Osijek, 2022.

ELPROS
REGIONALNI CENTAR KOMPETENTNOSTI
ELEKTROTEHNIČKA I PROMETNA ŠKOLA OSIJEK

Bojana Stojanović

OSIGURAVANJE KVALITETE

(ENGL. QUALITY ASSURANCE)

Priručnik za polaznike

ESF UČINKOVITI LJUDSKI POTENCIJALI

EUROPSKI STRUKTURNI I INVESTICIJSKI FONDOVI

IMPRESUM

Autorica: **Bojana Stojanović** (ALGEBRA d.o.o.)

Urednik i stručnjak za metodologiju izrade nastavnih materijala: **Sara Sharifi** (ALGEBRA d.o.o.)

Grafičko oblikovanje: **BARREK d.o.o.** (ALGEBRA d.o.o.)

Lektorica: **Sara Sharifi** (ALGEBRA d.o.o.)

Naslov: **Osiguravanje kvalitete (engl. *Quality Assurance*)** (priručnik za polaznike u programu obrazovanja Stručnjak u području osiguravanja kvalitete (engl. *Quality Assurance*) – obrazovanje odraslih, razina 5 HKO)

Fotografije: Freepik, Siniša Kovačić (ALGEBRA d.o.o.), snimke zaslona iz programa Jira i platforme BlazeMeter (rad autora)

Naručitelj i nositelj prava: **Elektrotehnička i prometna škola Osijek**

Odgovorna osoba: **ravnatelj Antun Kovačić, dipl. ing. el.**

Mjesto i godina izdanja: **Osijek, prosinac 2022.**

Regionalni centar kompetentnosti ELPROS

Osijek, 2022.

OSIGURAVANJE KVALITETE

(engl. Quality Assurance)

Priručnik za polaznike

Algebra d.o.o.

Osijek, 2022.

Projekt je sufinancirala Europska unija iz Europskog socijalnog fonda.

Sadržaj:

1. Testiranje u životnom ciklusu razvoja softvera

U OVOM POGLAVLJU NAUČIT ĆETE:

- osnovne pojmove vezane uz softversko testiranje
- modele životnog ciklusa razvoja softvera
- faze testiranja softvera
- važnost kvalitete softverskog proizvoda
- kako kvaliteta testiranja utječe na kvalitetu softverskog proizvoda.

1.1. Koncepti u testiranju softvera

Kako biste što bolje razumjeli testiranje softvera, aplikacija ili hardvera, potrebno je objasniti osnovne pojmove vezane uz sam proces testiranja. U ovom poglavlju predstavljamo najvažnije pojmove koji se u tom procesu javljaju.

1.1.1. Osnovni pojmovi

Testiranje softvera je proces kojim se utvrđuje ispravnost, cjelovitost i kvaliteta softvera. Riječ je o aktivnosti koja prikazuje odgovara li aktualan rezultat očekivanom rezultatu odnosno, postoje li u softverskom sustavu kakve anomalije.

U testiranju softvera **anomalije** predstavljaju podatke ili događaje koji su drugačiji od očekivanih rezultata ili ponašanja. Mogu biti posljedica greške u kodu programa ili nečeg što je van kontrole programera, poput hardverskih problema. Na primjer, ako se očekuje da softver ispiše poruku ili izvrši određeni zadatak, a on to ne čini, navedeno se smatra anomalijom. Anomalije se mogu pronaći tijekom testiranja ili tijekom korištenja softverskog proizvoda. Neke od anomalija su: pogreška (engl. *Bug*), nedostatak (engl. *Defect*), odstupanje (engl. *Deviation*), kvar (engl. *Error*), zastajanje/ispad (engl. *Failure*), incident (engl. *Incident*) ili propust (engl. *Problem*).

> **Pogreška, propust i nedostatak** predstavlja „pogrešku" komponente ili sustava koja može uzrokovati neizvršavanje tražene funkcije komponente ili sustava, npr. neispravna linija ili definicija podataka.
>
> **Odstupanje od specifikacije softvera** označava događaj čije pojavljivanje zahtijeva istragu i djelovanje. Kvar predstavlja „ljudsko djelovanje" koje proizvodi neispravan rezultat. Ako se kvar utvrdi, potrebno je istaknuti kako je ljudski faktor presudan tj. kako do kvara dolazi ljudskom pogreškom.
>
> **Zatajenje** predstavlja odstupanje komponente ili sustava od zahtijevanog ponašanja (primjerice, kada se aplikacija iz nepoznatih razloga ne pokrene).

Testiranje provodi kvalificirani stručnjak, odnosno tester. **Testeri** su stručnjaci koji pružaju usluge testiranja softvera. Njihova je zadaća testirati radi li softver ispravno te zadovoljava li zahtjeve funkcionalnosti i kvalitete. Testeri mogu testirati softver pomoću

automatiziranih i ručnih testova, osigurati da svi elementi softvera rade ispravno i bez problema, identificirati problematična područja softvera i dati pregled kvalitete softvera.

Osiguranje kvalitete (engl. *Quality assurance*, QA) izraz je koji se koristi za opisivanje sustavnih napora kojima se osigurava da isporučeni proizvod ispunjava ugovorna i druga dogovorena očekivanja naručitelja softvera, u pogledu učinka, dizajna, pouzdanosti i mogućnosti održavanja.

Osiguranje kvalitete usmjereno je na proces razvoja i testiranja softvera, a ne samo na proizvod. Cilj je osigurati da se svi procesi slijede ispravno. Osiguranje kvalitete treba provoditi kroz faze planiranja, dizajna, razvoja i testiranja softvera. Ono se može izvesti provođenjem statičke analize koda, ručnih testova i automatiziranih testova. Osiguranje kvalitete također treba uključivati testiranje korisničkog prihvaćanja i testiranje performansi, koje ćemo kasnije objasniti u nastavku priručnika. Osiguranje kvalitete treba provoditi redovito kako bi se osiguralo da softverski proizvod zadovoljava zahtjeve krajnjih korisnika i da nema nedostataka.

Kada govorimo o osiguranju kvalitete, važno je spomenuti dva temeljna ciklusa - životni ciklus razvoja softvera i životni ciklus testiranja softvera.

Životni ciklus testiranja softvera (engl. *Software Testing Life Cycle,* STLC) sastoji se od niza aktivnosti koje testeri provode kako bi testirali softverski proizvod. Iako STLC **koristi** izraz "testiranje", on ne uključuje samo testere, već se u nekim slučajevima uključuju i programeri.

Životni ciklus razvoja softvera (engl. *Software Development Life Cycle**,*** SDLC) je slijed aktivnosti koje provode programeri kako bi dizajnirali i razvili visokokvalitetan softver. **SDLC** predstavlja metodu sustavnog razvoja načina softvera, čime se osigurava njegova kvaliteta, ali i povećava vjerojatnost dovršenja projekta u roku. Životni ciklus razvoja softvera sastoji se od niza faza koje svi sudionici u razvoju softvera trebaju slijediti kako bi se razvio kvalitetan softver.

Životni ciklus razvoja softvera i životni ciklus testiranja softvera detaljno su opisani u poglavlju Modeli životnog ciklusa razvoja softvera.

Među osnovne pojmove u testiranju ubrajamo i **testni slučaj** i **testne scenarije. Testni slučaj** je skup aktivnosti koje se provode kako bi se ispitalo je li softver isporučen prema očekivanjima, odnosno ispunjava li zahtjeve koje su postavljeni. Testni slučaj može biti konkretan scenarij koji se temelji na zahtjevima ili može biti pokretanje određenih funkcija i procesa kako bi se odredilo ispunjavaju li zadane uvjete. Testni slučaj također pomaže u određivanju koliko je softver kvalitetan i koliko ga je moguće poboljšati.

Primjer testnog slučaja:

Test slučaj id: TC-001
Naziv testnog slučaja: Verifikacija uspješne prijave (*login*)
Zahtjevi: Korisnički račun je već kreiran

Testni koraci:

1. otvoriti stranicu za prijavu
2. unijeti ispravno korisničko ime i lozinku
3. kliknuti na gumb za prijavu.

Očekivani rezultat: Korisnik je uspješno prijavljen i usmjeren na početnu stranicu.

Test slučaj ID: TC-002
Naziv testnog slučaja: Verifikacija neuspješne prijave
Zahtjevi: Nema ih

Testni koraci:

1. otvoriti stranicu za prijavu
2. unijeti neispravno korisničko ime i lozinku

3. kliknuti na gumb za prijavu.

Očekivani rezultat: Mora biti prikazana poruka o pogrešci koja naznačuje da je prijava nevaljana.

Testni slučaj: TC-003
Naziv testnog slučaja: Verifikacija prijave bez unosa u polja prijave
Zahtjevi: Nema ih

Testni koraci:

1. otvoriti stranicu za prijavu
2. ostaviti prazna polja gdje se unosi korisničko ime i lozinka
3. kliknuti na gumb za prijavu.

Očekivani rezultat: Mora biti prikazana poruka o pogrešci koja naznačuje da su polja ostavljena prazna.

Testni slučaj: TC-004
Naziv testnog slučaja: Verifikacija prijave uz nepotpuna ispunjena polja
Zahtjevi: Nema ih

Testni koraci:

1. otvoriti stranicu za prijavu
2. onijeti samo korisničko ime ili samo lozinku
3. kliknuti na gumb za prijavu.

Očekivani rezultat: Mora biti prikazana poruka o pogrešci koja naznačuje da oba dva polja moraju biti ispunjena kako bi se prijava mogla ostvariti.

Testni scenarij je svaka funkcionalnost koja se može testirati. Također se naziva testni uvjet ili testna mogućnost. Kao tester, možete se staviti na mjesto krajnjeg korisnika i

pokušati razumjeti scenarije upotrebe aplikacije koja se testira. Testiranje scenarija je varijanta testiranja koja koristi testne scenarije. Scenariji pomažu u lakšem načinu testiranja kompliciranijih sustava.

Testni scenariji se kreiraju iz sljedećih razloga:

- osiguravaju potpunu testnu pokrivenost softverskog proizvoda
- osiguravaju da softver radi za najčešće slučajeve upotrebe
- služe kao brzi alat za određivanje radnog napora testiranja i prema tome kreiraju prijedlog za klijenta ili organiziraju radnu snagu
- pomažu u određivanju najvažnijih *end-to-end* transakcija ili stvarne upotrebe softverskih aplikacija.

Svaki testni scenarij treba biti povezan s najmanje jednim zahtjevom ili **korisničkom pričom** (engl*. User story*) koja kratko objašnjava funkcionalnosti sustava koji se testira u skladu s metodologijom projekta. Prije kreiranja testnog scenarija koji provjerava više zahtjeva odjednom, treba provjeriti postoji li testni scenarij koji provjerava taj zahtjev posebno. Preporučljivo je izbjegavati stvaranje prekompliciranih testnih scenarija koji obuhvaćaju više zahtjeva. Broj scenarija može biti velik, a skupo ih je sve pokrenuti. Na temelju prioriteta naručitelja softvera pokreću se samo odabrani testni scenariji.

