritičniji sustavi će biti testirani rigoroznije, detaljnije i češće, no opseg i način provođenja testiranja u pravilu ovisi o procjeni tima odgovornog za osiguranje kvalitete [1]. Članovi tima koji provodi QA mogu biti manualni testeri, QA inženjeri i programski inženjeri u testiranju (engl. s*oftware developer engineer in test*, SDET). Na višim pozicijama s većom razinom odgovornosti nalaze se i QA inženjer i SDET pošto je i jedna i druga uloga zadužena za planiranje testnih procesa, pripremanje testnih scenarija po zahtjevima korisnika te izradu opsežnih izvještaja testiranja. Pri opisu sposobnosti i dužnosti njihovih pozicija izvori se ne slažu. Neki izvori ne prepoznaju SDET-a kao zasebnu poziciju, nego sva zaduženja uz testiranje pripisuju QA inženjeru što sugerira kako je SDET noviji pojam koji se pojavio sve većom ulogom testiranja u planiranju i razvoju programskih rješenja [2]. Izvori koji spominju obje pozicije u pravilu se dijele na dva uvjerenja. Njihova temeljna razlika je u očekivanoj razini znanja programiranja te uključenosti u razvojnoj fazi planiranja i dizajniranja proizvoda.

Prema prvom uvjerenju za QA inženjera smatra se osoba koja razumije samo osnove programskih jezika i vješta je u korištenju alata za funkcionalno testiranje, no ne razvija automatizirane testove nego osmišljava manualne testove i delegira ih na testere unutar svog tima. Za razliku od QA inženjera, SDET aktivno sudjeluje u procesu planiranja razvoja programskog rješenja, ima pristup, razumije i pregledava kôd koji pišu razvojni programeri (engl. *developers*) te po njemu piše i osobno pokreće izvođenje automatizirane funkcionalne i nefunkcionalne testove [3].

Drugo uvjerenje pak tvrdi kako se SDET-i i QA inženjeri međusobno nadopunjuju. SDET osmišlja i automatizira testove te pruža podršku testovima u obliku nadogradne programskog okvira (engl. *framework*) i održavanja automatizacije, ali ih osobno ne pokreće. Tu nastupa QA inženjer koji po određenom rasporedu pokreće automatizirane testove i piše izvješća nakon svakog testnog ciklusa [4].

2.1. Testiranje unutar različitih modela razvoja programske podrške

Razvoj metodologija razvoja programske podrške (engl. *software*) odvijao se postepeno kako se uviđalo da stariji modeli preuzeti od industrijskih pogona nemaju učinkovit odgovor na neke bitne pojave koje utječu na proces razvoja programske podrške. Pojave koje su ponukale promjene modela razvoja programske podrške su: konstantne promjene i/ili dorade zahtjeva klijenata, pojava poteškoća pri razvoju i naglasak na što češće provođenje kontrole kvalitete. Klijenti će možda imati nepotpune vizije programskog rješenja u fazi planiranja te će promijeniti svoje zahtjeve kada vide kako ono radi, što dovodi do redizajna, ponovnog razvoja i testiranja. Također, dizajneri i razvojni programeri prilikom dizajna možda neće biti svjesni budućih poteškoća zbog kojih će se trebati odlučiti je li bolje revidirati dizajn ili ustrajati bez obzira na ograničenja. Kako bi se osigurala kvaliteta proizvoda kroz sve te promjene zahtjeva ili revidiranje dizajna, testiranje se počinje provoditi nakon svake inkrementalne promjene. Cilj novijih modela razvoja programske podrške je povećati fleksibilnost i agilnost procesa razvoja programske podrške uz održavanje željenog standarda kvalitete neprekidnim testiranjem (engl. *continuous testing*).

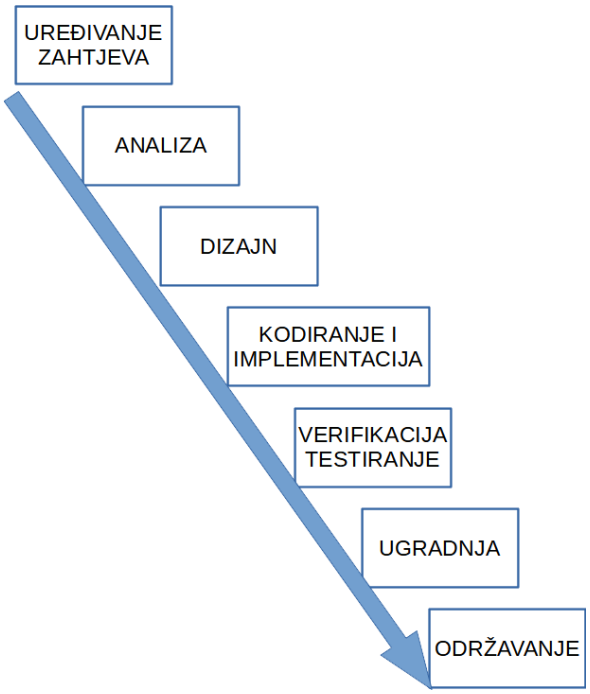
2.1.1. Testiranje u vodopadnom modelu razvoja programske podrške

Najjednostavniji pristup životnom ciklusu razvoja sustava programske podrške je vodopadni model (engl. *waterfall model*). On predstavlja sekvencijalan pristup koji koristi jednosmjeran slijed koraka projekta nalik na putanju vode koja teče preko ruba litice i postavlja različite krajnje točke ili ciljeve za svaku razvojnu fazu [5]. Jednom kada su dovršene, krajnje točke nije nemoguće ponovno dorađivati, ali je vrlo kompliciranо i vremenski zahtjevno. Razvojni stadiji vodopadnog modela su (Slika 2.1.):

1. **Uređivanje zahtjeva** – definiraju se planovi projekta bez definicije konkretnog procesa izvedbe. Određuju se krajnji rokovi i željene specifikacije sustava. Sve se dokumentira u dokument o zahtjevima korisnika.
2. **Analiza** – izrađuju se modeli proizvoda i poslovna logika za kontrolu proizvodnje analizom zahtjeva iz prošlog koraka. Procjenjuje se izvedivost po dostupnim financijskim i tehničkim sredstvima. Sve se dokumentira u dokument o analizi zahtjevima korisnika.
3. **Dizajn** – određuju se zahtjevi tehničkog dizajna kao što su programski jezik, hardver, izvori podataka, arhitektura. Stvara se dokument koji objedinjuje sve specifikacije dizajna.

4. **Kodiranje i implementacija** – razvija se izvorni kôd prateći dokument specifikacije dizajna iz prošlog koraka. Manji dijelovi sustava razvijaju se zasebno prije nego ih se zajedno integrira u gotov sustav.
5. **Verifikacija** (testiranje) – verificira se kako sustav ima željeno djelovanje. Identificiraju se poteškoće koje je potrebno popraviti. Pronalaženje grešaka može značiti potrebu za ispravcima, odnosno za ponovnim kodiranjem. Nakon otklanjanja svih grešaka, proces se nastavlja dalje.
6. **Ugradnja** – krajnji proizvod smatra se potpuno funkcionalnim i radi se migracija na živom okruženju (engl. *live environment*) tj. isporučuje se klijentu.
7. **Održavanje** – održavanje funkcionalnosti proizvoda provodi se na neodređeno vrijeme. Mogu uključivati ažuriranja radi poboljšanja funkcionalnosti, odnosno nadogradnje verzije (engl. *version update*) ili rješavanje novonastalih grešaka zakrpama (engl. *patches*).

Nakon svakog razvojnog stadija ispituje se je li projekt spreman prijeći u idući stadij. Ovaj model pretpostavlja stabilnost dizajna proizvoda kroz proces razvoja. Prije nego se primjenjivao za razvoj programskih sustava vrlo uspješno se koristio u proizvodnim pogonima i građevinskim procesima. Međutim, primjena ovog modela na razvoj programske podrške ima svoje nedostatke. Pošto se razvojni stadiji odvijaju sekvencijalno jedan iza drugoga, nisu vidljivi nikakvi rezultati sve dok se ne dovrši stadij kodiranja i implementacije. Otkrivanje nedostatka ili pogrešne implementacije bitnog korisničkog zahtjeva zahtijevat će dodatan trud kako bi se nedostatak ispravio. Što je nedostatak kasnije otkriven, to ga je u pravilu teže za ispraviti. Osim toga, ako dođe do probijanja rokova u nekom od prethodnih stadija, može postati primamljivo dobiti na vremenu na štetu stadija testiranja. Bez obzira na svoje mane, ovaj model je bitan jer sve ostale metodologije predstavljaju varijaciju modela vodopada razlikujući se samo po brzini implementacije i nivou fleksibilnosti [6].



Sl. 2.1. Prikaz vodopadnog modela razvoja programske podrške

2.1.2. Testiranje u V- modelu razvoja programske podrške

V-model razvoja programske podrške još je poznat kao validacijski i verifikacijski model. Također ga karakterizira sekvencijalni slijed izvršavanja, no za razliku od modela vodopada postoji iskorak u obliku dodatnih faza testiranja. Model ima oblik „V“ gdje je lijeva strana predstavlja detaljnu analizu za razvoj sustava programske podrške, spoj lijeve i desne strane predstavlja kodiranje i implementaciju, a desna strana predstavlja faze testiranja. Svaka razvojna faza povezana je s odgovarajućom testnom fazom. Kao i kod modela vodopada, prelazak na iduću fazu moguć je tek nakon završetka prethodne faze te svaka faza završava pisanjem popratne dokumentacije [7]. Razvojne faze V- modela su (Slika 2.2.):

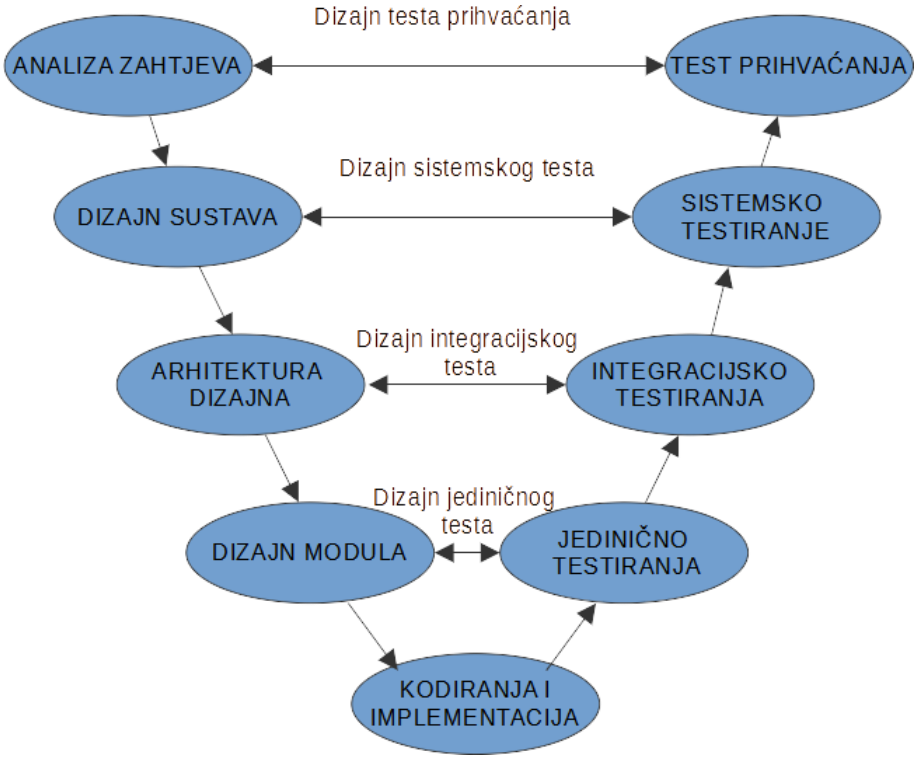
1. **Analiza zahtjeva** – određuju se funkcionalnosti koje zamišljen sustav treba imati. Međutim, ne određuje kako će programski sustav biti dizajniran ili izrađen. Na kraju se stvara dokument o zahtjevima krajnjih korisnika. Zahtjevi definirani u ovom dokumentu validirat će se testom prihvaćanja u 9. fazi. Dizajn testa prihvaćanja osmišlja poslovni tim krajnjih korisnika.
2. **Dizajn sustava** – razvija se pregled tehničkog dizajna programskog sustava visoke razine. Analizira se i objašnjava poslovna logika predloženog sustava i razmatraju se moguće tehnologije koje bi mogle biti korištene za implementaciju. Stvara se dokument koji opisuje generalni sadržaj sustava, strukture podataka, primjere korisničkih sučelja i ostale nacrte po kojima će se vršiti implementacija. Dizajn

sistemskog testiranja ne izrađuju testeri, nego poslovni tim krajnjih korisnika, a testiranje će se provesti u 8. fazi.