Primjer korisničke priče:

Kao korisnik, želim se moći prijaviti na stranicu kako bih mogao pristupiti svom osobnom profilu i obavljati radnje kao što su objavljivanje sadržaja, komentiranje i pregledavanje mog profila. Da bih to postigao, morat ću unijeti svoje korisničko ime i lozinku na stranici za prijavu. Ako su moji podaci za prijavu ispravni, sustav bi mi trebao omogućiti pristup mom računu. Ako su moji podaci za prijavu netočni, sustav bi trebao prikazati poruku o pogrešci i zatražiti od mene da pokušam ponovno.

Primjer testnog scenarija:

Kako bismo bolje objasnili testne scenarije i njihove značaje u testiranju softvera, uzet ćemo za primjer opisanu korisničku priču.

Testni Scenarij 1: Pogledati funkcionalnost prijave (*login*).

Na ovom primjeru ćemo prikazati razliku između testnog scenarija i testnih slučajeva. Specifični testni slučajevi za ovaj testni scenarij bili bi:

- provjera ponašanja sustava kada se unese važeće korisničko ime i lozinka
- provjera ponašanja sustava kada se unese nevažeće korisničko ime i važeća lozinka
- provjera ponašanje sustava kada se unese važeće korisničko ime i nevažeća lozinka
- provjera ponašanja sustava kada se unese nevažeće korisničko ime i nevažeća lozinka
- provjera ponašanja sustava kada se korisničko ime i lozinka ostavljaju prazni i kada je prijava unesena.

1.1.2. Vrste testiranja softvera

Testiranje se može odvijati na tri različite razine - **jedinično testiranje, integracijsko testiranje i testiranje sustava. Jedinično testiranje** koristi se za testiranje ispravnosti rada jedne funkcije programa. **Testiranje integracije** koristi se za provjeru interakcije između komponenti sustava. Preduvjet mu je da je testiranje jedinice dovršeno na svim komponentama koje čine sustav. **Testiranje sustava** koristi se za provjeru i validaciju ponašanja cijelog sustava u odnosu na izvorne ciljeve sustava. Ove razine detaljno su opisane u poglavlju Faze/Nivoi testiranja u softverskom projektu s obzirom na model razvoja i životni ciklus softvera.

Testiranje se obično klasificira u **tri kategorije:**

- funkcionalno testiranje (engl*. Functional Testing*)
- nefunkcionalno testiranje ili testiranje performansi (engl*. Non-Functional testing* ili *Performance testing*)
- održavanje (engl*. Regression and Maintenance*).

Funkcionalno testiranje provjerava obavlja li sustav ili komponenta ispravno svoju predviđenu funkciju. Postoji nekoliko različitih vrsta funkcionalnog testiranja, poput jediničnog testiranja, *Smoke testa*, integracijskog testiranja.

Nefunkcionalno testiranje fokusira se na izvedbu i druge nefunkcionalne aspekte sustava ili komponente. Nefunkcionalno testiranje bavi se time kako se sustav ponaša i radi, a ne time što radi. Neki primjeri nefunkcionalnog testiranja uključuju testiranje performansi, skalabilnost, test izdržljivosti, upotrebljivost.

Funkcionalno i nefunkcionalno testiranje mogu biti izvršeni **manualno (ručno)** ili **automatizirano**. Za manualno testiranje nije potrebno poznavanje programskih jezika, dok za automatizirano jest kako bi tester mogao izraditi skripte pomoću kojih se izvodi testiranje.

Kod određivanja alata koji će se koristiti kod automatiziranog testiranja potrebno je:

- ispitati trenutni proces testiranja i odrediti gdje treba biti prilagođen za korištenje automatiziranih alata za testiranje.
- procijeniti spremnost da se promijene trenutni načini izvođenja testova
- uključiti dio tima koji će koristiti alat za pomoć u dizajniranju automatiziranog postupka testiranja
- napraviti skup kriterija ocjenjivanja za funkcije koje se žele koristite kao alat za automatsko testiranje. Ti kriteriji mogu uključivati sljedeće:
    - ponovljivost testa
    - kritičnost/rizik aplikacije

- operativnu jednostavnost
- jednostavnost automatizacije
- razinu dokumentiranosti funkcije (zahtjevi, itd.).

Održavanje je proces mijenjanja, modificiranja ili prelaska na novu verziju softvera kako bi se udovoljilo zahtjevima naručitelja softvera. Održavanje se radi nakon što je softverski proizvod pušten u produkcijsko okruženje.

1.2. Testiranje u životnom ciklusu razvoja softvera

Ključan element u razvoju softvera je i razumijevanje njegovog životnog ciklusa. Kako bismo što bolje razumjeli **razvoj životnog ciklusa softvera** potrebno ga je odvojiti u dvije kategorije - sekvencijalni i iterativni inkrementalni razvojni model koje ćemo opisati u sljedećem dijelu.

1.2.1. Model životnog ciklusa razvoja softvera

Model životnog ciklusa razvoja softvera opisuje vrste aktivnosti koje se izvode u svakoj fazi projekta **razvoja softvera** i kako su te aktivnosti logično i kronološki povezane jedna s drugom. Postoji niz različitih modela životnog ciklusa razvoja softvera, od kojih svaki zahtijeva različite pristupe testiranju. Uobičajeni životni ciklus razvoja softvera spada u jednu od sljedećih kategorija:

- **sekvencijalni razvojni model** (engl. *Sequential development mode*)
- **iterativni i inkrementalni razvojni model** (engl. *Iterative and incremental development mode*).

Faze koje su prisutne u gotovo svakom **razvoju** softvera su:

1. planiranje
2. analiza zahtjeva

3. dizajn arhitekture softvera
4. implementacija
5. testiranje i integracija
6. održavanje.

Na samom početku životnog ciklusa razvoja softvera su **faza planiranja** i **faza analize** zahtjeva. Ove faze izuzetno su važne jer predstavljaju temelj za razvoj ostalih faza. One podrazumijevaju pripremne radnje kao što su definiranje korisničkih zahtjeva, identificiranje neophodnih funkcionalnosti, planiranje troškova, planiranje rizika, definiranja snaga i slabosti sustava te ukupnih mogućnosti sustava. Ukratko, tijekom faze planiranja i faze analize zahtjeva nastoji se utvrditi ukupni opseg projekta, što podrazumijeva ekonomske, operativne i kadrovske čimbenike, kao i jasno definirane vremenske rokove.

Nakon planiranja i kompletne analize svih zahtjeva, slijedi **faza dizajniranja** arhitekture softvera. U ovoj se fazi pripremaju dokumenti za dizajn sustava prema dokumentima dobivenim iz prethodne dvije faze.

Ti dokumenti najčešće sadrže sljedeće informacije:

- definicije funkcionalnosti sustava
- odnosi i ovisnost među različitim dijelovima sustava
- definiranje baze podataka s njenim ključnim elementima
- tehničke specifikacije.

Faza dizajniranja postaje temelj za sljedeće faze koje slijede u životnom ciklusu razvoja softvera, a to je faza **implementacije, odnosno razvoja softvera**. U ovoj fazi počinje razvoj programa odnosno pisanje koda odabranim programskim jezikom. Pisanje koda podijeljeno je na nekoliko modula, odnosno manjih zadataka koji se zatim dodjeljuju timu za razvoj. Tim za razvoj čine programeri. Tijekom ove faze programeri koriste različite alate za pisanje koda kao što su kompajleri, interpreteri, programi za prepoznavanje

grešaka, itd. **Faza programiranja** također predstavlja fazu u životnom ciklusu razvoja softvera koja spada pod fazu implementacije i za nju su zaduženi programeri.

Sljedeća faza o kojoj će biti više riječi u sljedećim poglavljima je **testiranje**. Ova faza započinje kada je dovršen odnosno implementiran kompletan softver ili dijelovi softvera, ovisno o metodi koja se koristi prilikom razvoja softvera. Tijekom testiranja nastoje se utvrditi greške i nedostaci u radu softvera koji se zatim prijavljuju programerima kako bi ih ispravili. Ovaj postupak ponavlja se tako dugo dok nije istestiran cijeli sustav i dok nisu uklonjene sve greške. Faza testiranja završava tek kada se testovima prihvaćanja (engl. *Acceptance Testing*) potvrdi kako je sustav stabilan i radi u skladu s poslovnim potrebama i specifikacijama. Nakon potvrde da je sustav stabilan i spreman za rad, slijedi integracija sustava u proizvodno okruženje te njegovo svakodnevno korištenje.

Posljednja faza je **održavanje**. Ona služi rješavanju različitih problema koji se pojavljuju prilikom korištenja sustava. Važno je napomenuti da se radi o produkcijskom okruženju koji dolazi nakon testne faze razvoja. Osim toga, može se reći kako ova faza također pruža mogućnost poboljšanja funkcionalnosti ili nadogradnje sustava prema različitim potrebama naručitelja sustava. Važno je organizirati metodologiju testiranja u manjim dijelovima, kao što su testni slučajevi koje ćemo opisati u sljedećem odlomku.

Drugi važan ciklus je **Životni ciklus testiranja softvera (STLC)**. On predstavlja niz aktivnosti koje se provode **radi testiranja softvera**. Suprotno uvriježenom mišljenju, testiranje softvera nije samo jedna aktivnost. Sastoji se od niza aktivnosti koje se metodološki provode kako bi se pomoglo u realizaciji softverskog proizvoda. U STLC-u sudjeluje testni tim, dok u SDLC-u sudjeluje cijeli projektni tim zadužen za realizaciju softverskog proizvoda.

Slika 1.1. Prikaz softverskog testnog životnog ciklusa kroz faze
(preuzeto s https://www.bugraptors.com/blog/software-testing-life-cycle)

Faze koje su prisutne u gotovo svakom **testiranju** softvera su:

1. analiza zahtjeva
2. planiranje testiranja
3. razvoj testnog slučaja
4. postavljanje testne okoline
5. izvršenje testa
6. zatvaranje testnog ciklusa.

1. ANALIZA ZAHTJEVA

Tijekom ove faze, testni tim proučava zahtjeve kako bi se odredio tijek testiranja. Tim za osiguranje kvalitete može komunicirati s različitim dioničarima (klijent, poslovni analitičar,

tehnički voditelji, arhitekti sustava itd.) kako bi detaljno razumjeli zahtjeve. Zahtjevi mogu biti **funkcionalni** (definiranje onoga što softver mora raditi) ili **nefunkcionalni** (definiranje performansi sustava/dostupnosti sigurnosti). Izvedivost automatizacije za dani projekt testiranja također se radi u ovoj fazi.

Aktivnosti:

- odrediti vrste testova koje treba izvesti
- prikupiti detalje o prioritetima testiranja i fokusu
- pripremiti matricu sljedivosti zahtjeva (engl. *Requirement Traceability Matrix,* RTM[1])
- identificirati pojedinosti testnog okruženja u kojem se testiranje treba provesti
- analizirati izvedivosti automatizacije (ako je potrebna).

Rezultati:

- matrica sljedivosti zahtjeva
- izvješće o izvedivosti automatizacije (ako je primjenjivo).

2. PLANIRANJE TESTIRANJA

U ovoj fazi određuje se procjena napora i troškova za projekt te će pripremiti i finalizirati plan testiranja. U ovoj fazi utvrđuje se i **strategija testiranja**, dokument koji opisuje tehniku testiranja koja će se koristiti u životnom ciklusu razvoja softvera i potvrđuje vrste ili razine testiranja koje će se provoditi na proizvodu.

Aktivnosti:

- pripremiti testni plan/strateški dokument za različite vrste testiranja
- odabrati testni alat
- procijeniti vrijeme koje će se uložiti u testiranje

[1] Matrica sljedivosti zahtjeva je dokument koji pokazuje odnos između zahtjeva i drugih artefakata. Koristi se za dokazivanje da su zahtjevi ispunjeni i obično dokumentira zahtjeve, testove, rezultate testova i probleme.

- planirati resurse i određivanje uloga i odgovornosti.

Rezultati:

- plan testiranja/strateški dokument
- dokument o vremenu utrošenom za testiranje.

3. RAZVOJ TESTNOG SLUČAJA

Ova faza uključuje stvaranje, provjeru i preradu testnih slučajeva i testnih skripti. Testni podaci se identificiraju, kreiraju i pregledavaju, a zatim ponovno obrađuju.

Aktivnosti:

- stvoriti testne slučajeve, skripte za automatizaciju (ako je primjenjivo)
- pregledati i osnovne testne slučajeve i skripte
- stvoriti testne podatke (ako je testno okruženje dostupno).

Rezultati:

- testni slučajevi/skripte
- testni podaci.

4. POSTAVLJANJE TESTNOG OKRUŽENJA

Testno okruženje je skup softverske i hardverske konfiguracije. Postavljanje testnog okruženja jedan je od ključnih aspekata procesa testiranja i može se raditi paralelno s razvojnom fazom testnog slučaja. Testni tim možda neće biti uključen u ovu aktivnost ako korisnik ili razvojni tim osigurava testno okruženje, ali u tom slučaju testni tim mora izvršiti provjeru spremnosti okruženja na zahtjeve testiranja (engl. S*moke Test*).

Aktivnosti:

- razumijevanje potrebne arhitekture, postavke testnog okruženja i priprema popisa hardverskih i softverskih zahtjeva za testno okruženje.

- postavljanje testnog okruženja i testni podaci
- izvršavanje *Smoke testa* na verziji.

Rezultati:

- testno okruženje koje pokazuje testne podatke
- rezultati *Smoke testa.*

5. IZVRŠENJE TESTA

Tijekom ove faze testeri će provesti testiranje na temelju pripremljenih planova i testnih slučajeva. Pogreške će biti prijavljene razvojnom timu radi ispravljanja i ponovnog testiranja.

Aktivnosti:

- izvršiti testove prema planu
- dokumentirati rezultate testiranja i zabilježite nedostatke za neuspjele slučajeve
- mapirati nedostatke u testne slučajeve u RTM-u
- ponovno testirati popravke nedostataka
- pratiti nedostatke do zatvaranja

Rezultati:

- dovršena RTM sa statusom izvršenja
- testni slučajevi ažurirani rezultatima
- izvještaji o kvarovima

6. ZATVARANJE TESTNOG CIKLUSA

Tim za testiranje analizirat će proizvod testiranja (primjerice softver ili aplikaciju) kako bi identificirali strategije koje se moraju implementirati u sljedećem testnom ciklusu. Svaki ciklus daje povratnu informaciju o strategiji testiranja u sljedećem ciklusu.

Aktivnosti:

- ocijeniti kriterije završetka ciklusa na temelju vremena, pokrivenosti testa, cijene, softvera, kritičnih poslovnih ciljeva, kvalitete
- pripremiti metriku testa na temelju gore navedenih parametara
- pripremiti izvješće o zatvaranju testa
- kvalitativno i kvantitativno izvijestiti o kvaliteti proizvoda rada za krajnje korisnike
- analizirati rezultate testa kako bi se saznala raspodjela kvarova prema vrsti i težini.

Rezultati:

- izvješće o zatvaranju testa
- test metrike, odnosno pokazatelji koje smo dobili nakon testiranja, a utemeljeni su na ulaznim kriterijima.

Svaka od ovih faza ima određene **ulazne** i **izlazne kriterije**, **aktivnosti** i **rezultate** povezane s njom.

Ulazni kriteriji definiraju stavke koje se moraju ispuniti prije početka testiranja. Neki od ulaznih kriterija mogu biti:

1. uvjet za instalaciju i verifikaciju postojanja potrebnih zahtjeva za instalaciju.
2. testiranje zahtjeva za instalaciju.
3. testiranje softvera na kompatibilnosti s operacijskim sustavom.
4. testiranje sigurnosnih postavki i zaštite od virusa.

Izlazni kriteriji definiraju stavke koje moraju biti ispunjene prije nego što se testiranje može zaključiti. Pod izlazne kriterije spada:

1. potvrda da je softver instaliran i da je ispravno funkcioniranje.
2. potvrda da je softver kompatibilan s operacijskim sustavom.
3. potvrda da je sigurnosna zaštita ispravno postavljena.
4. potvrda da je softver u potpunosti funkcionalan.

SEKVENCIJALNI MODEL RAZVOJA

Razlikujemo dvije kategorije modela životnog ciklusa razvoja softvera - sekvencijalni i iterativni model. **Sekvencijalni razvojni model** opisuje proces razvoja softvera kao linearni tijek uzastopnih aktivnosti. To znači da svaka faza u procesu razvoja treba započeti tek kada je prethodna faza završena. U teoriji nema preklapanja faza, ali u većini projekata postoji njihovo blago preklapanje. To omogućuje povratne informacije za sljedeće faze. Najpoznatiji primjeri modela sekvencijalnog razvoja su model vodopada (engl. *Waterfall)* i V- model.

MODEL VODOPADA

Ovaj model izvorno je osmislio američki računalni znanstvenik Winston W. Royce 1970. godine. Pri opisivanju modela nije upotrijebio riječ vodopad, ali je zbog načina na koji je predstavio bit razvoja softvera model nazvan upravo tako. Ključni detalj Royceovog modela bilo je pozicioniranje razvojnih faza prema slijedu njihova izvođenja. Ako se pretpostavi da je vrijeme u modelu vodopada raspoređeno po vodoravnoj osi, tada se može zaključiti da sljedeća faza razvoja slijedi tek nakon što je završena prethodna. Model vodopada glavni je predstavnik tradicionalnih modela vođenja projekata iz kojeg su prilagođavanjem nastale i ostale metodologije u tradicionalnom kontekstu. On predstavlja planski vođen proces jer se cijeli proces razvoja mora planirati i vremenski definirati. Proces razvoja softvera podijeljen je u nekoliko zasebnih faza, a ishod jedne faze omogućuje ulaz u sljedeću fazu u nizu.

Faze modela vodopada:

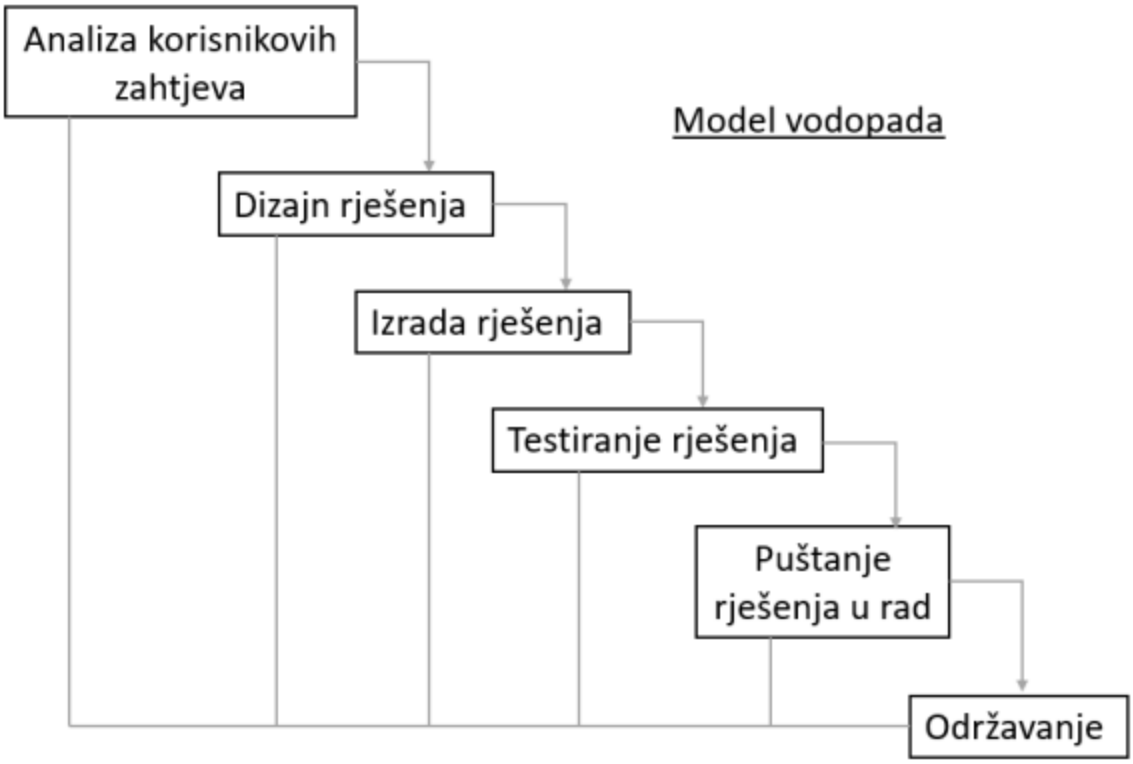
1. analiza i definiranje zahtjeva (specifikacija)
2. oblikovanje (engl. *Design*)
3. implementacija - testiranje
4. postavljanje aplikacije na poslužitelje (engl. *Deployment*)
5. održavanje

U fazi **analize i definiranja zahtjeva** uz konzultaciju s naručiteljima softverskog proizvoda definiraju se ciljevi, ograničenja, zahtjevi s poslovne i tehničke strane. Svi zahtjevi softverskog proizvoda koji se trebaju razviti u ovoj fazi obuhvaćeni su i dokumentirani u dokumentu sa specifikacijama zahtjeva.