3. **Arhitektura dizajna** – projektira se veza i ovisnosti između sučelja pojedinih modula te se nabrajaju i kratko opisuju moduli koji će međusobnim djelovanjem u cjelini tvoriti programski sustav. Rezultat projektiranja je dokument po kojem testeri mogu pripremiti dizajn integracijskog testiranja koji će se provesti u 7. fazi.
4. **Dizajn modula** – razvija se detaljan pregled tehničkog dizajna programskog sustava niske razine. Svaki zaseban modul koji čini sustav objašnjen je tako da je programeru razumljiv te po njemu može izravno započeti s kodiranjem. Sadrži funkcionalnu logiku u obliku opisa i pseudo kôda po kojima programer piše metode u kôdu. Također, sadrži opis svih tablica baza podataka, sve reference aplikacijskog programskog sučelja, sve ovisnosti (engl. *dependencies*), sve moguće poruke grešaka (engl. *error messages*), opis očekivanih ulaza i izlaza pojedinog modula itd. Prva testiranja odvijaju se upravo nad rezultatima ove faze razvoja.
5. **Kodiranje i implementacija** – razvija se izvorni kôd prateći dokument specifikacije dizajna modula iz prošlog koraka. Manji dijelovi sustava razvijaju se zasebno prije nego ih se zajedno integrira u gotov sustav.
6. **Jedinično testiranje** (engl. *unit testing*) – testovi koji djeluju na najnižoj razini tj. na razini kôda ili programske jedinice/modula. *Unit testovi* verificiraju kako najmanja programska jedinica, određena metoda ili funkcija, točno funkcionira dok je izolirana.
7. **Integracijsko testiranje** – osiguravaju pravilnu i nesmetanu međusobnu komunikaciju modula sustava tj. testiraju sučelja modula koji su razvijani zasebno i neovisno jedan o drugom.
8. **Sistemsko testiranje** (engl. *system testing*) – osigurava da sustav kao cjelina ispunjava sve funkcionalne i nefunkcionalne zahtjeve definirane testnim planom od strane poslovnog tima korisnika.
9. **Test prihvaćanja** (engl. *user acceptance testing,* UAT) – plan testiranja također izrađuje poslovni tim korisnika. Sustav se pokreće u okruženju koji zrcali poslovno okruženje (engl. *production environment*) te mu se u realnom vremenu i s realnim korisničkim podacima šalju korisnički zahtjevi. Cilj je uvjeriti se kako je sustav spreman za korištenje i da se zadovoljavaju svi zahtjevi korisnika prije nego se razvijeni sustav isporuči korisniku i ugradi u poslovno okruženje.

Ovaj model veći fokus stavlja na stvaranje proizvoda besprijekorne kvalitete te se zbog toga često koristi u razvijanju sustava kritičnih po sigurnost, kao npr. u zdravstvenoj, automobilskoj i

zrakoplovnoj industriji. Svi napori se ulažu u to kako bi se svi rizici u ranoj fazi razvojnog procesa identificirali i uklonili. Međutim, radi svih opsežnih testiranja i izrade dokumentacije on oduzima mnogo vremena samom procesu kodiranja i implementacije. Također, poput modela vodopada, on nema velikih prostora za odstupanja od početnih zahtjeva, što ga čini nefleksibilnim [8].



Sl. 2.2. Prikaz V-modela razvoja programske podrške

2.1.3. Testiranje u Agilnom Scrum modelu razvoja programske podrške

Scrum model slijedi Agilni pristup razvoju programske podrške. Kada se za model kaže da je agilan, tada se smatra da je on hitar, okretan, fleksibilan i spretan [10]. U modelu se smatra kako se ne treba strogo držati dugoročnog cilja koji je detaljno i precizno isplaniran kao kod prethodna dva modela. Umjesto toga naglasak stavlja na iterativni pristup razvoja, stoga definira kraće vremenske okvire unutar kojih timovi rade na cijelom životnom ciklusu razvoja programskog sustava. Ti vremenski okviri obično ne traju duže od četiri tjedna i jedan okvir čini jednu iteraciju [11]. Ideja je da rezultat svake iteracije bude isporuka nekog funkcionalnog dijela sustava koji se može prikazati klijentu i primiti povratne informacije. Razmjena informacija odvija se češće i manja je šansa za nesporazume između timova i klijenata jer se isporuka odvija sustavno kroz proces razvoja čak i ako se zahtjevi kroz proces mijenjaju. Svaka iteracija sadrži (Slika 2.3.):

1. **Planiranje** – određivanje rokova, ciljeva, timova, potrebno vrijeme za razvoj sustava.

2. **Analiza zahtjeva** – definiranje zahtjeva klijenta, poslovna logika, što će sustav morati moći obavljati.
3. **Dizajn** – izrada toga dijagrama, UML dijagrama za bolje razumijevanje elemenata programskog sustava.
4. **Kodiranje / implementacija** – po razumijevanju iz prethodnog koraka implementiraju se opisane funkcionalnosti.
5. **Testiranje / kontrola kvalitete** – provjerava se da nova funkcionalnost zadovoljava korisnikove uvjete te da djeluje bez poteškoća. Također, pokreću se i svi ostali testovi u ciklusu regresijskog testiranja (engl. *regression testing*) kako bi se uvjerilo da nova funkcionalnost ne ometa neku drugu funkcionalnost.
6. **Ugradnja** – funkcionalnost se ugrađuje na radno okruženje (engl. *development environment*) kako bi se cijeli sustav pokrenuo i isprobao.
7. **Povratna informacija** – ako su klijenti zadovoljni rezultatima ove iteracije, vrši se nadogradnja *live environment* sustava s novom funkcionalnosti i time završava jedna iteracija. Iduća iteracija označava rad na idućoj funkcionalnosti itd.

Agilni model više je adaptivne nego predvidive prirode što mu omogućuje veću fleksibilnost kod potrebe za promjenama i dopunama u korisničkim zahtjevima [11]. Testiranje minimalno kasni za kodiranjem, a komunikacija testera i programera odvija se kontinuirano jer zajedno sudjeluju u razvojnom ciklusu svake pojedine funkcionalnosti. Kontinuirano testiranje i povratna informacija igraju važnu ulogu u ranom identificiranju nedostataka kôda, a podjela procesa na iteracije omogućuje brzu i efikasnu korektivnu akciju.

Sl. 2.3. Prikaz jedne iteracije agilnog scrum modela razvoja programske programske podrške [11]

2.2. Vrste testiranja programske podrške

Najosnovnija podjela testiranja je ona na testiranje s poznavanjem unutarnje strukture kôda (engl. *white box testing*) i testiranje bez poznavanja unutarnje strukture kôda (engl. *black box testing*) [12]. Kod poznatog izvornog kôda (engl. *source code*) mogu se provjeravati sva grananja poput „*if-else*“ u kôdu (engl. *branch testing*), svi mogući logički putevi (engl. *path testing*), ispravno ponašanje svih petlji (engl. *loop testing*) itd. Međutim, ovo „kirurško seciranje“ programskog sustava često nije moguće održivo provoditi radi vremenske zahtjevnosti, pogotovo kod glomaznijih programa, ali je vrlo efektivno kod manjih programa koji su prekritični da bi si mogli priuštiti podbacivanje (engl. *to-critical-to-fail*) [12]. Jedino *white box testiranje* koje se spominje u nabrojanim modelima (V-model u poglavlju 2.1.2.) je *unit* testiranje koje u pravilu razvijaju programeri, a ne testeri [13][14]. Ono po definiciji spada u *white box testiranje*, no svojim načinom verifikacije i određivanja ispravnosti sličnije je *black box* testiranju, stoga se često koristi u kombinaciji s tom vrstom testiranja. U ovom će poglavlju biti nanovo spomenuto pod kategorijom *black box* testiranja, u svrhu jasnije podijele testova po razinama sustava.

Mnogo jednostavnije za provođenje, a time i mnogo raširenije je *black box testiranje*. Pošto kôd nije poznat, ono se oslanja na validaciju izlaza „iz kutije“ u odnosu na poznate ulaze „u kutiju“. Ovisno o razini testova, ta „kutija“ može biti:

- **Cijeli sustav** – sustavno testiranje i prihvatno testiranje
- **Podsustavi** – integracijsko testiranje
- **Pojedinačni moduli sustava** – *unit* testovi

Ovi testovi već su opisani u poglavlju 2.1.2. kod V-modela razvoja programske podrške. *Black box* testiranje je moguće delegirati na vanjske suradnike (engl. *third party*), koji su specijalizirani upravo za opsežno i kontinuirano provođenje *black box testiranja.* Pri tome nije upitna sigurnost izvornog kôda programa pošto mu treća strana nema pristup.

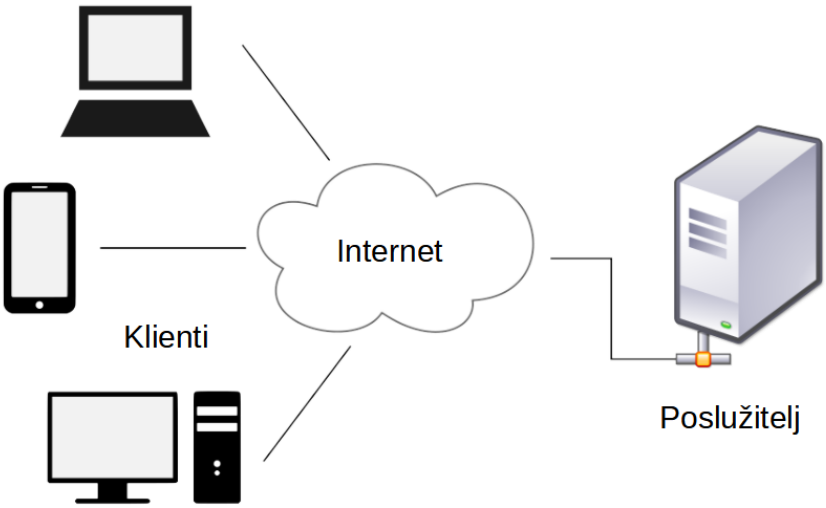
2.3. Metodologije testiranja integracije klijent-poslužitelj

Poslužitelj je računalo ili sustav računala koji preko mreže pruža resurse, podatke, usluge ili programe drugim računalima, laptopima ili mobilnim uređajima zvanim klijentima. Postoje mnoge vrste poslužitelja uključujući web poslužitelje, aplikacijske poslužitelje, poslužitelje e-pošte, virtualne poslužitelje i druge [15]. Uređaj mora biti konfiguriran tako da sluša zahtjeve klijenata putem mrežne veze da bi mogao djelovati kao poslužitelj. Kada klijent zatraži podatke ili funkcionalnost od poslužitelja, on šalje zahtjev preko mreže, a poslužitelj prihvaća zahtjev i

odgovara odgovarajućim informacijama. Ovaj model povezanosti klijenta i poslužitelja naziva se zahtjev-odgovor (engl. *request-response*) model, a još je poznat i kao poziv-odgovor (engl. *call-response*) model.

Poslužitelj može obavljati brojne druge zadatke kao dio jednog zahtjeva i odgovora uključujući verifikaciju identiteta podnositelja zahtjeva, osiguravanje da je klijent ovlašten za pristup traženim podacima ili resursima i pravilno formatiranje traženog odgovora u očekivani oblik. Jedna od najčešćih vrsta poslužitelja na današnjem tržištu je web poslužitelj. Web poslužitelj je posebna vrsta aplikacijskog poslužitelja koji putem interneta ili intraneta korisnicima pruža pristup i mogućnost manipulacije statičkim resursima s pomoću HTTP metoda. Web poslužitelj odgovara na zahtjeve preglednika koji radi na klijentskim uređajima za web stranice ili druge web usluge kao što je prikazano slikom 2.4. [16].