U drugoj fazi, **oblikovanje ili dizajniranje**, proučavaju se zahtjevi iz prve faze te se priprema dizajn softverskog proizvoda. Dizajn pomaže u specificiranju hardverskih i softverskih zahtjeva, ali i u utvrđivanju cjelokupne arhitekture softverskog sustava čime se definira njihova međuovisnost. Rezultati iz faze dizajniranja softverskog proizvoda dijele se na skupine manjih programskih jedinica koje se integriraju u sljedećoj fazi. Svaka programska jedinica razvijena u ovoj fazi testira se jediničnim testovima (engl. *Unit test*). Nakon što se provedu sva funkcionalna i nefunkcionalna testiranja, softver se isporučuje naručitelju.

Faza održavanja sustava obično je najduža faza. U ovoj fazi softverski proizvod već je u upotrebi. Održavanje obuhvaća ispravljanje grešaka koje nisu otkrivene u ranijim fazama testiranja, poboljšanje softvera u smislu izdavanja novih verzija softvera koje će također biti skladu sa svim postavljenim specifikacijama.

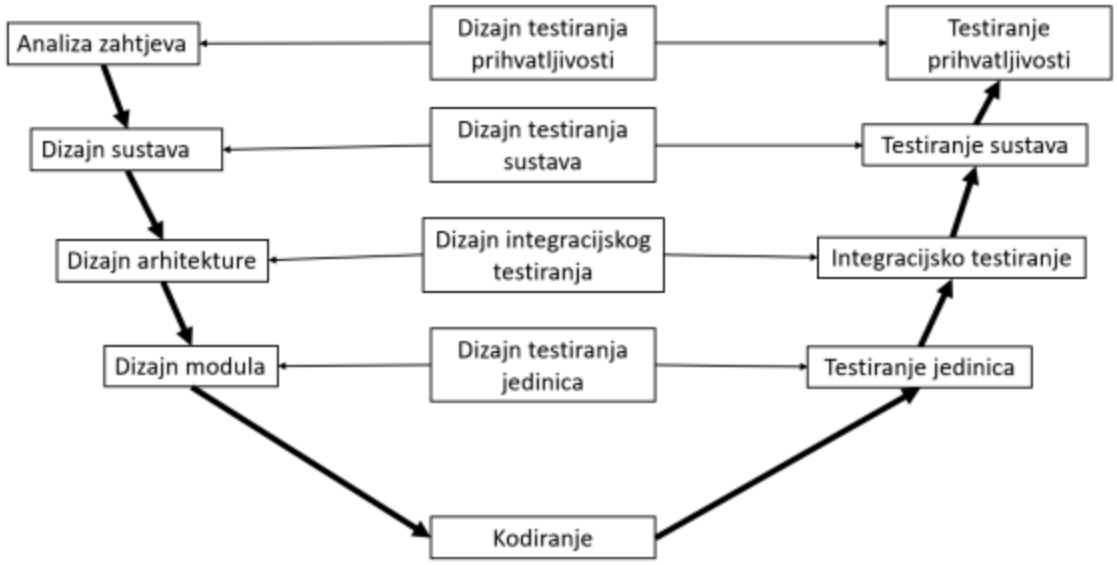
Zahvaljujući načinu na koji je model vodopada definiran, omogućeno je detaljno planiranje cijelog procesa i može se jednostavno ustanoviti trenutna faza projekta. Osim toga, ovaj model omogućuje čvrstu kontrolu nad projektom uz pomoć detaljne dokumentacije. Nedostatak ovog modela je da ne dozvoljava puno odmaka od planiranog. Jednom kada je aplikacija u fazi testiranja, vrlo je teško vratiti se i promijeniti nešto što u prethodnim fazama nije bilo dobro dokumentirano.



Slika 1.2. Primjer modela vodopada kroz njegove faze

V-MODEL

V-model je proširenje modela vodopada. On integrira proces testiranja kroz razvojni proces kako bi se implementirao princip ranog testiranja, odnosno kako bi se na vrijeme primijetile anomalije. V-model uključuje razine testiranja povezane sa svakom odgovarajućom razvojnom fazom.



Slika 1.3 Primjer V-modela kroz njegove faze

Aktivnosti na lijevoj strani V-modela usmjerene su na kreiranje proizvoda, za razradu početnih zahtjeva i zatim pružanje više tehničkih detalja timu za razvoj samih proizvoda:

- **specifikacija zahtjeva**: objašnjenje korisničkih potreba
- **funkcionalna specifikacija**: definicija funkcija potrebnih za zadovoljenje potreba korisnika
- **tehnička specifikacija:** tehnički dizajn ili arhitektura funkcija identificiranih u funkcionalnoj specifikaciji
- **specifikacija programa**: detaljan dizajn svakog modula ili jedinice koja će biti izgrađena kako bi zadovoljila traženu funkcionalnost.

Sredina V-modela pokazuje da planiranje testiranja može započeti čim određene funkcionalnosti softverskog proizvoda budu spremne za testiranje. Na primjer, kada su specifikacije zahtjeva spremne, može se započeti s planiranjem testiranja. Desna strana fokusirana je na aktivnosti testiranja (dinamičko testiranje). Dinamičko testiranje je metoda testiranja koja se odnosi na testiranje softvera ili hardvera čiji se ponašanje mijenja tijekom vremena. Ovo uključuje testiranje kako bi se osiguralo da se softver ili hardver nastavi ponašati u skladu s traženim specifikacijama i da se mogu prilagoditi

promjenama u okruženju. **Dinamičko testiranje softvera** obično uključuje testiranje funkcionalnosti, performansi i pouzdanosti softvera, dok **dinamičko testiranje hardvera** obično uključuje testiranje komponenti i sustava, kao i testiranje kompatibilnosti. Desna strana V-modela potvrđuje zahtjev korištenjem tehnika dinamičkog testiranja. **Faze dinamičkog testiranja** su:

- testiranje prema specifikaciji programa odvija se u fazi testiranja jedinice
- testiranje prema tehničkoj specifikaciji odvija se u fazi testiranja integracije
- testiranje prema funkcionalnoj specifikaciji odvija se u fazi testiranja sustava
- testiranje prema specifikaciji zahtjeva odvija se u fazi testiranja prihvatljivosti.

Nedostatci modela vodopada

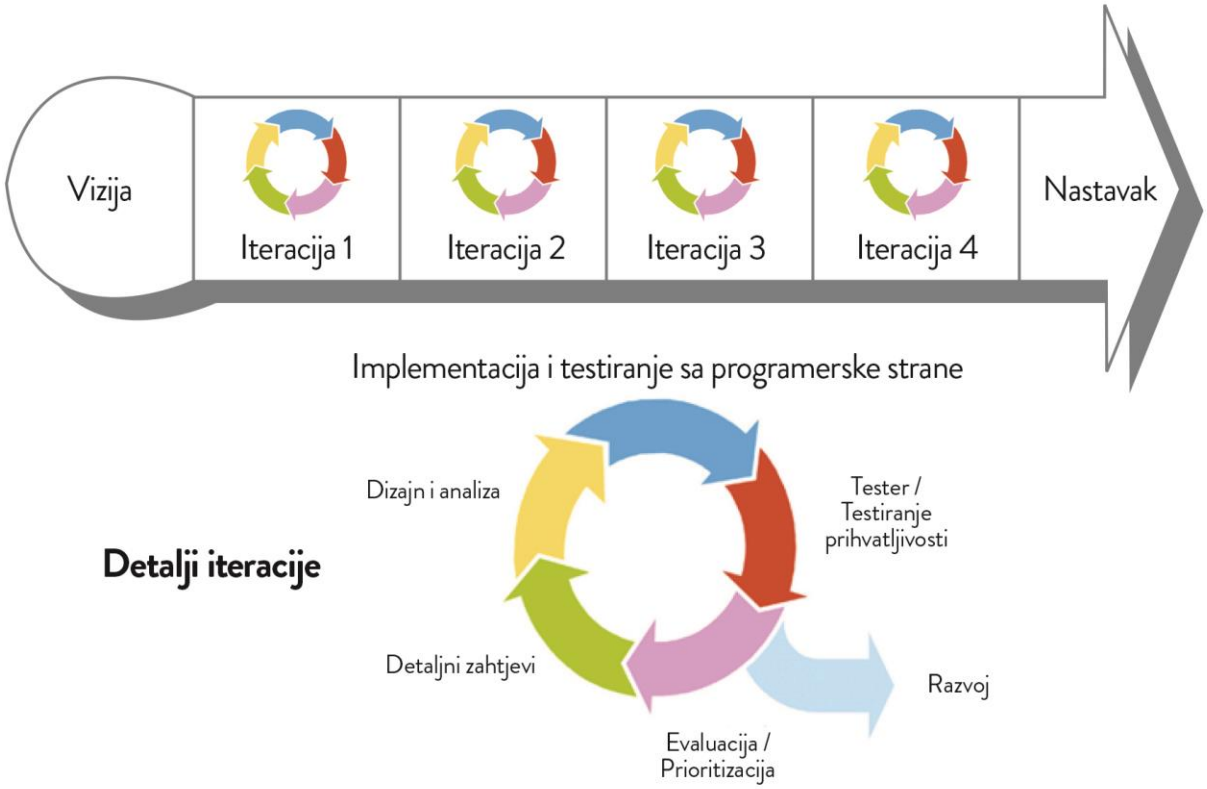
Testiranje u ovom modelu počinje tek nakon završetka implementacije neke značajke. Ako radite na velikom projektu, gdje su sustavi složeni, lako je propustiti ključne detalje u samoj fazi zahtjeva. U takvim će slučajevima klijentu biti isporučen potpuno pogrešan proizvod i možda ćete morati ispočetka započeti s projektom. Drugi slučaj je da uspijete zabilježiti zahtjeve ispravno, ali napravite ozbiljne pogreške u dizajnu i arhitekturi svog softvera. Tada je potrebno redizajnirati cijeli softver kako bi se pogreška ispravila. Kako bi se riješio ovaj problem, razvijen je V-model testiranja gdje za svaku fazu postoji odgovarajuća faza testiranja, kao što smo objasnili kod primjera V-modela.

ITERATIVNI MODEL RAZVOJA

Druga kategorija modela životnog ciklusa razvoja softvera je **iterativni model**. Iterativni odnosno **agilni model** razvoja je proces uspostavljanja zahtjeva, projektiranja, izgradnje i testiranja sustava, koji se izvodi kao niz manjih inkrementalnih razvoja. U iterativnom razvoju zahtjevi se mogu razjasniti ili otkriti kako se broj ponavljanja povećava. Pristup je “malo gradi, malo testiraj”. Svaka iteracija daje povratnu informaciju za sljedeću iteraciju. Osim toga, svaka iteracija završava s novom verzijom softverskog proizvoda, stoga je potrebno izvršiti **regresijsko testiranje** nakon svake pojedine iteracije. **Regresijsko testiranje** je vrsta testiranja softvera koja najčešće opisuje iterativni model razvoja, a

kojom se nastoji potvrditi da nedavna promjena funkcionalnosti ili koda nije nepovoljno utjecala na postojeće značajke same funkcionalnosti.