Dok poslužitelj klijentima pruža usluge i resurse preko korisničkog sučelja (engl. *user interface*, UI), obrađuje zahtjeve i vraća odgovore, API definira komunikacijske protokole među aplikacijama i omogućuje njihovu međusobnu integraciju. Aplikacije se često sastoje od tri sloja: podatkovnog sloja, servisnog sloja (API) i prezentacijskog sloja (UI). API sloj sadrži poslovnu logiku aplikacije - pravila o tome kako korisnici mogu komunicirati s uslugama, podacima ili funkcijama aplikacije. Budući da se API sloj izravno dotiče i podatkovnog sloja i prezentacijskog sloja, on predstavlja dobro mjesto za osiguranje kvalitete. Za razliku od testiranja korisničkog sučelja koje se svodi na provjeru izgleda web sučelja ili funkcionalnosti određenog gumba za neku akciju, testiranje API-ja stavlja naglasak na testiranje poslovne logike, brzine odziva podataka te sigurnosnih i uskih grla (engl. *bottlenecks*) u izvedbi. API testiranje općenito se sastoji od slanja zahtjeva jednoj ili ponekad više krajnjih točaka (engl. *endpoints*) API-ja i provjere odgovora, iako može uključivati i ostale aspekte poput procjene performansi sustava, skalabilnost i iskorištenost resursa [17]. Primarni cilj testiranja je provjera rada poslovnih značajki i funkcionalnosti aplikacije od strane klijenta i poslužitelja te osiguravanje ispravne razmjene zahtjeva i odgovora [18].



Sl. 2.4. Prikaz spoja klijenata na poslužitelj preko internetske mreže [16]

2.4. Alati za testiranje odgovora aplikacijskog programskog sučelja poslužitelja na zahtjeve

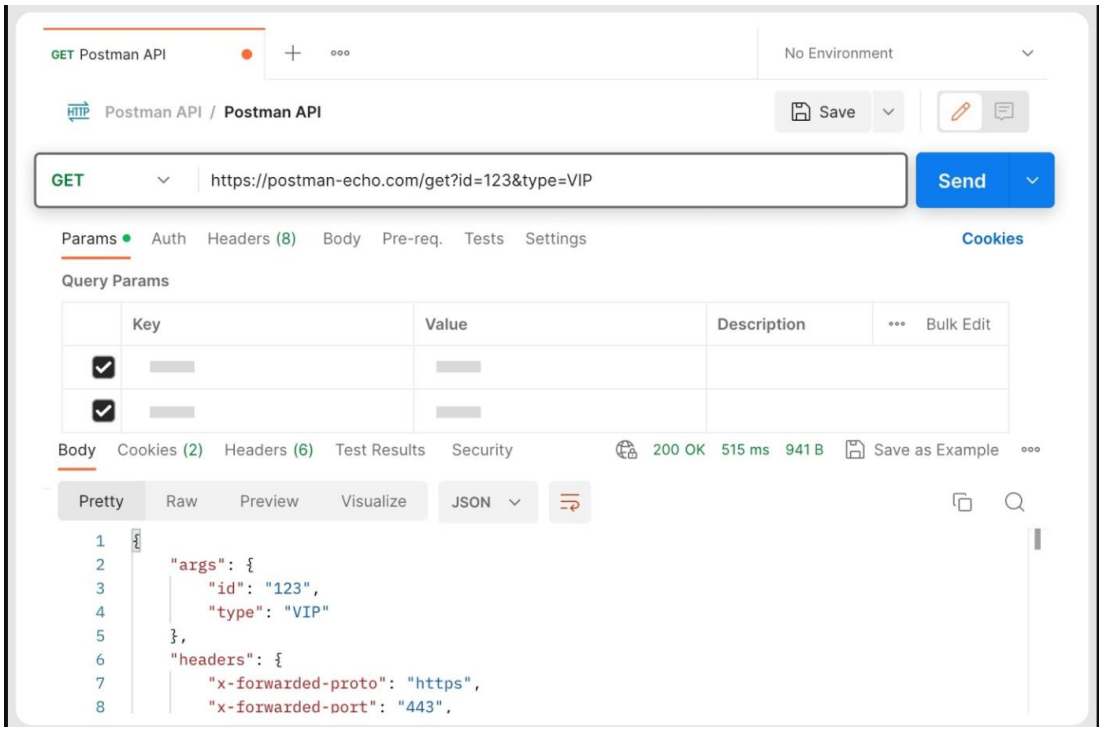
Alati za testiranje aplikacijskog programskog sučelja ključni su za osiguravanje točnosti, učinkovitosti i pouzdanosti odgovora poslužitelja na različite zahtjeve klijenta. Oni pomažu programerima i testerima pri provjeri funkcionalnosti, izvedbe i sigurnosti API-ja koji čine okosnicu modernih web i mobilnih aplikacija. Simulacijom različitih zahtjeva klijenata i analizom odgovora poslužitelja, ovi alati mogu identificirati probleme poput nevažećih podataka, sporog vremena odgovora i sigurnosne ranjivosti smanjujući rizik od pogrešaka i poboljšavajući ukupnu kvalitetu softvera. U ovom odlomku analizirat će se neki poznati alati za testiranje API-ja, njihovih funkcionalnosti i kako doprinose pouzdanosti modernih aplikacija.

Alati kojima se testiraju API-ji mogu se kategorizirati po tome kako se korisnici njima služe. Alati korisnicima mogu pružati grafičko korisničko sučelje (engl. *graphical user interface*, GUI) kojim testeri stvaraju i izvršavaju API zahtjeve. Alati ove vrste često ne zahtijevaju programersko predznanje što ih čini jednostavnima za korištenje u svrhu ručnog testiranja. Za razliku od njih, postoje alati čije je korisničko sučelje temeljeno na sučelju naredbenog retka (engl. *command line interface*, CLI) te funkcioniraju s pomoću naredbi terminala. Alati ove vrste nude mogućnost automatizacije i određenu fleksibilnost korisnicima kojima je skriptiranje preferirani oblik pisanja testova. Isto tako, postoje i alati koji su dizajnirani s fokusom na integraciju s razvojnim okruženjima. Ova vrsta alata korisnicima omogućuje programiranje API testova u sklopu kôda unutar zasebnog programskog rješenja namijenjenog testiranju i dobro su prilagođeni za kontinuiranu integraciju (engl. *continuous integration*, CI) i kontinuiranu isporuku (engl.

continuous deploy, CD). Timovi zaduženi za QA odabiru one alate koji će najbolje odgovarati njihovim specifičnim procesima i potrebama testiranja. Neki od najpoznatijih alata za testiranje API-ja su: Postman, SoapUI i Katalon Studio koji su pretežito GUI alati, Apache JMeter koji je CLI alat te Rest Assured i Karate DLS koji su oboje programabilni alati.

2.4.1. Postman alat za testiranje API-ja

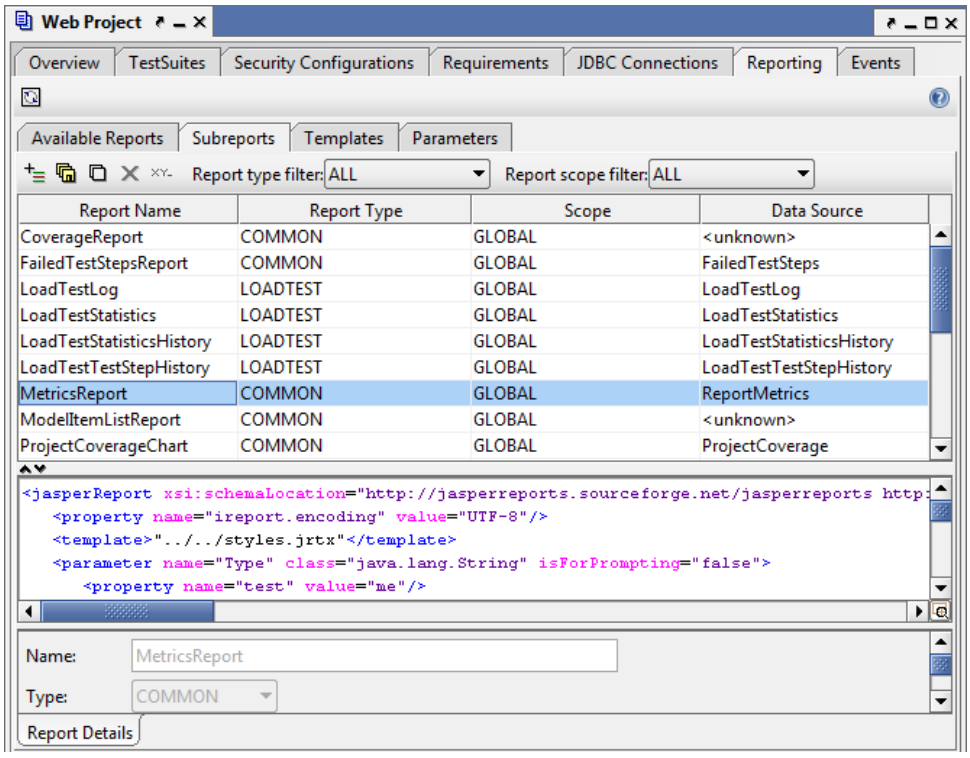
Razvijanje alata započeo je Abhinav Asthana kako bi pojednostavio proces testiranja API-a, međutim projekt je brzo narastao te su mu se u osnivanju tvrtke *Postman, Inc.* pridružili dvojica kolega: Ankit Sobti i Abhijit Kane. Do danas su postali vodeća svjetska API platforma [19]. Postman je alat koji pojednostavljuje korake izrade i korištenja API-ja. Alat pruža grafičko korisničko sučelje kojim korisnici mogu jednostavno stvarati i slati HTTP zahtjeve API-jima (Slika 2.5.). Zahtjeve koji se izvršavaju uzastopno moguće je pohraniti u zbirke zahtjeva te tako automatizirati proces API testiranja. Postman podržava GET, POST, PUT i DELETE metode i prikladan je za testiranje *RESTful* usluga. Također, pruža mogućnost naprednog upravljanja varijablama okruženja koje korisnicima služi za lako prebacivanje između različitih konfiguracija. Korisnik varijable može pohraniti na razini kolekcije zahtjeva, radne okoline i na globalnoj razini. Osim toga omogućuje izradu API dokumentacija i njihovo dijeljenje s ostalim članovima tima.



Sl. 2.5. Prikaz korisničkog sučelja Postman alata za API testiranje

2.4.2. Soap UI alat za testiranje API-ja

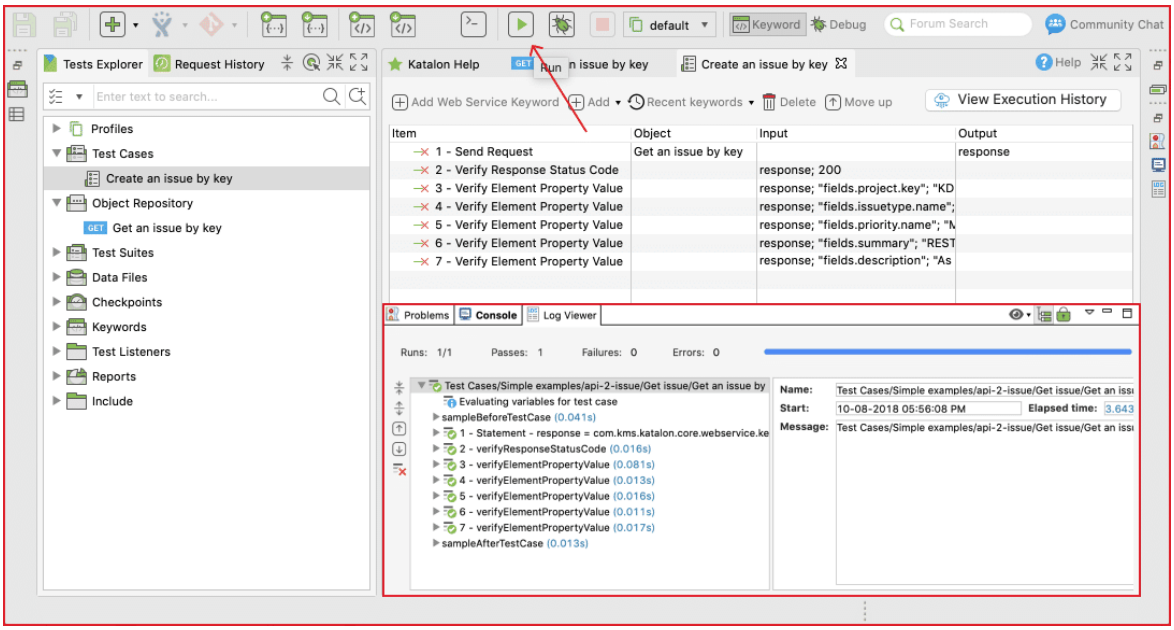
SoapUI je alat koji se koristi pri testiranju SOAP i REST web servisa, za razliku od Postman alata koji je fokusiran samo na REST API-je. Korisnicima omogućuje stvaranje složenih testnih tvrdnji i scenarija za provjeru API odgovora. Također, nudi skup značajki za testiranje opterećenja i sigurnosti što mu omogućuje stvaranje svestranih testnih scenarija. Ovaj alat spada u GUI vrstu alata za testiranje API-ja, ali isto tako podržava skriptiranje u *Groovy* programskom jeziku (baziranom na *Typescript* programskom jeziku) te tako pruža fleksibilnost prilagođenog ponašanja testiranja. Pro verzija SoapUI nudi dodatne napredne značajke kao što su testiranje temeljeno na podacima (engl. *data driven testing*) koje korisnicima omogućuje pokretanje testova s različitim skupovima ulaznih podataka, *debugiranje* pri pokretanju testova i poboljšane mogućnosti izvješćivanja koje pomažu pri analizi rezultata testova. SoapUI ima nešto složeniji GUI od Postmana (Slika 2.6.) i može zahtijevati nešto više vremena za učenje kako bi se učinkovito koristilo alatom. Međutim nakon inicijalnog upoznavanja s funkcionalnostima SoapUI alata, on pruža snažno i pouzdano testno okruženje [20].