Inkrement proizveden iteracijom može se testirati na nekoliko razina kao dio njegovog razvoja. Regresijsko testiranje sve je važnije na svim iteracijama nakon prve. Inkrementalni razvoj uključuje utvrđivanje zahtjeva, projektiranje, izgradnju i testiranje sustava u dijelovima, što znači da značajke softvera rastu postupno. Veličina ovih povećanja značajki varira, pri čemu neke metode imaju veće, a neke manje dijelove. Povećanja značajki mogu biti samo jedna promjena na zaslonu korisničkog sučelja ili nova opcija upita.



Slika 1.4. Iterativni razvojni model (slika preuzeta s freepik.com)

Primjeri iterativnog modela razvoja:

Scrum (*Agile*) je metodologija upravljanja procesom izrade softvera koja se temelji na ideji iterativnog i inkrementalnog razvoja, gdje se proizvod razvija u malim koracima koji omogućuju fleksibilnost i brz odgovor na promjene. Naglašava suradnju, odgovornost i transparentnost. Svaka iteracija može biti relativno kratka (npr. dani ili tjedni), a prirast funkcionalnosti je shodno tome malen, kao što je nekoliko poboljšanja i/ili dvije ili tri nove funkcionalnosti. Često se koristi za razvoj softvera, ali se može primijeniti i na druge vrste projekata.

Glavni elementi Scruma su:

- *scrum tim*: samoorganizirajući i multifunkcionalni tim odgovoran za isporuku proizvoda
- *scrum Master*: voditelj koji pomaže timu da slijedi Scrum principe i prakse
- Vlasnik proizvoda (engl*. Product owner*): odgovoran za definiranje značajki i prioriteta proizvoda
- *sprint*: kratka, vremenski ograničena razdoblja (obično 1-4 tjedna) u kojima se stvara softverski proizvod koji se može iskoristiti i koji se potencijalno može isporučiti
- zaostatak *Sprinta* (engl*. Backlog*): popis zadataka koje treba izvršiti tijekom sprinta
- dnevni *Scrum*: dnevni sastanak na kojem članovi tima dijele svoj napredak i sve prepreke s kojima se suočavaju
- pregled *Sprinta*: sastanak na kraju svakog sprinta na kojem tim demonstrira obavljeni posao i prima povratne informacije
- retrospektiva *Sprinta*: sastanak na kraju svakog sprinta na kojem tim promišlja o svom procesu i identificira područja za poboljšanje.

Kanban (*Agile*) je metodologija upravljanja procesom izrade softvera. Sastoji se od dvaju glavnih elemenata - kartica i stupaca. Kartice predstavljaju zadatke (*task*) unutar projekta, a stupci faze projekta koje su uglavnom dijele na 3 dijela, primjerice “dizajn”, “razvoj” i “testiranje”. Kartice se mogu pomicati po fazama projekta. Primjerice, ako testiramo

značajku nekog softvera i ako ona nije pokazala zadovoljavajuć rezultat, može biti vraćena u fazu “razvoj” gdje će biti poboljšana i ponovno poslana na testiranje.

1.2.2. Generalni principi u testiranju

Kako bi se bolje kvalitetno pristupilo testiranju, testeri se mogu voditi generalnim principima u testiranju kojima gledaju proces testiranja iz različitih strana (programerskih ili korisničkih) i tako najlakše mogu ustanoviti zadovoljava li proizvod zahtjevima ili ne. Sedam je generalnih principa testiranja.

1. **Iscrpno testiranje nije moguće**

Iscrpno testiranje je teorijska metoda koja se koristi za potpuno testiranje svih mogućih izvedbenih scenarija određenog softvera. Ono se provodi na svim mogućim ulaznim podacima i kombinacijama ulaznih podataka, uključujući sve moguće granice i uvjete. Cilj iscrpnog testiranja je osigurati da je sve što je na neki način povezano sa softverom testirano i da se svi eventualni problemi otkriju.

2. **Grupiranje nedostataka** (engl. *Defect Clustering*)

Grupiranje nedostataka je proces koji omogućuje podjelu nedostataka na manje, lakše razumljive skupine. To daje jasniju sliku o tome gdje treba usmjeriti napore kako bi se poboljšali načini testiranja. Postoje različiti načini za grupiranje nedostataka, ali najčešći su prema vrsti, uzrocima, razinama i težini validacije nedostataka.

3. **Paradoks pesticida**

Ponavljana uporaba iste mješavine pesticida za iskorjenjivanje insekata tijekom uzgoja s vremenom će dovesti do toga da kukci razviju otpornost na pesticide, što znači da su pesticidi postali neučinkoviti na insekte. Isto vrijedi i za testiranje softvera. Ako se provede

isti niz ponovljenih testova, metoda će biti beskorisna za otkrivanje novih nedostataka. Kako bi se to prevladalo, testne slučajeve treba redovito pregledavati i revidirati, dodajući nove i različite testne slučajeve kako bi se pronašlo više nedostataka.

4. **Testiranje pokazuje prisutnost nedostataka**

Testiranje govori o prisutnosti nedostataka, a ne o odsutnosti nedostataka. Drugim riječima, testiranje softvera smanjuje vjerojatnost da u softveru ostanu neotkriveni nedostaci, ali čak i ako se nedostaci ne pronađu, to nije dokaz ispravnosti.

5. **Odsutnost pogreške - zabluda** (engl*. Absence of Error*)

Zabluda, odnosno pronalaženje i popravljanje nedostataka ne pomaže ako je konstrukcija sustava neupotrebljiva i ne ispunjava potrebe i zahtjeve korisnika.

6. **Rano testiranje**

Testiranje treba započeti što je prije moguće u životnom ciklusu razvoja softvera kako bi se svi nedostaci u zahtjevima ili fazi projektiranja detektirali što prije. Jeftinije je popraviti kvar u ranim fazama testiranja. Testiranje se preporučuje odmah nakon što se definiraju zahtjevi.

7. **Testiranje ovisi o kontekstu**

Način na koji testirate *web*-stranicu e-trgovine razlikovat će se od načina na koji testirate gotovu komercijalnu aplikaciju. Nisu svi razvijeni softveri identični. Možete koristiti različite pristupe, metodologije, tehnike i vrste testiranja, ovisno o vrsti aplikacije. Na primjer, testiranje bilo kojeg POS sustava u maloprodajnoj trgovini razlikovat će se od testiranja bankomata.

Načela testiranja pomoći će vam da stvorite učinkovitu strategiju testiranja i testne slučajeve za otkrivanje pogrešaka. Međutim, učenje principa je kao da učite voziti automobil. U početku, dok te učite voziti, obraćate pozornost na sve, kao što je mijenjanje brzina, rukovanje mjenjačem itd. Kada steknete iskustvo, usredotočite se samo na vožnju, a ostalo dolazi samo po sebi. Isto vrijedi i za principe testiranja. Iskusni testeri su usvojili ove principe do te razine da ih primjenjuju bez razmišljanja.

1.2.3. Svrha testiranja

Svrha testiranja softvera je **pronaći nedostatke prije nego li postanu kvarovi u stvarnom okruženju** (proizvodno okruženje), odnosno osigurati da su korisnički zahtjevi ispunjeni, inače softver neće raditi kako je predviđeno. Kada se projekt započne, bilježe se korisnički zahtjevi koji postaju osnova za tehničku specifikaciju proizvoda i tako nastaje **Dokument poslovnih zahtjeva** (engl*. Business requirements document,* BRD). On se koristi kao osnova za izradu testnih slučajeva za testiranje. Ti se testni slučajevi pokreću kako bi se osiguralo da su testiranjem pokriveni svi poslovni zahtjevi.

Testiranjem se osigurava da sustav ispunjava zahtjeve korisnika. Kako bi se to točno utvrdilo, potrebno je upotrijebiti procese verifikacije i validacije. **Verifikacija i validacija** su procesi koji se provode kroz sve faze razvoja programskog proizvoda, a odnose se na provjere i analize samog proizvoda (aplikacije).

Verifikacijom se provjerava je li programski proizvod usklađen sa specifikacijom. Njome ispitujemo funkcionalne i nefunkcionalne zahtjeve proizvoda koju izvodimo nakon svakog iteracijskog ciklusa, a izvršava je programer. S druge strane, **validacija** je općenita provjera usklađenosti programskog proizvoda s očekivanjima korisnika. Slijedi nakon verifikacije i označava provjeru cjelokupnog proizvoda, a provodi je tester.

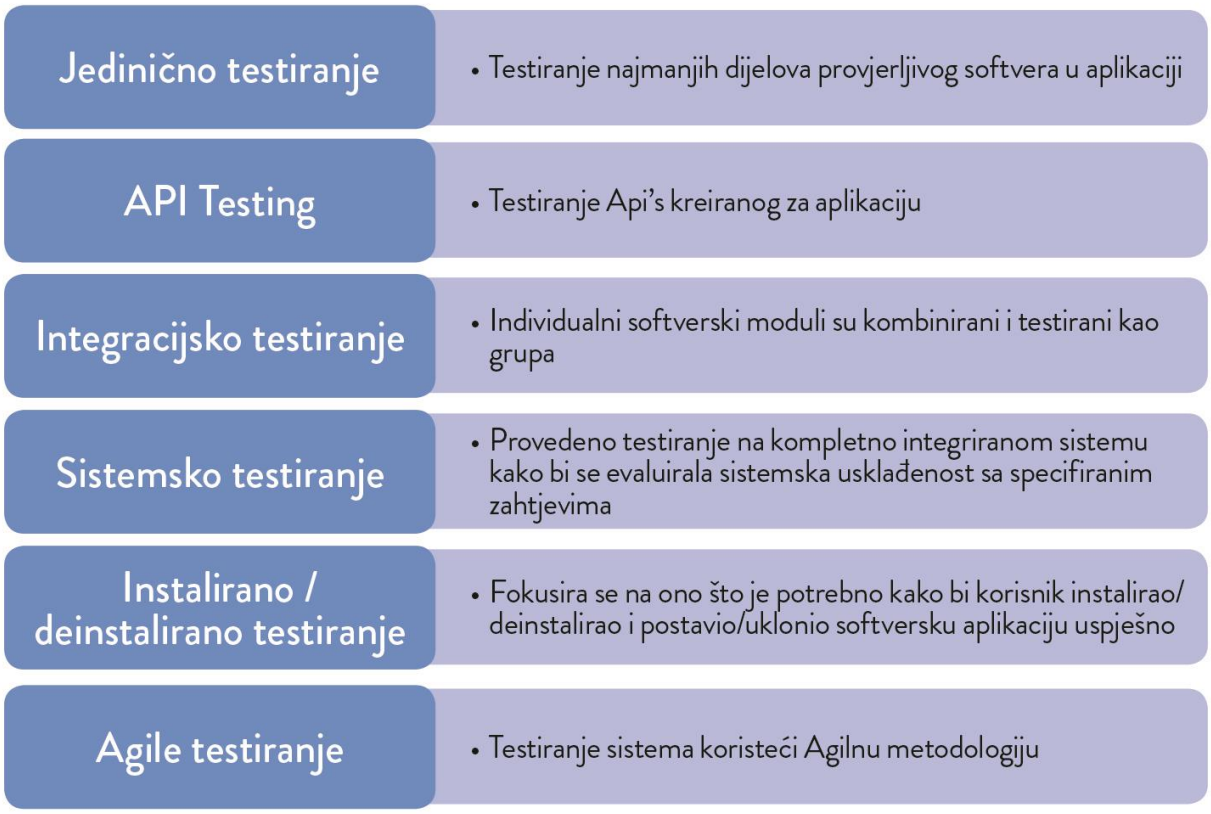
Dokumentiranje rizika i problema

Rizik je neizvjestan budući događaj koji ima određenu vjerojatnost da će se dogoditi i nosi određeni potencijal gubitka. Kada se rizik stvarno dogodi, on postaje “problem”. U planu testiranja kvalitete potrebno je dokumentirati rizike. Primjeri rizika koji se mogu dogoditi u testiranju softvera:

Rizik	**Smanjenje rizika**
Član tima nema potrebne vještine za testiranje *web*-stranice.	Planirajte tečaj za usavršavanje svojih članova.
Raspored projekta je pretijesan; teško je dovršiti projekt na vrijeme.	Postavite prioritet testiranja za svaku aktivnost testiranja.
Voditelj testiranja ima slabu vještinu upravljanja.	Planirajte obuku vođenja za menadžera.
Nedostatak suradnje negativno utječe na produktivnost zaposlenika.	Ohrabrite svakog člana tima u njegovom zadatku i potaknite ih na veće napore.
Pogrešna procjena proračuna i prekoračenje troškova.	Odredite opseg prije početka rada, posvetite puno pažnje projektu.

1.2.4. Faze/Nivoi testiranja u softverskom projektu s obzirom na model razvoja i životni ciklus softvera

Faze testiranja odnose se na postupak pronalaženja nedostataka u samom procesu razvoja softverskog proizvoda, kao i na izbjegavanje preklapanja i ponavljanja između faza životnog ciklusa razvoja. Kako bismo testirali bilo koju aplikaciju, moramo proći kroz sve faze SDLC-a. Postoji više razina testiranja, što nam pomaže u održavanju kvalitete softvera. Dolje prikazana slika prikazuje faze testiranja u softverskom projektu.



Slika 1.5. Grafički prikaz faza testiranja u životnom ciklusu softvera, preuzeto s freepik.com

1.2.4.1 Jedinično testiranje

Jedinično testiranje vrsta je funkcionalnog testiranja koja se koristi za testiranje ispravnosti rada jedne funkcije softvera. U modelima SDLC, STLC i V, jedinično testiranje

je prva razina testiranja koja se provodi prije integracijskog testiranja. Tijek rada jediničnog testiranja je:

1. stvaranje testnih slučajeva
2. implementacija prema napisanim testnim slučajevima
3. pregled/prerada napisanog koda
4. izvršavanje testnih slučajeva.

1.2.4.2. Integracijsko testiranje

Integracijsko testiranje je vrsta testiranja u kojem su komponente softvera logički integrirane i testirane kao grupa. Tipičan softverski projekt sastoji se od više komponenti softvera koje su kodirali različiti programeri. Integracijsko testiranje usredotočeno je na provjeru podatkovne komunikacije između ovih modula. Stoga se također naziva I&T (engl. *Integration and Testing*). Cilj integracijskog testiranja jest provjeriti radi li sustav kao cjelina sastavljena od pojedinih modula.

1.2.4.3. Sistemsko testiranje

Sistemsko testiranje je testiranje cjelovitog i potpuno integriranog softverskog proizvoda. Obično je softver samo jedan element većeg računalnog sustava i povezan je s drugim softverskim/hardverskim sustavima. **Sistemsko testiranje odvija se nakon integracijskog testiranja i prije testiranja prihvaćanja.**

Kada koristiti sistemsko testiranje:

- kod potpuno integriranih aplikacije kako bi se provjerilo kako komponente međusobno djeluju i sa sustavom u cjelini. Ovo se također naziva scenarijem testiranja od kraja do kraja
- za temeljito testiranje svakog ulaza u aplikaciji kako biste provjerili željene rezultate
- za testiranje korisničkog iskustva tijekom korištenja aplikacije.

Postoji više od 50 vrsta testiranja sustava, a ovdje navodimo one najčešće korištene:

- testiranje upotrebljivosti - uglavnom se fokusira na postizanje jednostavnosti korištenja aplikacije
- testiranje opterećenja - provjerava se hoće li softverski proizvod raditi pod stvarnim opterećenjima
- regresijsko testiranje - provjerava da nijedna promjena napravljena tijekom razvojnog procesa nije uzrokovala nove greške. Također osigurava da se stare greške ne pojave zbog dodavanja novih softverskih modula tijekom vremena
- testiranje migracije - provjerava može li se softver premjestiti sa starijih infrastruktura sustava na trenutne infrastrukture sustava bez ikakvih problema
- funkcionalno testiranje - provjerava funkcionalnosti pojedinih značajki. Testeri mogu napraviti popis dodatnih funkcija koje bi proizvod mogao imati kako bi ga poboljšali tijekom funkcionalnog testiranja.

1.2.4.5. Korisničko prihvaćanje

Testiranje korisničkog prihvaćanja je vrsta testiranja koje provodi Klijent kako bi potvrdio da je sustav u skladu sa zahtjevima koji su dogovoreni. Ovo se testiranje događa u završnoj fazi testiranja prije lansiranja softverske aplikacije na tržište ili u produkcijsko okruženje Ovo testiranje provodi se u zasebnom okruženju koje oponaša realno okruženje korisnika kada koristi aplikaciju.

Nakon što je softver prošao jedinično testiranje, testiranje integracije i sustava, test korisničkog prihvaćanja može se činiti suvišnim, no takvo je testiranje potrebno jer programeri kodiraju softver na temelju tehničke specifikacije koji je njihovo "vlastito" razumijevanje zahtjeva i možda zapravo nije ono što klijent želi od softvera.

1.3. Kvaliteta softverskog proizvoda

Prije definiranja pojma osiguranje kvalitete softvera potrebno je definirati pojmove **kvalitete softvera** i **kontrole kvalitete**. **Kvalitetan softver** mora raditi ono što je predviđeno da radi prema određenim specifikacijama, mora izvoditi zadatke ispravno i

prema očekivanom. S druge strane, **kontrola kvalitete** predstavlja podskup aktivnosti u osiguranju kvalitete, a definira se kao dio upravljanja kvalitetom usmjeren na ispunjavanje zahtjeva kvalitete.

Osiguranje kvalitete odnosi se na to kako se izvodi postupak osiguranja kvalitete, dok je **kontrola kvalitete** usmjerena na to tko provodi i nadzire postupak upravljanja kvalitetom. To znači da se nad softverom provode određena mjerenja i testiranja radi ocjenjivanja jedne ili više karakteristika i jesu li te karakteristike usklađene s definiranim zahtjevima.

Osiguranje kvalitete softvera predstavlja skup planiranih i sistematiziranih aktivnosti čije je izvođenje neophodno kako bi se osiguralo da neki proizvod zadovoljava određene tehničke preduvjete. Osiguranje kvalitete softvera trebalo bi se provlačiti kroz sve faze procesa razvoja softvera jer je to jedini način da se osigura njegova kvaliteta. Ono je usredotočeno na poboljšanje procesa razvoja softvera, kao i njegovo stvaranje prema definiranim standardima koji imaju cilj osigurati kvalitetu. Osim osiguranja kvalitete kroz upravljanje kvalitetom često se provlači i pojam kontrola kvalitete. Iako se ova dva pojma često naizmjenično koriste, ipak postoji određena razlika između njih.

1.3.1. Elementi softverskog proizvoda

Svaki softverski proizvod sastoji se od elemenata koje je nužno testirati kako bismo potvrdili da softver odgovara zahtjevima naručitelja softvera i potvrdili njegovu kvalitetu. Svaki od ovih elemenata potrebno je testirati:

1. **struktura** - elementi koji sačinjavaju strukturu softverskog proizvoda, poput:
    - kod: strukture koda koje čine proizvod, od izvršnih datoteka do pojedinačnih rutina
    - usluga: bilo koji poslužitelj ili proces koji radi neovisno o drugima koji može sadržavati proizvod

- neizvršne datoteke: sve datoteke osim multimedijskih ili programa, poput tekstualnih datoteka, uzoraka podataka ili datoteka pomoći
- garancija: sve izvan toga također je dio proizvoda, poput papirnatih dokumenata, *web*-stranica, pakiranja, licencnih ugovora itd.

2. **funkcija** - mogućnosti unutar aplikacije:
    - interaktivnost korisnika: svaka je funkcija osmišljena da olakša interakciju među ljudima ili da omogući istodobni pristup drugim resursima
    - računalna: bilo koja aritmetička funkcija ili aritmetičke operacije ugrađene u druge funkcije
    - sigurnosna: zaštita podataka; enkripcija; zaštita prednjeg kraja naspram stražnjeg kraja; ranjivosti u podsustavima
    - pokretanje/isključivanje; svaka metoda i sučelje za pozivanje i inicijalizaciju kao i izlazak iz proizvoda
    - multimedijska: zvukovi, video zapisi ili bilo koji grafički prikaz ugrađeni u proizvod koji se mogu kao takvi koristiti
    - testabilnost: sve funkcije koje se nude za pomoć pri testiranju proizvoda, kao što su dijagnostika, datoteke dnevnika, tvrdnje, izbornici za testiranje, itd.

3. **podaci** - sve što proizvod obrađuje:
    - ulaz/izlaz: svi podaci koje obrađuje proizvod i svi podaci koji proizlaze iz te obrade
    - unaprijed postavljeni podaci: svi podaci koji se isporučuju kao dio proizvoda ili su na drugi način ugrađeni u njega, kao što su gotove baze podataka, zadane vrijednosti itd.
    - nizovi/kombinacije: bilo kakvo sređivanje ili permutacija podataka, npr. red riječi, sortirani naspram nesortiranih podataka, redoslijed testova
    - veličine podataka: varijacije u veličini i agregaciji podataka
    - životni ciklus: transformacija tijekom životnog vijeka podatkovnog entiteta dok se stvara, pristupa, mijenja i briše.

4. **sučelja.** Pod sučelja podrazumijevamo kanale kojima se pristupa proizvodu, a pod njih spadaju:
    - korisnička sučelja: svaki element koji posreduje u razmjeni podataka s korisnikom (npr. zasloni, gumbi, polja, bilo fizički ili virtualni)
    - sučelja sustava: bilo koje sučelje s nečim što nije korisnik, kao što su drugi programi, tvrdi disk, mreža itd.
    - API/SDK: sva programska sučelja ili alati koji omogućuju razvoj novih aplikacija pomoću ovog proizvoda
    - sučelja za uvoz/izvoz svih funkcija koje pripremaju podatke za korištenje drugim modulima.

5. **platforma** - softverske i hardverske komponente nužne za pokretanje softvera:
    - vanjski hardver: hardverske komponente i konfiguracije koje nisu dio proizvoda za otpremu, ali su potrebne (ili izborne) proizvodu za rad: sustavi, poslužitelji, memorija, tipkovnice
    - vanjski softver: softverske komponente i konfiguracije koje nisu dio proizvoda za otpremu, ali su potrebne (ili opcionalne) proizvodu za rad: operacijski sustavi, aplikacije koje se istovremeno izvršavaju, upravljački programi, fontovi itd
    - ugrađene komponente: biblioteke i druge komponente koje su ugrađene u proizvod, ali su nisu nastale unutar vašeg projekta.

6. **operacije** - korisnikove mogućnosti upotrebe same aplikacije:
    - ovlasti korisnika: razina dopuštenja koju korisnik ima u upravljanju aplikacijom
    - okruženje: fizičko okruženje u kojem proizvod radi, uključujući elemente kao što su buka, svjetlo i smetnje.

7. **vrijeme** - vremenska ograničenja ili vremenske funkcionalnosti unutar same aplikacije:

- brzo/sporo: testiranje s "brzim" ili "sporim" unosom; najbrži i najsporiji unos; kombinacije brzog i sporog unosa
- promjenjive stope mjerenja vremena: ubrzavanje i usporavanje podataka u prijenosu (šiljci, *burstovi*, zastoji, uska grla, prekidi)
- konkurentnost: mogućnost da se više od jedne stvari događaju simultano (interaktivnost, dijeljenje vremena, niti, dijeljeni podaci).

1.3.2. Aspekti kvalitete softverskog proizvoda

Nekoliko je ključnih aspekata koje trebamo uzeti u obzir kada razvijamo softverski proizvod koji možemo procijeniti kao kvalitetan. Ovisno o tome na koje aspekte kvalitete stavljamo naglasak u razvoju softvera, prema tome ćemo planirati i kako ćemo ga testirati. Kvaliteta softverskog proizvoda može biti analizirana uzimajući u obzir ove aspekte:

1. **sposobnost -** može li proizvod obavljati potrebne funkcije?
    - dostatnost: proizvod posjeduje sve sposobnosti potrebne da služi svojoj svrsi
    - ispravnost: moguće je da proizvod funkcionira kako je predviđeno i da daje prihvatljiv rezultat.

2. **pouzdanost** - hoće li proizvod dobro raditi i odupirati se kvaru u svim potrebnim situacijama? Odnosi se na:
    - robusnost: proizvod nastavlja funkcionirati tijekom vremena bez degradacije, pod razumnim uvjetima
    - integritet podataka: podaci u sustavu zaštićeni su od gubitka ili oštećenja
    - sigurnost: proizvod se neće pokvariti na takav način da ošteti život ili imovinu.

3. **upotrebljivost** - koliko je jednostavno stvarnom korisniku koristiti proizvod? Odnosi se na:

- mogućnost učenja: krajnji korisnik ima vještinu potrebnu za korištenje proizvoda
- operativnost: proizvodom se može upravljati uz minimalan napor
- pristupačnost: proizvod zadovoljava relevantne standarde pristupačnosti i radi sa značajkama pristupačnosti.

4. **privlačnost proizvoda** - koliko je proizvod privlačan za upotrebu krajnjim korisnicima? Odnosi se na:
    - estetiku: proizvod privlači korisnike svojim izgledom
    - jedinstvenost: proizvod je nov ili poseban na neki način
    - dojam: proizvod daje željeni dojam kvalitete.

5. **sigurnost** - odnosi se na razinu zaštite softverskog proizvoda od neovlaštene upotrebe ili neželjenih napada:
    - autentifikacija: načini na koje sustav provjerava da je korisnik ono za što se predstavlja
    - autorizacija: prava koja su dodijeljena autentificiranim korisnicima na različitim razinama povlastica
    - privatnost: načini na koje su podaci krajnjih korisnika zaštićeni od neovlaštenih osoba
    - sigurnosne rupe: načini na koje sustav ne može osigurati sigurnost.

6. **skalabilnost** - odnosi na proširenje kapaciteta za procesiranje, memorijskih kapaciteta i slično u slučajevima povećanja broja korisnika, podataka i slično.

7. **kompatibilnost** - koliko dobro proizvod funkcionira s vanjskim komponentama i konfiguracijama?
    - kompatibilnost aplikacija: proizvod radi u kombinaciji s drugim softverskim proizvodima
    - kompatibilnost operacijskog sustava: proizvod radi s određenim operacijskim sustavom

- kompatibilnost hardvera: proizvod radi s određenim hardverskim komponentama i konfiguracijama
- kompatibilnost s prethodnim verzijama: proizvodi rade sa svojim ranijim verzijama.

8. **performanse** - koliko je proizvod brz i responzivan u izvođenju.

9. **mogućnost ugradnje** - koliko se lako može instalirati na ciljanu platformu ili više njih.

10. **razvoj** - koliko dobro možemo proizvod izraditi, testirati i modificirati?
    - mogućnost podrške: koliko će biti ekonomično pružiti podršku korisnicima proizvoda
    - mogućnost testiranja: koliko se učinkovito proizvod može testirati
    - mogućnost održavanja: koliko je ekonomično izgraditi, popraviti ili poboljšati proizvod
    - prenosivost: koliko će biti ekonomično prenijeti ili ponovno upotrijebiti tehnologiju negdje drugdje
    - lokalizacija: koliko će biti ekonomično prilagoditi proizvod za druga mjesta.

Navedeni aspekti kvalitete softvera su manje ili više važni ovisno o tome za što će softver biti korišten. Primjerice, kod izrade softvera za bankovnu aplikaciju naglasak ćemo staviti na sigurnost, pouzdanost i upotrebljivost, dok će aplikacija za mobilne igre naglasak imati na performanse, privlačnost i skalabilnost. Idealno je zadovoljiti sve aspekte kvalitete softvera, što danas suvremene aplikacije uglavnom i postižu.

Pitanja za ponavljanje:

1. Što je softversko testiranje?
2. Što su to anomalije i kada se pojavljuju?
3. Nabrojite neke od anomalija,
4. Što je to testni slučaj? Navedi primjer testnog slučaja.
5. Nabrojite i objasnite tri kategorije kod vrste testiranja.
6. Nabrojite koje su faze prisutne kod razvoja softvera.
7. Objasnite fazu održavanja.
8. Nabrojite koje su faze prisutne kod testiranja softvera.
9. Objasnite model vodopada i nabrojati njegove faze.
10. Objasnite prednosti i mane iterativnog modela razvoja.

2. Metode testiranja softvera

U OVOM POGLAVLJU NAUČIT ĆETE:

- metode testiranja softvera
- što je to nefunkcionalno testiranje i što ono podrazumijeva
- različite vrste automatiziranog testiranja.

Metoda testiranja softvera ima jednaku važnost kao i sam softver, što nam govori koliko je ključan odabir metode testiranja. Postoji mnogo različitih metoda i pristupa koji se mogu koristiti i ne postoji jedna koja bi se mogla izdvojiti kao najbolja. U praksi je odabir metode najčešće kombinacija nekoliko različitih metoda.

Metode testiranja koje se najčešće koriste su:

- statičko testiranje
- dinamičko testiranje
- testiranje crne kutije (engl. *Black-box testing*)
- testiranje bijele kutije (engl. *White-box testing*)
- testiranje sive kutije (engl. *Gray-box testing*)

- agilno testiranje
- *ad hoc* testiranje
- ručno testiranje
- automatizirano testiranje.

2.1 Metoda crne kutije (engl. *Black-box testing*)

Testiranje crne kutije je tehnika testiranja u kojoj se testira funkcionalnost aplikacije pod testom (engl. *Application under test, AUT*), ne uzimajući u obzir internu strukturu koda, detalje implementacije i znanja o internim putovima softvera. Ova vrsta testiranja u potpunosti se temelji na softverskim zahtjevima i specifikacijama.

Slika 2.1. Prikaz komponenti kod testiranja crne kutije

Crna kutija u testiranju crne kutije simbolizira nemogućnost uvida u unutarnji rad softvera, tako da se može testirati samo iskustvo krajnjeg korisnika. Sustav se promatra kao neprozirna kutija, pa njezina unutarnja aktivnost stoga nije vidljiva testeru. Tester se koncentrira na ulaze i izlaze sustava, bez osvrta na njegovu unutarnju strukturu. U tehnikama testiranja crne kutije (koje se nazivaju i biheviоralne tehnike ili tehnike temeljene na ponašanju) testni slučajevi dizajnirani su na temelju analize dokumentacije baze testova i odražavaju specificirano ponašanje sustava koji se testira. Najčešće korištene tehnike za odabir odgovarajućeg broja testnih slučajeva kod testiranja crne kutije su:

- podjela ekvivalencije
- analiza granične vrijednosti.

Podjela ekvivalencije je tehnika projektiranja testa koja dijeli ulaze i izlaze sustava u različite klase ili particije. Odabirom jedne vrijednosti iz svake particije osigurat ćete pokrivenost za sve vrijednosti iz te particije. Većina sustava sadrži velik broj mogućih ulaza i izlaza. Testiranje sustava uzimanjem u obzir sve njih gotovo je nemoguće. U takvim slučajevima, **tehnika podjele ekvivalencije** može biti korisna jer pomaže testeru u definiranju smanjenog broja testnih slučajeva koji mogu učinkovito testirati sustav. Particioniranje se koristi za stvaranje **klasa ekvivalencije** (često se nazivaju particije ekvivalencije) koje su skupovi vrijednosti koje se obrađuju na isti način. Odabirom jedne reprezentativne vrijednosti iz particije, pretpostavlja se pokrivenost za sve stavke u istoj particiji.

Primjer: Banka dopušta klijentima da se prijave za *online* bankovni račun. Dob korisnika uzima se kao ulaz i ako je dob korisnika između 18 i 60 godina, može nastaviti s *online* prijavom. Polje dobi korisnika prihvaća samo cjelobrojne vrijednosti veće od nule. Ako koristimo tehniku podjele ekvivalencije za testiranje ovog sustava, možemo podijeliti ulaz (dob krajnjeg korisnika) u tri klase ili podjele ekvivalencije.