Sl. 2.6. Prikaz korisničkog sučelja Soap UI alata za API testiranje [20]

2.4.3. Katalon Studio alat za testiranje API-ja

Katalon Studio je alat koji osim testiranja API-ja podržava i testiranje web, mobilnih i desktop aplikacija koji se jednostavno postavljaju s pomoću grafičkog korisničkog sučelja što ga čini dostupnim testerima različitih razina stručnosti. Podržava ručne i automatizirane testove te skriptiranje u *Groovy* programskom jeziku čime pruža mogućnost stvaranja opsežnih testnih scenarija. Dodatno povećava automatizaciju testiranja pružanjem integracije s raznim alatima CI/CD cjevovoda (engl. *pipeline*). Također, omogućuje različite značajke izvješćivanja kao što su prilagođavanje sadržaja i formata izvješća poput: HTML, PDF, CSV i Junit. Prilagođavanjem sadržaja i formata izvještaja moguće je poštivanje standarda izvješćivanja određenog unutar tima i jednostavna integracija s nekim drugim alatima. Fleksibilnosti doprinosi i mogućnost integracije s JIRA, Jenkins, Git i ostalim alatima [21]. Grafičko sučelje Katalon Studio alata prikazano je slikom 2.7.



Sl. 2.7. Prikaz Katalon Studio alata za API testiranje [21]

2.4.4. Apache Jmeter alat za testiranje API-ja

Apache JMeter alat je otvorenog kôda korišten za kontrolu kvalitete izvedbe web aplikacija koje uključuje i testiranje API-ja. Smatra se CLI alatom pošto pruža široku funkcionalnost za pokretanje skriptiranih testova izravno iz naredbenog retka s pomoću *jmeter* naredbe. Taj način rada podržava pokretanje više testnih planova paralelno ili uzastopno bez korištenja grafičkog korisničkog sučelja. To je posebno korisno za izvršavanje testova performansi većeg opsega i testnih scenarija regresijskog testiranja. Osim toga, korištenje Jmetera u CLI načinu rada manje opterećuju resurse sustava u usporedbi s GUI načinom rada što osigurava da se sustav koji izvodi

testove ne preoptereti. Također, ovaj način rada podržava daljinsko, odnosno distribuirano testiranje tako što se može pokrenuti više Jmeter instanci koje pokreću testove na različitim poslužiteljima, a kontrolira ih se iz središnje naredbene linije. CLI sučelje Jmeter alata prikazano je slikom 2.8.

```
jmeter -Ljmeter.engine=DEBUG
jmeter -Lorg.apache.jmeter.engine=DEBUG
jmeter -Lcom.example.foo=DEBUG
jmeter -LDEBUG
```

Sl. 2.8. Prikaz poziva jmeter alata s pomoću komadnog korisničkog sučelja [22]

2.4.5. Rest Assured alat za testiranje API-ja

Za razliku od prethodnih alata koji su samostalne aplikacije, Rest Assured alat za testiranje API-ja je biblioteka otvorenog kôda koja omogućuje pisanje testova izravno kao dio kôda unutar Java projekta [23]. Također, pošto ovaj alat zahtjeva razvojno okruženje, prvenstveno ga koriste programeri te je njegovo korištenje nešto teže za širu publiku, no zato pruža veću fleksibilnost i kontrolu od prethodnih alata. Većoj kontroli doprinosi mogućnost da se testni kôd drži u istom ekosustavu kao i kôd testirane aplikacije. S Rest Assured, programeri mogu jednostavno stvoriti čitljive i održive testne skripte koje se neprimjetno integriraju u projekte temeljene na Javi. Pružanjem jezika specifičnog za domenu (engl. *domain-specific language,* DSL) pri pisanju testova, smanjuje se potrebna količina kôda za obavljanje uobičajenih zadataka kao što su postavljanje parametara zahtjeva, slanje zahtjeva i provjera valjanosti odgovora. Fluidna sintaksa Rest Assureda povećava čitljivost i čini testove razumljivijim i lakšim za održavanje, kao što je prikazano programskim kôdom 2.1. Jedna od ključnih značajki Rest Assureda je mogućnost rukovanja složenim strukturama zahtjeva i odgovora, uključujući JSON i XML sadržaje. To ga čini idealnim za testiranje raznih API krajnjih točaka i scenarija [24].

```java
import static io.restassured.RestAssured.*;
import static io.restassured.matcher.RestAssuredMatchers.*;
import static org.hamcrest.Matchers.*;

import org.junit.Test;

public class ApiTest {

    @Test
    public void testGetEndpoint() {
        given().
            baseUri("https://jsonplaceholder.typicode.com").
            header("Content-Type", "application/json").
        when().
            get("/posts/1").
        then().
            statusCode(200).
            body("userId", equalTo(1)).
            body("id", equalTo(1)).
            body("title", equalTo("test title"));
    }
}
```

Programski kôd 2.1. Prikaz primjera API testa u java kôdu koristeći Rest Assured alat

2.4.6. Karate DSL alat za testiranje API-ja

Poput Rest Assured alata, Karate DSL alat za testiranje API-ja spada u programabilne alate. Nešto je svestraniji alat, jer uz API-je omogućuje i testiranje web grafičkih korisničkih sučelja. Kao i do sada spomenuti alati podržava različite vrste HTTP zahtjeva i tvrdnji za REST testiranje, no također podržava SOAP i GraphQL API testiranje te uključuje značajke za testiranje komponenti web sučelja s pomoću Seleniuma. Karate DSL alat je platforma za automatizaciju testiranja otvorenog koda koja se temelji na Cucumber okviru, što znači da omogućuje korištenje sintakse u stilu razvoja programa vođenog ponašanjem (engl. behaviour-driven development, BDD). BDD omogućuje pisanje testnih scenarija u formatu prirodnog jezika. To ih čini čitljivima i tehničkim i netehničkim sudionicima razvoja, odnosno klijentima, sponzorima, poslovnim analitičarima, menadžerima itd. [25]. Također, podržava Java i JavaScript pozive izvan prirodnog jezika.

```
Feature: Simple API Test

  Scenario: Verify GET request to fetch a post
    Given url 'https://jsonplaceholder.typicode.com/posts/1'
    When method GET
    Then status 200
    And match response.userId == 1
    And match response.id == 1
    And match response.title == 'test title'
```

Programski kôd 2.2. Prikaz primjera API testa u java kôdu koristeći Karate DSL alat

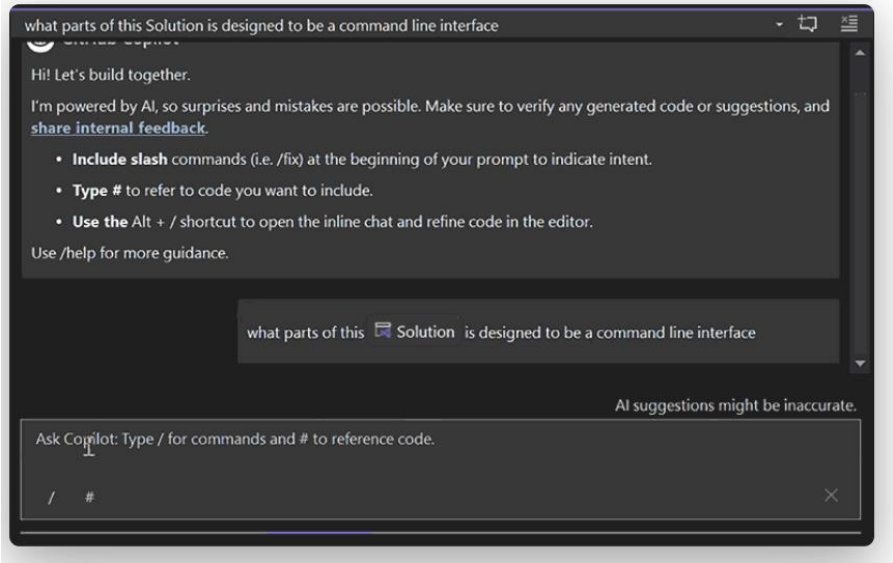
3. TEHNOLOGIJE KORIŠTENE PRI IZRADI RADA

Pri razvoju programskog alata u diplomskom radu korišteno je više tehnologija. Programski kôd pisan je u *C#* programskom jeziku unutar Visual Studio razvojnog okruženja (engl. *integrated development environment,* IDE). GUI je ostvaren s pomoću Windows Forms obrazaca, a njegova funkcionalnost ostvarena je kombinacijom različitih biblioteka. Dodatne biblioteke korištene za slanje HTTP zahtjeva i korištenje *Assert* metoda za validaciju. Također su opisane u ovom poglavlju kao i još neke biblioteke koje su omogućile uspješnu implementaciju potrebnih funkcionalnosti alata.

3.1. Razvojno okruženje Visual Studio 2022

Visual Studio razvojno je okruženje koje je razvila tvrtka Microsoft. On je uređivač izvornog kôda koji uključuje IntelliSense komponentu za potpomognuti dovršetak i refaktoriranje kôda. Osim toga, ima ugrađen program za ispravljanje grešaka (engl. *debug*) na razini kôda i na razini izvođenja programa, odnosno omogućuje postavljanje prijelomnih točaka (engl. *breakpoints*) te pruža korisničko sučelje za povezivanje s alatima za verzioniranje kôda poput Gita. Sadrži podršku za razne programske jezike kao što su *C++*, *C#*, Java, Python, PHP, Go, JavaScript i TypeScript. Dostupan je na Windows, MacOS i Linux operativnom sustavima [26].

Blagodati umjetne inteligencije (engl. *artificial inteligence,* AI) također su podržane u obliku ugrađenog GitHub Copilota. To je alat koji pomaže učinkovitije kodiranje tako što prati kôd koji se tipka u uređivaču te na temelju njega predlaže završetak cijele linije kôda. Osim usluge dovršavanja kôda, postoji i prostor za čavrljanje (engl. *chat*) gdje se AI može ljudskim rječnikom pitati za savjet vezan uz dio projekta i po danom savjetu AI može generirati isječak koda (engl. *code snippet*), kao na primjer kôd za opisane klase ili jedinične testove, kao što je prikazano slikom 3.1. [27].



Sl. 3.1. Prikaz upita GitHub Copilota za savjet koristeći prostor za čavrljanje [27]

3.2. Biblioteka RestSharp

RestSharp je biblioteka *C#* programskog jezika za olakšano slanje HTTP zahtjeva. Ona omotava (engl. *wraps*) osnovnu *.NET* klasu *HttpClienta* i dodaje mu mogućnost podešavanja standardnih parametara bilo koje vrste, a ne samo zaglavlja [28]. Također, olakšava dodavanje raznih parametara HTTP zahtjevima na poslužitelj poput različitih segmenata URL-a, nizova URL upita (engl. *query strings*), zaglavlja (engl. *header*), kolačića (engl. *coockie*) i tijela (engl. *body*). Osim toga, nudi veći izbor načina za dodavanje tijela zahtjeva, poput: JSON, XML, URL-kodiranih podataka obrasca (engl. *url-encoded form data*) i drugih. Raščlanjivanje (engl. *parse*) dobivenih odgovora u objekte klasa (i obratno) olakšano je ugrađenom JSON, XML i CSV serijalizacijom i deserijalizacijom te je moguće i dodavanje prilagođenih serijalizatora. RestSharp je projekt otvorenog kôda (engl. *open source*) sa zajednicom od preko 252 održavatelja i koristi se u preko 71.4 tisuće projekata na *Githubu* [29].

3.3. NUnit biblioteka

NUnit je biblioteka otvorenog kôda, izvorno prenesena iz JUnit biblioteke za *Java* programski jezik, čije metode omogućuju širok raspon testiranja unutar .NET programskog okvira u C#-*u*. Podržava sve od jediničnih testova, implementacije razvoja programa vođenog testiranjem (engl. *test driven development*, TDD) do potpunog testiranja sustava i integracije [31]. Pruža sveobuhvatan okvir s mnogo različitih načina za validaciju ponašanja kôda prema određenim očekivanjima. Mnogi aspekti NUnit biblioteke po potrebi se mogu proširiti za specifične svrhe.

Prve verzije NUnit biblioteke razvili su Charlie Poole, James Newkirk, Alexei Vorontsov, Michael Two i Philip Craig, a najnoviju verziju NUnit 3 koja je potpuno iznova napisana kreirali su Charlie Poole, Rob Prouse, Simone Busoli, Neil Colvin i brojni suradnici zajednice. NUnit sastoji se od nekoliko projekata: osnovni projekt (engl. *core project*), nadogradnje Visual Studio okruženja (engl. *Visual Studio exstension*) i NUnit nadogradnja za pogonski sklop (engl. *nunit engine extension*) [30]. Licenca pod kojom je biblioteka objavljena omogućuje njeno korištenje u besplatnim i komercijalnim aplikacijama i bibliotekama bez ograničenja. Različiti paketi NUnit biblioteke broje više od 600 milijuna preuzimanja na *NuGet.org* [31].

3.4. Gherkin sintaksa za pisanje testnih scenarija

Gherkin sintaksa je jezik osmišljen tako da tehnički i netehnički dionici lako čitaju kôd i razumiju ponašanje razvijanog programa. Koristi se u BDD razvoju, a temelji se na jednostavnim jezičnim strukturama "pošto je dano-kada-tada" (engl. *given-when-then*) koji specificiraju preduvjete, radnje i očekivane rezultate testnog slučaja. Pojednostavljeni primjer sintakse prevedene na hrvatski jezik izgledao bi nešto poput: „Pošto je dano da sam uspješno prijavljen na aplikaciju, kada pritisnem gumb profil, tada se otvori stranica mog profila“, a programskim kodom 3.1. prikazana je implementacija tog testa. *Given* je ključna riječ koja označava početno stanje prije poduzimanja akcije, *When* opisuje radnju ili događaj koji pokreće testirano ponašanje, a ključna riječ *Then* označava očekivani rezultat poduzete radnje. Ova struktura pomaže i kao programska dokumentacija, jer pruža jasnu specifikaciju ponašanja sustava [32].

```
Scenario: TC001
    Given I am succesfully logged in
    When I press Profil button
    Then Profil page is displayed
```

Programski kôd 3.1. Prikaz jednostavnog primjera testa napisanog Gherkin sintaksom

4. RAZVOJ PROGRAMSKOG RJEŠENJA

Kombinacija tehnologija iz prethodnog odlomka korištena je u izradi programskog rješenja koje je opisano u ovom dijelu diplomskog rada. Prvo su navedeni su zahtjevi i objašnjene funkcionalnosti koje će alat morati biti u stanju obavljati. Nakon toga je prikazana struktura Visual Studio projekta te pobliže objašnjene uloge svake datoteke. Prikazano je i opisano grafičko programsko sučelje razvijenog alata i njegovi elementi. Također, u ovom poglavlju detaljnije je opisan programski kôd i prikazane su sve bitnije metode programa alata.

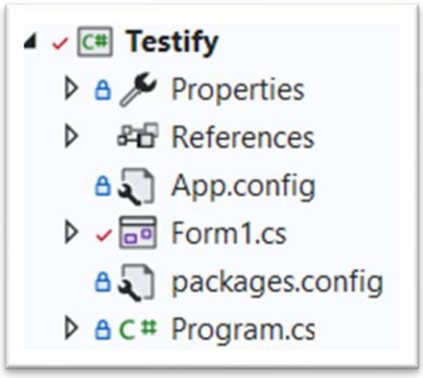
4.1. Opis funkcionalnosti alata

Alat razvijen u ovom diplomskom radu ima grafičko korisničko sučelje kojim se šalju HTTP zahtjevi i pohranjuju različiti testni scenariji. GUI je intuitivan i jednostavan za navigaciju zbog čega korisnici mogu brzo i efikasno kreirati nove testove, što doprinosi dobrom korisničkom iskustvu (engl. *user experience, UX*). Također, omogućuje vizualno definiranje GET, POST, PUT i DELETE metoda te parametre zaglavlja i tijela zahtjeva. Koraci validacije vrše provjeru statusnih kodova i pohranjuju vrijednosti unutar odgovora. Svaki korak testa opisan je prirodnim jezikom po Gherkin sintaksi čime se olakšava komunikacija i korištenje alata tehničkom i netehničkom osoblju. Automatizacija provođenja testova, koju alat pruža, smanjuje ručni rad i povećava pouzdanost testnih rezultata. Time podržava kontinuirano testiranje i brzu povratnu informaciju o kvaliteti kôda, što je ključno kod iterativnog razvoja gdje često dolazi do promjene kôda te ga je potrebno često testirati. Provedeni testovi generiraju izvješća koja sadrže informacije o uspješnosti njihovog izvršavanja te eventualne greške. Izvješća pružaju brz uvid u stanje API-a te omogućuju identificiranje problema prije nego dođu do krajnjih korisnika. Olakšanu analizu i dijeljenje s timom omogućuje dostupnost izvješća u različitim formatima, uključujući .txt, .csv i .html. Alat podržava spremanje svakog testa te ih lako nanovo učitava i provodi po potrebi. Izvještaj testiranja sprema se u mapu s imenom testa, što čini upravljanje testovima preglednim. Spremanje testova olakšava regresijsko testiranje, odnosno njihovo ponovno provođenje kako bi se osiguralo da nove promjene ne utječu negativno na postojeće funkcionalnosti. Navedene značajke alat čine pogodnim za hitro, istraživačko testiranje bez potrebe za kodiranjem. Korištenjem ovog alata, može se smanjiti broj grešaka, poboljšati ukupnu kvalitetu i stabilnost API-ja kroz sve faze razvoja.

4.2. Struktura projekta

Prije opisivanja detalja implementacije potrebno je napraviti uvid u strukturu Windows Forms projekta i organizaciju mapa. Windows Forms projekt moguće je kreirati s pomoću čarobnjaka unutar Visual Studio razvojnog okruženja koji kroz kreaciju projekta stvara predefinirane datoteke kao što su (Slika 4.1.):

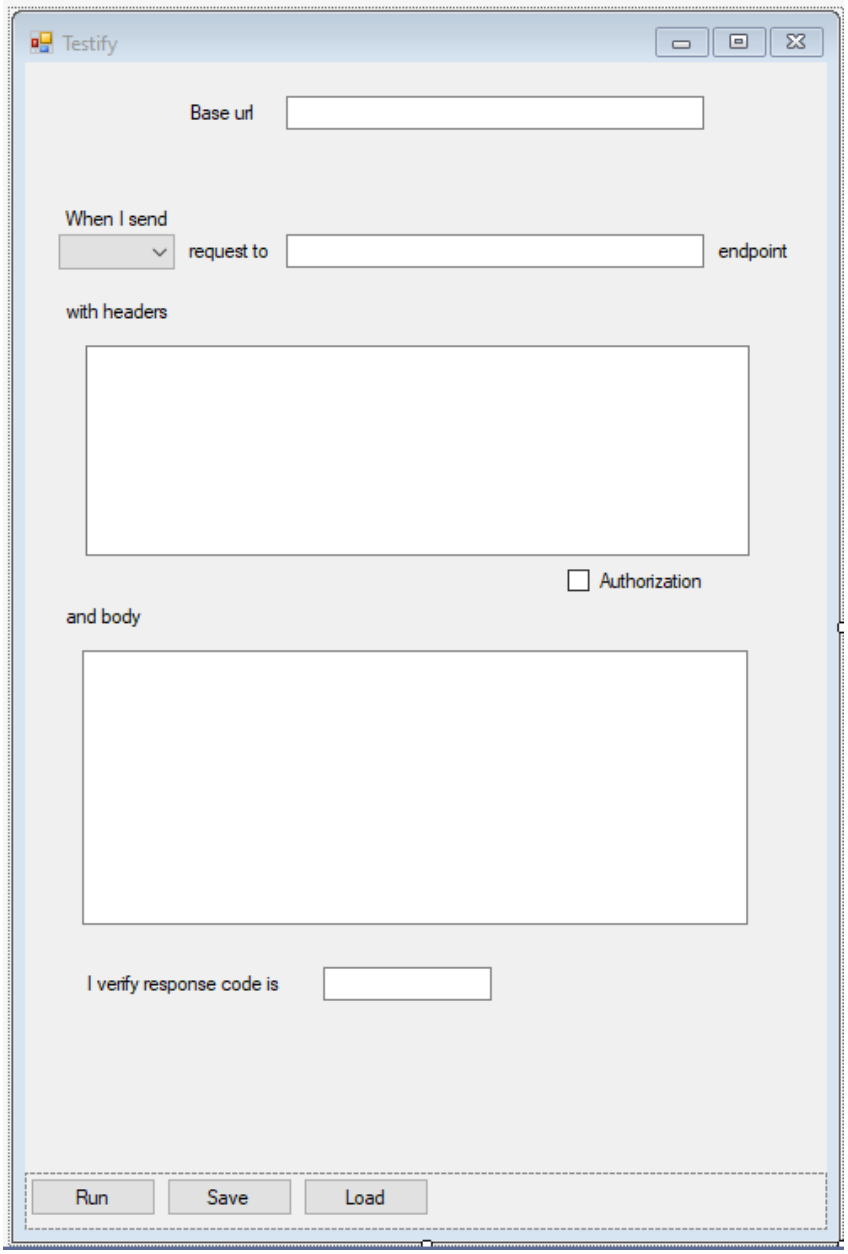
- **Properties** – sadrži konfiguracijske postavke koje se primjenjuju na čitav Visual Studio projekt. Postavke uključuju kompilacijske opcije, opcije sigurnosti i implementacije itd.
- **References** – referenca u kontekstu projekta podrazumijeva identificiranje binarnih datoteka potrebnih za uspješno pokretane aplikacije. To su najčešće .dll datoteke, no mogu biti i asemblerski sklopovi čije klase koristi aplikacija.
- **App.config** – datoteka specifična za Visual Studio projekte. To je XML datoteka kojom se može prilagoditi na koji se način lociraju i učitavaju asemblerski sklopovi. Pri izgradnji (engl. *build*) projekta razvojno okruženje automatski kopira app.config datoteku, mijenja joj naziv tako da odgovara nazivu izvršne datoteke te ju premješta u *bin* mapu.
- **Form1.cs -** datoteka klase Windows Forms u kojoj su napisane potrebne metode, funkcije i svi ostali kodovi kojima se implementiraju funkcionalnosti Windows Forms aplikacije.
- **Form1.Designer.cs –** automatski generirana datoteka korištena za inicijaliziranje elemenata dizajna Windows Forms obrasca. U većini slučajeva ju nije potrebno mijenjati, jer se za sve ručne izmjene kôda koristi *Form1.cs* datoteka.
- **Program.cs** – glavni dio aplikacije te se ujedno i prvi izvršava kod pokretanja aplikacije. Nakon pokretanja, stvara obrazac i otvara ga da korisnik počne koristiti aplikaciju.



Sl. 4.1. Prikaz strukture Windows Forms projekta u Visual Studio razvojnom okruženju

4.3. Implementacija alata za izradu testnih izvještaja poslužitelja

Windows forms obrasci korišteni su kod stvaranja GUI-ja. Definiran je prozor koji prima ulaze koji su parametri zahtjeva na API. Taj prozor ujedno omogućuje pokretanje i spremanje testa tipkom *save*. Osim toga, omogućuje učitavanje prethodno stvorenih testova s pomoću *load* gumba. HTTP metoda odabire se s pomoću padajućeg izbornika, a ostali parametri zahtjeva na API unose se u *textboxove*. Bazni URL odvojen je zasebnim *textboxom* jer je vezan uz okruženje (engl. *environment*) poslužitelja. Promijeni li se *host* URL poslužitelja potrebno je korigirati samo bazni dio URL-a, dok *endpointi* ostaju isti. Autorizacija se može dodati u zaglavlje s pomoću *Authorization checkboxa*. Kao što je prikazano slikom 4.2. pokretanje testa vrši se tipkom *run*.



Sl. 4.2. Prikaz Windows Forms GUI-a

Kod pokretanja alata, GUI stvara definirane grafičke elemente. Sve komponente Windows Forms obrasca inicijalizirane su unutar *Form1.Designer.cs* datoteke i služe kao podrška dizajna (engl. *design support*). Programski kôd 4.1. prikazuje stvaranje novih objekata kojima se kasnije pridružuje lokacija, ime, dimenzije i ostale karakteristike. U slučaju tipki, kao što je prikazano programskim kôdom 4.2., još se definira i upravitelj događajima (engl. *event handler*) koji na akciju pritiska tipke poziva određenu metodu. Također, kod inicijalizacije padajućeg izbornika (Programski kôd 4.3.) navedena je lista iz koje je moguće odabrati jedan element. U ovom slučaju to su GET, POST, PUT i DELETE. Tipke *run, save* i *load* smještene su uvijek uz donji rub obrasca s pomoću *flowLayoutPanela* koji ih omotava. Njegova inicijalizacija prikazana je programskim kôdom 4.4.

```
private void InitializeComponent()
        {
            this.btnRun = new System.Windows.Forms.Button();
            this.txtResponseCode = new System.Windows.Forms.TextBox();
            this.lblResponseCode = new System.Windows.Forms.Label();
            this.lblAndBody = new System.Windows.Forms.Label();
            this.txtBody = new System.Windows.Forms.TextBox();
            this.txtHeaders = new System.Windows.Forms.TextBox();
            this.lblWithHeaders = new System.Windows.Forms.Label();
            this.lblEndpoint = new System.Windows.Forms.Label();
            this.txtEndpoint = new System.Windows.Forms.TextBox();
            this.lblRequestTo = new System.Windows.Forms.Label();
            this.comboHttpMethods = new System.Windows.Forms.ComboBox();
            this.lblWhenISend = new System.Windows.Forms.Label();
            this.lblTestName = new System.Windows.Forms.Label();
            this.txtBaseUrl = new System.Windows.Forms.TextBox();
            this.flowLayoutPanel1 = new System.Windows.Forms.FlowLayoutPanel();
            this.btnSave = new System.Windows.Forms.Button();
            this.btnLoad = new System.Windows.Forms.Button();
            this.chkAuthorisationSettings = new System.Windows.Forms.CheckBox();
            this.flowLayoutPanel1.SuspendLayout();
            this.SuspendLayout();
```

Programski kôd 4.1. Prikaz stvaranja objekata dizajnerskih komponenti unutar *Form1.Designer.cs*

```
            //
            // btnRun
            //
            this.btnRun.Location = new System.Drawing.Point(3, 3);
            this.btnRun.Name = "btnRun";
            this.btnRun.Size = new System.Drawing.Size(75, 23);
            this.btnRun.TabIndex = 14;
            this.btnRun.Text = "Run";
            this.btnRun.UseVisualStyleBackColor = true;
            this.btnRun.Click += new System.EventHandler(this.button1_Click_1);
```

Programski kôd 4.2. Prikaz inicijalizacije tipke *run* unutar *Form1.Designer.cs*

```
            //
            // comboHttpMethods
            //
            this.comboHttpMethods.DropDownStyle =
System.Windows.Forms.ComboBoxStyle.DropDownList;
            this.comboHttpMethods.FormattingEnabled = true;
            this.comboHttpMethods.Items.AddRange(new object[] {
            "GET",
            "POST",
            "PUT",
            "DELETE"});
            this.comboHttpMethods.Location = new System.Drawing.Point(20, 102);
            this.comboHttpMethods.Name = "comboHttpMethods";
            this.comboHttpMethods.Size = new System.Drawing.Size(69, 21);
            this.comboHttpMethods.TabIndex = 18;
```

Programski kôd 4.3. Prikaz inicijalizacije liste mogućih odabira i ostalih parametara padajućeg izbornika unutar *Form1.Designer.cs*

```
            //
            // flowLayoutPanel1
            //
            this.flowLayoutPanel1.Controls.Add(this.btnRun);
            this.flowLayoutPanel1.Controls.Add(this.btnSave);
            this.flowLayoutPanel1.Controls.Add(this.btnLoad);
            this.flowLayoutPanel1.Dock = System.Windows.Forms.DockStyle.Bottom;
            this.flowLayoutPanel1.Location = new System.Drawing.Point(0, 659);
            this.flowLayoutPanel1.Name = "flowLayoutPanel1";
            this.flowLayoutPanel1.Size = new System.Drawing.Size(476, 34);
            this.flowLayoutPanel1.TabIndex = 28;
```

Programski kôd 4.4. Prikaz inicijalizacije *fowLayoutPaela* koji omotava tipke GUI-a unutar *Form1.Designer.cs*

Metode koje aktiviraju *event handleri* napisane su unutar *Form1.cs* datoteke, kao i svi ostali kodovi kojima se implementiraju funkcionalnosti Windows Forms aplikacije. Tipka *save* služi za spremanje sadržaja popunjenog obrasca. Pritiskom *save* tipke pokreće se *btnSave_Click()* metoda koja prvo stvara ime koristeći kombinaciju unosa HTTP metode i *endpointa* uklanjajući posebne znakove (engl. *special characters*) poput „/“. Ako već ne postoji, stvara tekstualnu datoteku s tim imenom. Prolazi i iščitava sve unesene vrijednosti unutar GUI-a te nakon toga poziva *SaveSerializedParams()* metodu koja ih pohranjuje u tekstualnu datoteku. Datoteka je smještena u mapu *Saved* unutar mape aplikacije, a vrijednosti su međusobno odvojene posebnim znakovima. Implementacija kôda za spremanje sadržaja popunjenih GUI polja prikazan je programskim kôdovima 4.5. i 4.6.

```
private void btnSave_Click(object sender, EventArgs e)
        {
            var str = txtEndpoint.Text;
            string name = new string((
from c in str where char.IsWhiteSpace(c) || char.IsLetterOrDigit(c) select c
                    ).ToArray());
            string path = Application.StartupPath + "\\Saved\\" + name +
comboHttpMethods.Text + ".txt";

            SaveSerializedParams(path);
        }
```

Programski kôd 4.5. Prikaz implementacije akcije pritiska na tipku *save* unutar *Form1.cs*

```
private void SaveSerializedParams(string path)
        {
            StreamWriter sw = new StreamWriter(path);
            sw.WriteLine(txtBaseUrl.Text + "|||");
            sw.WriteLine(txtEndpoint.Text + "|||");
            sw.WriteLine(comboHttpMethods.Text + "|||");
            sw.WriteLine(txtHeaders.Text + "|||");
            sw.WriteLine(txtBody.Text + "|||");
            sw.WriteLine(txtResponseCode.Text + "|||");
            sw.WriteLine(chkAuthorisationSettings.Checked + "|||");
            sw.Close();
        }
```

Programski kôd 4.6. Prikaz tijela *SaveSerializedParams* metode za spremanje parametara u datoteku

Spremljeni sadržaj obrasca grafičkog korisničkog sučelja moguće je nanovo učitati u alat korištenjem tipke *load*. Pritiskom tipke *load* pokreće se *btnLoad_Click()* metoda kojoj je prva zadaća određivanje datoteke. To čini s pomoću *openFileDialog* objekta koristeći *ShowDialog()* metodu koja otvara izbornik datoteka unutar *Saved* mape. Duplim klikom na datoteku korisnik odabire željenu datoteku te metoda nastavlja dalje s učitavanjem njenog sadržaja u alat. Iz toka datoteke (engl. *filestream*) metoda *LoadDeserializedParams()* izdvaja zasebne vrijednosti te njima pravilno popunjava elemente obrasca. Implementacija kôda za popunjavanje GUI polja sadržajem iz spremljenih datoteka prikazan je programskim kôdovima 4.7. i 4.8.

```
private void btnLoad_Click(object sender, EventArgs e)
        {
            var filePath = string.Empty;

            using (OpenFileDialog openFileDialog = new OpenFileDialog())
            {
                openFileDialog.InitialDirectory = Application.StartupPath +
"\\Saved\\";
                openFileDialog.Filter = "txt files (*.txt)|*.txt|All files
(*.*)|*.*";
                openFileDialog.FilterIndex = 2;
                openFileDialog.RestoreDirectory = true;

                if (openFileDialog.ShowDialog() == DialogResult.OK)
                {
                    //Get the path of specified file
                    filePath = openFileDialog.FileName;

                    //Read the contents of the file into a stream
                    var fileStream = openFileDialog.OpenFile();

                    LoadDeserializedParams(fileStream);
                }
            }
        }
```

Programski kôd 4.7. Prikaz implementacije akcije pritiska na tipku *load* unutar *Form1.cs*

```
private void LoadDeserializedParams(Stream fileStream)
        {
            using (StreamReader sw = new StreamReader(fileStream))
            {
                string text = sw.ReadToEnd();

                string[] separatingStrings = { "|||\r\n", "|||" };

                System.Console.WriteLine($"Original text: '{text}'");

                string[] words = text.Split(separatingStrings,
System.StringSplitOptions.None);


                txtBaseUrl.Text = words[0];
                txtEndpoint.Text = words[1];
                comboHttpMethods.Text = words[2];
                txtHeaders.Text = words[3];
                txtBody.Text = words[4];
                txtResponseCode.Text = words[5];
                chkAuthorisationSettings.Checked = bool.Parse(words[6]);
            }
        }
```

Programski kôd 4.8. Prikaz tijela *LoadDeserializedParams()* metode za učitavanje parametara iz datoteke

Središnja metoda razvijenog alata je *btnRun_Click()* metoda, čije je tijelo prikazano programskim kôdom 4.9. Njome se pokreću napisani testovi, šalju zahtjevi na poslužitelj i stvaraju testni izvještaji. Metoda prvo vrijednosti unesene u obrazac grafičkog korisničkog sučelja sprema u zasebne varijable. Zatim *body requesta* kodira u „*aplication/json*“ tip sadržaja s pomoću *StringContent()* metode. Unutar *try-catch* bloka šalje se zahtjev na server korištenjem *SendRequestAsync()* metode kojoj se kao parametre predaje klijent za izvršavanje *requesta, endpoint,* vrsta *requesta* i *body.* Sigurnosni koraci su dodani kako bi se osiguralo da se kôd dalje ne izvršava u slučaju da *response* na poslan *request* nije uspješno zaprimljen. S *if* petljom provjerava se je li zaprimljen *response* objekt te s *return()* završava s izvođenjem metode u slučaju da je *response* jednak *null*. Iz *responsea* se izvlači kôd stanja odgovora (engl. *response status code*) te se uspoređuje s očekivanim *response status codeom*. Ako je kôd zadovoljavajuć, alat pokazuje zelenu kvačicu, a u suprotnom crveni križić. Na kraju se pokreću metode za generiranje izvještaja.

Metoda *btnRun_Click()* pri svom izvođenju poziva još i druge metode poput *GethttpClient()* s pomoću koje dolazi do klijenta za izvršavanje zahtjeva. Za inicijalizaciju klijenta potreban je bazni URL, odnosno *host* URL poslužitelja. Njegovo stvaranje odvija se unutar *try-catch* bloka kako bi se u slučaju greške pri stvaranju uhvatila iznimka (engl. *exception*) čija se poruka potom u *catch* dijelu prikazuje s pomoću *MessageBox.Show()* metode. Osim baznog URL-a, klijentu se dodaje i autorizacijsko zaglavlje, ako je *authorization checkbox* na alatu označen. Na posljetku, metoda vraća stvorenog klijenta. Implementacija *GetHttpClient()* metode prikazana je programskim kodom 4.10.

Metoda *btnRun_Click()* poziva još i metode koje generiraju izvješća. Tijelo *GenerateTxtReport()* metode koja generira tekstualna izvješća prikazano je programskim kodom 4.11. Metoda kao parametre primaju naziv po kojem će nazvati datoteku izvješća te *response* koji je primljen od poslužitelja. Unutar *try-catch* bloka se ispisuju provedeni koraci testa slijedeći *gherkin* sintaksu. *NUnit* metoda *Assert()* provjerava točnost verifikacije *response status codea* te u slučaju nepodudaranja podiže *exception*. Poruka *exceptiona* pobliže objašnjava razlog nepodudaranja te se ispisuje u izvještaj i izvođenje se završava. U slučaju podudaranja *response status codea* s očekivanom vrijednosti, izvođenje se nastavlja te se na kraju ispiše da je test uspješno završen.

```
private async void btnRun_Click(object sender, EventArgs e)
        {
            string baseUrl = txtBaseUrl.Text;
            string endpoint = txtEndpoint.Text;
            string headers = txtHeaders.Text;
            string body = txtBody.Text;

            HttpClient client = GetHttpClient(baseUrl);

            StringContent content = new StringContent(body, Encoding.UTF8,
"application/json");
            HttpResponseMessage result = null;
            string resultContent = string.Empty;

            try
            {
                result = await SendRequestAsync(client, endpoint,
comboHttpMethods.Text, content);
                if (result == null) throw new Exception("Something went wrong.
Could not get the response.");

                resultContent = await result.Content.ReadAsStringAsync();
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message, "Error", MessageBoxButtons.OK,
MessageBoxIcon.Error);
                return;
            }

            if (txtResponseCode.Text == ((Int32)result.StatusCode).ToString())
            {
                lblResponseCodeCheck.Text = "✓";
                lblResponseCodeCheck.ForeColor = Color.Green;
                lblResponseCodeCheck.Visible = true;
            }
            else
            {
                lblResponseCodeCheck.Text = "X";
                lblResponseCodeCheck.ForeColor = Color.Red;
                lblResponseCodeCheck.Visible = true;
            }

            var str = txtEndpoint.Text;
            string name = new string((from c in str
                                      where char.IsWhiteSpace(c) ||
char.IsLetterOrDigit(c)
                                      select c
                   ).ToArray());

            GenerateTxtReport(name, result, resultContent);
            GenerateCsvReport(name, result, resultContent);
            GenerateHtmlReport(name, result, resultContent);
            }
```

Programski kôd 4.9. Prikaz tijela *btnRun_Click()* metode

```
private HttpClient GetHttpClient(string baseUri)
        {
            HttpClient client = new HttpClient();
            try
            {
                client.BaseAddress = new Uri(baseUri);
                client.Timeout = new TimeSpan(0, 2, 0);

                if (chkAuthorisationSettings.Checked)
                {
                    var filePath = string.Empty;
                    filePath = Application.StartupPath +
"\\Saved\\Authorization.txt";

                    using (StreamReader sw = new StreamReader(filePath))
                    {
                        string text = sw.ReadToEnd();
                        client.DefaultRequestHeaders.Add("Authorization", "Bearer
" + text);
                        Console.WriteLine(text);
                    }
                }
                client.DefaultRequestHeaders.Accept.Add(new
MediaTypeWithQualityHeaderValue("application/json"));
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message, "Error", MessageBoxButtons.OK,
MessageBoxIcon.Error);
            }
            return client;
        }
```

Programski kôd 4.10. Prikaz tijela *GetHttpClient()* metode

```
    private void GenerateTxtReport(string name, HttpResponseMessage result,
string resultContent)
        {
            StreamWriter sw = new StreamWriter(Application.StartupPath +
"\\ReportTxt\\" + name + comboHttpMethods.Text + "_REPORT.txt");
            try{
                sw.WriteLine("When I send " + comboHttpMethods.Text +
"request to " + txtBaseUrl.Text + txtEndpoint.Text + " endpoint");
                sw.WriteLine("With custom headers:" + txtHeaders.Text);
                sw.WriteLine("and body: ");
                sw.WriteLine(txtBody.Text);
                sw.WriteLine("I verify response code is: " +
txtResponseCode.Text);

                Assert.That(txtResponseCode.Text,
Is.EqualTo(((Int32)result.StatusCode).ToString()));
                sw.WriteLine("Test passed successfully");
            }
            catch (Exception ex)
            {
                sw.WriteLine(ex.Message);
            }
            sw.Close();}
```

Programski kôd 4.11. Prikaz tijela *GenerateTxtReport()* metode

5. TESTIRANJE PROGRAMSKOG RJEŠENJA

Ovo poglavlje sadrži testiranje i evaluaciju razvijenog programskog rješenja. Testiranje za svrhu ima donošenje procjene upotrebljivosti i funkcionalnosti programskog rješenja. Također, potrebno je uvjeriti se kako rješenje zadovoljava sve zadane zahtjeve te pruža pouzdano i učinkovito rješenje za automatiziranu izradu testnih izvještaja poslužitelja. Alat će biti manualno testiran tako što će se s pomoću njega napraviti API testovi stvarnog poslužitelja.

5.1. Opis stvarnog poslužitelja

Objekt testiranja na kojem će alat biti demonstriran stvarni je poslužitelj koji služi za upravljanje robotskim sustavom. Prije svega, sustav osigurava proces autentifikacije te samo ovlašteni korisnici imaju pristup određenim informacijama, stoga je obavezna prijava korisnika korisnkičkim imenom i lozinkom. Kada se potvrdi identitet korisnika, on dobiva odgovarajuć pristup resursima i ovlasti sustava. Neki od dostupnih resursa čine: prikaz liste robota, spajanje s robotom, prikaz detalja spojenog robota, izmjena imena spojenog robota itd. Korisnik resursima pristupa s pomoću *endpointa* na koje se šalju HTTP zahtjevi. Testirani *endpointi* su:

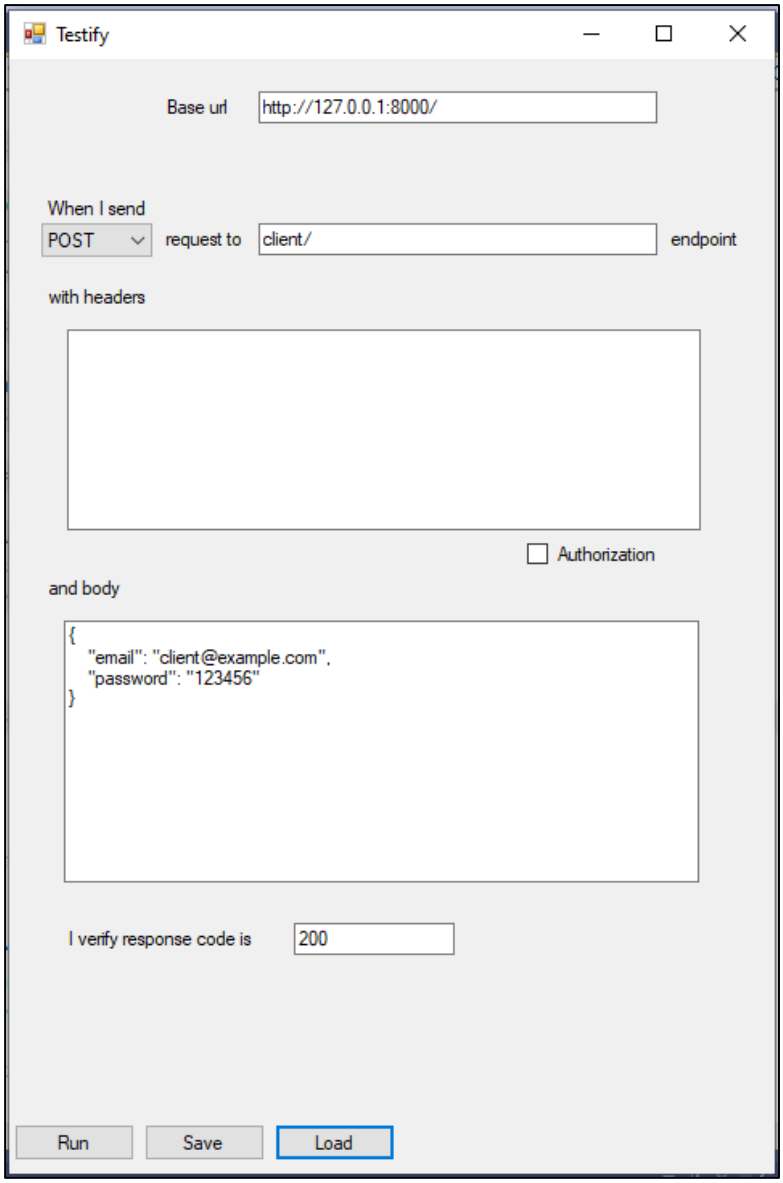
- /client/new – služi za stvaranje novog korisnika po predanim podacima
- /client – služi za prijavu stvorenog korisnika po predanim podacima
- /client/logout – služi za odjavu prijavljenog korisnika
- /client/detail/pk – služi za dohvaćanje, ažuriranje ili brisanje korisničkih podataka po primarnom ključu (engl. *primary key*)
- /client/robots – služi za ispis liste robota kojim korisnik može pristupiti
- /robot/new/client_id – služi za stvaranje novog robota kojim korisnik može upravljati
- /robot – služi za dohvaćanje podataka o prijavljenom robotu
- /robot/logout – služi za odjavu s prijavljenog robota
- /robot/detail/pk – služi za dohvaćanje, ažuriranje ili brisanje podataka o robotu

5.2. Dodavanje i upravljanje testnim scenarijima unutar razvijenog alata

Slijedeći dokumentaciju poslužitelja, za svaki će *endpoint* biti napisan test u kojem se šalje ispravan *request*. Očekivano je da će poslužitelj na tim testovima u *responseu* signalizirati uspješnu komunikaciju. Prvi korak kod testiranja alata jest unošenje podataka za stvaranje zahtjeva na poslužitelj. Podaci su ručno uneseni u grafičko korisničko sučelje. Radi verifikacije očekivanog odgovora poslužitelja dodaje se i očekivani *response status code*. Kôd 200 označava „Ok“,

odnosno da je komunikacija prošla uredno i da je poslužitelj uspješno primio *request*, prepoznao ga te na njega primjereno odgovorio. Svaki *response* dolazi sa svojim *response status codeom* što ga čini pogodnim objektom validacije.

Nakon dodavanja podataka u alat, provjereno je da se podaci mogu uspješno pohraniti za buduću upotrebu. To se provjerilo s pomoću tipki *save* i *load*. Pritiskom tipke *save* podaci su uspješno spremljeni u datoteku koja u svom nazivu nosi ime HTTP metode i *endpointa* te ju je po tom nazivu moguće učitati natrag u alat. Alat se nanovo pokreće kako bi se ispraznila unesena polja. Tipkom *load* otvara se *open file dialog* i odabire se datoteka te alat uspješno popunjava prazna polja, kao što je prikazano slikom 5.1.



Sl. 5.1. Prikaz GUI-a alata s poljima popunjenim s pomoću tipke *load*

5.3. Izrada testnih izvještaja stvarnog poslužitelja

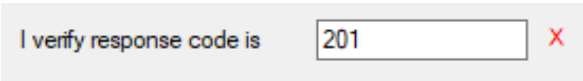
Na kraju se testirala funkcionalnost pokretanja testova i automatske izrade testnog izvješća. Nakon učitavanja spremljenog testa sa slike 5.1., on se pokrenuo tipkom *run*. Slikom 5.2. prikazana je indikacija na GUI-u alata da je test uspješno prošao, što potvrđuje i sadržaj stvorenog testnog izvješća prikazanog slikom 5.3. Još se ispitao i slučaj kada test ne prođe uspješno. Nakon promjene očekivanog *response codea* na alatu i pokretanjem testa GUI indicira kako je došlo do greške, kao što je prikazano slikom 5.4. Detaljnije pojašnjenje o pogrešnom *response codeu* dobiveno je u testnom izvješću koje je prikazano slikom 5.5. Istom metodom testirani su i ostali *endpointi* te su njihova izvješća pohranjena u *Report* mapi (Slika 5.4.).

Sl. 5.2. GUI alata daje indikaciju da je test uspješno prošao

```
When I send POST request to http://127.0.0.1:8000/client/ endpoint
With custom headers:
and body:
{
    "email": "client@example.com",
    "password": "123456"
}
I verify response code is: 200
Test passed successfully

Response:
{"refresh":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjo
icmVmcmVzaCIsImV4cCI6MTcyNDE3ODE4OCwiaWF0IjoxNzI0MDA1Mzg4LCJqdGkiOiI
3MTQ2OTlmMWJjMjc0ZTc3YmY3MDg1NzY4N2IwMWVmMCIsInVzZXJfaWQiOjF9.IgqOkj
C_5RXIQos6SYr2RBA36cmalHFCDrs8TWdhfDI","access":"eyJhbGciOiJIUzI1NiI
sInR5cCI6IkpXVCJ9.eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNzI0MDA1OT
g4LCJpYXQiOjE3MjQwMDUzODgsImp0aSI6IjczYTE1YjUzOWNiNzQ5NTg4NDRmOWU3OD
AyYjM2OWRmIiwidXNlcl9pZCI6MX0.R5tCkv3sE2RruNSQDTvpDMaDFkZ7N9tiRHwosf
8V3sg"}
```

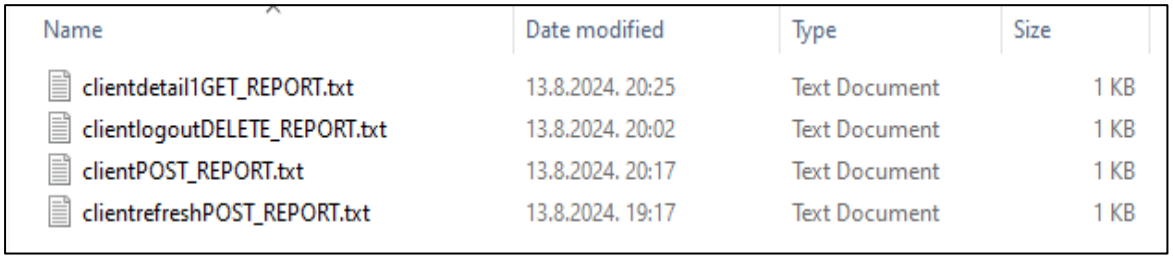
Sl. 5.3. Prikaz Windows Forms



Sl. 5.4. GUI alata daje indikaciju da test nije uspješno prošao

```
When I send POST request to http://127.0.0.1:8000/client/ endpoint
With custom headers:
and body:
{
    "email": "client@example.com",
    "password": "123456"
}
I verify response code is: 201
  Assert.That(txtResponseCode.Text,
  Is.EqualTo(((Int32)result.StatusCode).ToString()))
  String lengths are both 3. Strings differ at index 2.
  Expected: "201"
  But was:  "200"
  -------------^
```

Sl. 5.5. Prikaz Windows



Name	Date modified	Type	Size
clientdetail1GET_REPORT.txt	13.8.2024. 20:25	Text Document	1 KB
clientlogoutDELETE_REPORT.txt	13.8.2024. 20:02	Text Document	1 KB
clientPOST_REPORT.txt	13.8.2024. 20:17	Text Document	1 KB
clientrefreshPOST_REPORT.txt	13.8.2024. 19:17	Text Document	1 KB

Sl. 5.4. Prikaz datoteka testnih izvješća unutar mape

6. ZAKLJUČAK

U diplomskom radu razvijen je programski alat za automatiziranu izradu testnih izvještaja poslužitelja koja korisnicima omogućuje pokretanje i dokumentiranje testnih scenarija za REST API-je. S pomoću ovog alata omogućeno je efikasno održavanje željene razine pouzdanosti poslužitelja jer pruža temelje za automatsko i kontinuirano provođenje testova. U iterativnim razvojnim procesima gdje se poslužiteljske aplikacije često ažuriraju, kontinuirano testiranje je neophodno kako bi se osigurala stabilnost i pouzdanost svake nove iteracije. Taj proces može se značajno pojednostaviti tako što se s pomoću alata automatski pokrenu svi unaprijed definirani scenariji testiranja nakon svake promjene u kôdu, što osigurava da nove značajke i popravci ne uzrokuju nove pogreške te također oslobađa dragocjeno vrijeme programerima i testerima. Razvijen je korištenjem *C#* programskog jezika i Windows Forms obrasca u svrhu korisničkog sučelja. Pisanje scenarija testova korištenjem Gherkin sintakse omogućilo je njihovu lakšu čitljivost i održivost.

Alat uspješno izvršava svoju primarnu zadaću pružanja jednostavne i intuitivne podrške testerima pri izradi izvještaja kod testiranja poslužiteljskih API-ja, međutim ovo rješenje moguće je dalje unaprijediti dodatnim nadogradnjama. Na primjer, moguće je implementirati funkcionalnost integracije s cjevovodima kontinuirane integracije i isporuke, odnosno CI/CD cjevovodima. To bi osiguralo da svaka predaja (engl. *commit*) kôda na udaljeni repozitorij (eng. *remote repository*) automatski pokreće ciklus testiranja pružajući trenutne povratne informacije programerima. Nadalje, integracija podrške za SOAP protokol proširila bi doseg alata te povećala njegovu svestranost i upotrebljivost. Omogućila bi korištenje alata nad SOAP poslužiteljima i učinila ga sveobuhvatnijim rješenjem za testiranje web usluga što bi alat učinio privlačnijim organizacijama koje koriste i REST-ful i SOAP usluge. Navedene nadogradnje dodatno bi povećale stupanj automatizacije alata i lakšu integraciju u razvojni proces programskih sustava u kojima poslužitelj ima središnju ulogu osiguravajući dosljednost u različitim protokolima testiranja web usluga.

Korištenje alata testirano je na stvarnom poslužitelju pokretanjem raznih testnih scenarija za različite *endpointe* poslužitelja. Testiranje je zadovoljavajućim rezultatima potvrdilo sposobnost alata da automatski izrađuje testna izvješća za poslužiteljske aplikacije s ciljem smanjenja rizika od postavljanja neispravnog kôda na korištenje. Kako bi se alat dodatno poboljšao, moguće je dodavanje novih značajki koje bi mogle poboljšati njegovu praktičnost i učinkovitost za korisnike. Dodatno, takva ažuriranja bi podržala nastavak razvoja i istraživanja u području automatiziranog testiranja poslužiteljskih aplikacija.

LITERATURA

[1] What does it mean to be an SDET (Software Development Engineer in Test) in Exchange?, Microsoft, 01.07.2019., https://techcommunity.microsoft.com/t5/exchange-team-blog/what-does-it-mean-to-be-an-sdet-software-development-engineer-in/ba-p/610515 [22.06.2024.]

[2] M. Manic, 7 types of testing every QA engineer must know, Pontistechnology, 12.04.2023., https://pontistechnology.com/7-types-of-testing-every-qa-engineer-must-know/?cn-reloaded=1 [22.06.2024.]

[3] H. Kansara, SDET vs QA - A Comprehensive Guide To The Key Differences, Maruti techlabs, 21.7.2024., https://marutitech.com/differences-between-sdet-and-qa/ [22.06.2024.]

[4] D. Borcherding, SDET vs QA Tester: Key Differences and Similarities, Taazaa, 20.03.2024., https://www.taazaa.com/sdet-vs-qa-tester/ [22.06.2024.]

[5] B. Lutkevich, Waterfall model, Techtarger, https://www.techtarget.com/searchsoftwarequality/definition/waterfall-model [22.06.2024]

[6] Ž. Gavrić, Pregled metodologija i tehnika za razvoj softwera za generisanje asp.net web formi kao primjera dana-driven programiranja, Slobomir P Univerzitet Fakultet za informacione tehnologije 2012, https://fit.spu.ba/wp-content/uploads/sites/2/2018/03/DiplomskiRadZeljkoGavric.pdf [22.06.2024.]

[7] SDLC – V – model, Tutorialspoint, https://www.tutorialspoint.com/sdlc/sdlc_v_model.htm [22.06.2024]

[8] SDLC – V – model – Software Engineering, Geeks for geeks, 06.03.2024., https://www.geeksforgeeks.org/software-engineering-sdlc-v-model/ [22.06.2024.]

[9] Hrvatski jezični portal, Agilan definicija, https://hjp.znanje.hr/index.php?show=search_by_id&id=f19lUA%3D%3D&keyword=agilan [22.06.2024.]

[10] Agile Model, Javatpoint, https://www.javatpoint.com/software-engineering-agile-model [22.06.2024.]

[11] J. Jason, Agile and scrum methodology, Medium, 19.03.2022., https://jaenhosjsaon.medium.com/agile-and-scrum-methodology-32a14ba3746f [22.06.2024.]

[12] H. Ashtari, Black Box vs. White Box Testing: Understanding 3 Key Differences, Spiceworks, 29.09.20222., https://www.spiceworks.com/tech/devops/articles/black-box-vs-white-box-testing/ [22.06.2024.]

[13] J. Bannister, Why developers don't write unit tests, Linkedin, 12.03.2020., https://www.linkedin.com/pulse/why-developers-dont-write-unit-tests-justin-bannister/ [22.06.2024.]

[14] J.T. King, I Pity The Fool Who Doesn't Write Unit Tests , Coding horror, 20.07.2006., https://blog.codinghorror.com/i-pity-the-fool-who-doesnt-write-unit-tests/ [22.06.2024.]

[15] What is a server?, Paessler, https://www.paessler.com/it-explained/server#:~:text=A%20server%20is%20a%20computer,as%20clients%2C%20over%20a%20network. [23.06.2024.]

[16] V. Kanade, What is a server? Definition, types, and features, Spiceworks, 17.03.2023., https://www.spiceworks.com/tech/tech-general/articles/what-is-a-server/ [23.06.2024.]

[17] What is API testing?, Smartbear, https://smartbear.com/learn/api-testing/what-is-api-testing/ [23.06.2024.]

[18] Client server testing | What it is, advantages & challenges, testsigma, 21.03.2024., https://testsigma.com/blog/client-server-testing/ [23.06.2024.]

[19] About Postman, Postman, https://www.postman.com/company/about-postman/ [23.06.2024.]

[20] SDET Tools For API Testing: Postman Vs. SoapUI, Wedevx, 29.02.2024, https://www.wedevx.co/blog/sdet-tools-for-api-testing-postman-vs-soapui/ [25.06.2024.]

[21] H. Tran, REST API & WebServices Testing with Katalon Studio, Tooldsqa, 30.07.2021., https://toolsqa.com/katalon-studio/rest-api-webservices-testing-with-katalon-studio/ [25.06.2024.]

[22] Getting started with Apache JMeter, Apache JMeter, https://jmeter.apache.org/usermanual/get-started.html [25.06.2024.]

[23] REST-assured, Rest-assured, https://rest-assured.io/ [1.07.2024.]

[24] REST-Assured API test automation, Linkedin, 11.08.2023., https://www.linkedin.com/pulse/rest-assured-api-test-automation-vervesquare#:~:text=REST%2DAssured%20is%20an%20open,project%20management%20and%20build%20tool.https://rest-assured.io/ [01.07.2024.]

[25] API Testing made simple!, Karate Labs, https://www.karatelabs.io/ [1.07.2024.]

[26] Visual studio 2022, Microsoft, https://visualstudio.microsoft.com/ [23.06.2024.]

[27] Latest Visual Studio 2022 with GitHub Copilot woven in, Microsoft, https://visualstudio.microsoft.com/ [23.06.2024.]

[28] RestSharp: Simple REST and HTTP API Client for .Net, RestSharp, https://restsharp.dev/ [23.06.2024.]

[29] A. Zimarev, RestSharp – Simple .NET REST Client, GitHub, https://github.com/restsharp/RestSharp [23.06.2024.]

[30] What is NUnit?, NUnit, https://nunit.org/ [23.06.2024.]

[31] O. Terje, NUnit 4 Framework, GitHub, https://github.com/nunit/nunit [23.06.2024.]

[32] What is Gherkin, Cucumber, https://cucumber.io/docs/guides/overview/ [01.07.2024.]

SAŽETAK

Spoj poslužitelja i korisničkih aplikacija postao je standard u vođenju i automatizaciji mnogih poslovnih procesa čime se povećala važnost održavanja ispravnosti tih sustava. U ovom radu pojašnjeni su pristupi testiranja unutar različitih metodologija razvoja programskih rješenja. Proučava se prilagodba metodologije testiranja sve većoj potražnji za brzim i hitrim inkrementalnim promjenama kôda sustava. Cilj rada je razviti alat s grafičkim korisničkim sučeljem za stvaranje i pokretanje automatiziranih testnih scenarija za API poslužitelja te kreiranje izvještaja testiranja. U svrhu razvoja vlastitog alata proučeni su neki od najpoznatijih postojećih alata: Postman, Soap UI, Katalon studio, Apache Jmeter, Rest Assured i Karate DSL. Neke od ključnih funkcionalnosti alata su pisanje testnih scenarija koristeći Gherkin sintaksu, automatsko pokretanje testova, spremanje i učitavanje testova te ispis testnih izvješća. Uz pomoć pri njihovu stvaranju, korisnici mogu vršiti kontinuirano testiranje poslužitelja i automatsku izradu testnih izvješća. Alat se testirao nad stvarnim poslužiteljem te je uspješno izvršio dane testne scenarije i generirao popratna izvješća.

Ključne riječi: alati za izradu testnih izvješća, metodologije razvoja programskih rješenja, testiranje poslužiteljskog API-ja, Windows Forms radni okvir

ABSTRACT

Title: Software for server test report generetion

The combination of servers and user applications has become a standard in managing and automating many business processes, which has increased the importance of maintaining the correctness of these systems. In this paper, testing approaches within different software development methodologies are explained. Adaptation of the testing methodology to the increasing demand for quick and rapid incremental changes to the system code is studied. The aim of the work is to develop a tool with a graphical user interface for creating and running automated test scenarios for the API server and creating test reports. In order to develop our own tool, some of the most famous existing tools were studied: Postman, Soap UI, Katalon studio, Apache Jmeter, Rest Assured and Karate DSL. Some of the key functionalities of the tool are writing test scenarios using Gherkin syntax, automatically running tests, saving and loading tests, and printing test reports. With them, users can perform continuous server testing and automatically generate test reports. The tool was tested on a real server and successfully executed the given test scenarios and generated accompanying reports.

Ključne riječi: methodology of development of software solutions, server API testing, software for creating test reports, Windows Forms framework

PRILOZI

Uz ovaj rad priložen je CD s programskim kodom te izvješća generirana testiranjem razvijenog alata.