Možemo uzeti bilo koju od vrijednosti iz svake klase ekvivalencije. npr. 15, 20, 65. Dakle, tri testa će biti dovoljna. U ovom slučaju, vrijednosti iz prve i treće particije su nevažeće vrijednosti, a vrijednosti iz druge particije su važeće vrijednosti. Ako postoji mala varijacija u zahtjevu i sustav prihvaća bilo koje cijele brojeve, tada će se particije povećati. Postojat će jedna particija za testiranje negativnih vrijednosti (dob unesena sa znakom '-') i jedna particija za testiranje dobi unesene kao nula. Dakle, za provjeru svih klasa ekvivalencije trebat će nam pet testnih slučajeva. Oni će provjeriti sustav pomoću (-15), (0), (15), (20), (65) (jedna vrijednost iz svake particije).

Analiza graničnih vrijednosti (engl*. Boundary Value Analysis,* BVA*)* je proširenje particije ekvivalencije, ali se može koristiti samo kada je particija poredana, odnosno kada se sastoji od sekvencijalnih podataka. Prva i posljednja vrijednost particije njezine su granične vrijednosti. Vjerojatnije je da će ponašanje na granicama particija ekvivalencije

biti netočno nego ponašanje unutar particija. U većini slučajeva, i specificirane i implementirane granice mogu se pomaknuti na položaje iznad ili ispod predviđenih položaja, mogu se potpuno izostaviti ili se mogu nadopuniti neželjenim dodatnim granicama. Analiza i testiranje granične vrijednosti otkrit će gotovo sve takve nedostatke prisiljavanjem softvera da pokaže ponašanja iz particije koja nije ona kojoj bi granična vrijednost trebala pripadati. Postoje dva načina pristupa BVA-u:

1. **Testiranje dviju vrijednosti**. Kod testiranja s dvjema vrijednostima koristi se granična vrijednost (na granici) i vrijednost koja je neposredno iznad granice (s najmanjim mogućim povećanjem).
2. **Testiranje triju vrijednosti.** Za testiranje granica s trima vrijednostima koriste se vrijednosti prije, na i preko granice. Vrijednosti se temelje na riziku povezanom sa stavkom koja se testira, pri čemu se za stavke većeg rizika koristi pristup s tri granice.

I za particioniranje ekvivalencije i za analizu graničnih vrijednosti izlazne vrijednosti postavljenih klasa također trebaju ispitati kako bi se osiguralo da su valjane.

Sumirano, glavne karakteristike testiranja crne kutije su:

- često se naziva funkcionalnim testiranjem
- tehnika osmišljene bez znanja o unutarnjoj strukturi programa i dizajna
- zasnovana na temelju funkcionalnih zahtjeva.

2.2. Metoda bijele kutije (engl. *White-box testing*)

Testiranje bijele kutije je testiranje unutarnje strukture, dizajna i načina kodiranja softverskog rješenja. U ovoj vrsti testiranja kod je vidljiv testeru. Testiranje bijele kutije prvenstveno se fokusira na provjeru protoka ulaza i izlaza kroz aplikaciju, poboljšanje dizajna i upotrebljivosti te jačanje sigurnosti. Testiranje bijele kutije također je poznato kao strukturno testiranje, testiranje prozirne kutije, testiranje temeljeno na kodu ili testiranje staklene kutije. Obično ga izvode programeri. Ono se temelji na unutarnjem

radu aplikacije i fokusira na interno testiranje. Izraz bijela kutija korišten je jer simbolizira sposobnost da se kroz vanjsku ljusku (ili "kutiju") softvera vidi njegov unutarnji rad.

Testiranje bijele kutije uključuje testiranje softverskog koda za:

- unutarnje sigurnosne rupe
- neispravne ili loše strukturirane staze u procesima kodiranja
- protok specifičnih ulaza kroz kod
- očekivani rezultat
- funkcionalnost uvjetnih petlji
- testiranje svake izjave, objekta i funkcije na pojedinačnoj osnovi.

Testiranje se može provesti na razini sustava, integracije i jedinice razvoja softvera. Jedan od osnovnih ciljeva testiranja bijele kutije je provjera tijeka rada aplikacije. Uključuje testiranje niza unaprijed definiranih ulaza u odnosu na očekivane ili željene izlaze, tako da kada određeni unos ne rezultira očekivanim izlazom, radi se o pogrešci.

Dva su osnovna koraka u testiranju bijele kutije.

1. **Razumijevanje osnovnog koda**

Prvo je potrebno naučiti i razumjeti izvorni kod aplikacije. Budući da testiranje bijele kutije uključuje testiranje unutarnjeg rada aplikacije, tester mora dobro poznavati programske jezike koji se koriste u aplikacijama koje testira. Također, osoba koja testira mora poznavati praksu sigurnog kodiranja. Sigurnost je često jedan od primarnih ciljeva testiranja softvera. Tester bi trebao moći pronaći sigurnosne probleme i spriječiti napade hakera ili postupke naivnih korisnika koji bi mogli ubaciti zlonamjerni kod u aplikaciju, svjesno ili nesvjesno.

2. **Stvaranje testnih slučajeva i njihovo izvršavanje**

Drugi osnovni korak testiranja bijele kutije uključuje testiranje izvornog koda aplikacije, pazeći na pravilan tijek i strukturu testnih slučajeva. Jedan od načina je pisanje više koda za testiranje izvornog koda aplikacije. Tester će razviti male testove za svaki proces ili niz

procesa u aplikaciji. Ova metoda zahtijeva da tester dobro poznaje kod i često je radi programer. Ostale metode uključuju ručno testiranje, testiranje pokušaja i pogrešaka te korištenje alata za testiranje.

Glavna tehnika testiranja bijele kutije je **analiza pokrivenosti koda**. Analiza pokrivenosti koda uklanja nedostatke u paketu testnih slučajeva i identificira područja programa koja nisu obuhvaćena skupom testnih slučajeva. Nakon što se identificiraju praznine, stvaraju se testni slučajevi kako bi se testirali neprovjereni dijelovi koda, čime se povećava kvaliteta softverskog proizvoda. Dostupni su automatizirani alati za izvođenje analize pokrivenosti koda.

Najčešće korištene **tehnike analize pokrivenosti koda**:

- **pokrivenost izjave** - tehnika zahtijeva da se svaka moguća izjava u kodu testira barem jednom tijekom procesa testiranja softverskog inženjerstva
- **pokrivenost grana** - tehnika provjerava svaki mogući put softverske aplikacije.

Korištenjem pokrivenosti izjave i pokrivenosti grane moguće je postići 80-90% pokrivenosti koda. Osim navedenih, postoje brojne vrste pokrivenosti kao što su pokrivenost uvjetom, višestruka pokrivenost uvjeta, pokrivenost putanje, pokrivenost funkcija itd. Svaka tehnika ima svoje prednosti i pokušava testirati sve dijelove softverskog koda.

Testiranje bijele kutije obuhvaća nekoliko **tipova testiranja** koji se koriste za procjenu upotrebljivosti aplikacije, dijela koda ili određenog softverskog paketa, a to su:

- **jedinično testiranje**
- **testiranje curenja memorije**: curenje memorije učestali je uzrok sporijeg rada aplikacija. Tester koji ima iskustva u otkrivanju curenja memorije neophodan za detektiranje, a onda i za unapređenje spore softverske aplikacije.

Testiranje bijele kutije može biti prilično složeno. Složenost testiranja uvelike ovisi o aplikaciji koja se testira - jednostavna aplikacija koja izvodi jednu operaciju mogla bi se

testirati u nekoliko minuta, dok su većim aplikacijama za programiranje potrebni dani, tjedni, pa čak i više vremena za potpuno testiranje.

2.3. Alati za praćenje pokrivenosti izvršenog koda - Metoda sive kutije

Metoda sive kutije (engl. *Gray Box Testing*) je tehnika testiranja softverskog proizvoda ili aplikacije s djelomičnim poznavanjem unutarnje strukture aplikacije. Testiranje sive kutije je kombinacija testiranja bijele kutije i metode testiranja crne kutije, s obzirom na to da je unutarnja struktura (kod) djelomično poznata. Svrha testiranja sive kutije je pretraživanje i prepoznavanje nedostataka nastalih zbog nepravilne strukture koda ili nepravilnog korištenja aplikacija. U softverskom inženjerstvu, metoda sive kutije daje mogućnost testiranja obje strane aplikacije - i prezentacijskog sloja i dijela koda. Prvenstveno je korisna u integracijskom testiranju i testiranju prodora.

Testiranje sive kutije pruža kombinirane prednosti testiranja crne kutije i testiranja bijele kutije i provodi se iz sljedećih razloga:

- kombinira doprinos programera kao i testera i poboljšava ukupnu kvalitetu proizvoda
- smanjuje troškove dugog procesa testiranja funkcionalnih i nefunkcionalnih tipova
- programeru daje dovoljno slobodnog vremena da popravi nedostatke
- testiranje se provodi sa stajališta korisnika, a ne sa stajališta dizajnera.

Za izvođenje testiranja sive kutije nije nužno da tester ima pristup izvornom kodu. Test je dizajniran na temelju poznavanja algoritma, arhitekture, internih stanja ili drugih opisa ponašanja programa na visokoj razini. Za izvođenje testiranja sive kutije potrebno je primijeniti jednostavnu tehniku testiranja crne kutije. Testiranje bijele kutije temelji se na generiranju testnog slučaja, koji se kao takav unaprijed postavlja sa svim uvjetima.

Najčešće korištene tehnike za testiranje sive kutije su:

- **matrično testiranje**: ova tehnika testiranja uključuje definiranje svih varijabli koje postoje u kodu softverskog proizvoda
- **regresijsko testiranje**: ovo se testiranje koristi kako bi se provjerilo je li promjena u prethodnoj verziji umanjila druge aspekte programa u novoj verziji
- **testiranje uzorka**: ovo se testiranje izvodi na povijesnim podacima o prethodnim greškama sustava. Za razliku od testiranja crne kutije, testiranje sive kutije kopa po kodu i utvrđuje zašto je došlo do kvara.

2.4. Nefunkcionalno testiranje

Nefunkcionalno testiranje je vrsta testiranja softvera za provjeru nefunkcionalnih aspekata (izvedba, upotrebljivost, pouzdanost itd.) softverske aplikacije.

Ciljevi nefunkcionalnog testiranja:

- trebalo bi povećati upotrebljivost, učinkovitost, mogućnost održavanja i prenosivost softverskog proizvoda
- pomaže smanjiti proizvodni rizik i troškove povezane s nefunkcionalnim aspektima softverskog proizvoda
- optimizira način na koji se softverski proizvod instalira, postavlja, izvršava, upravlja i nadzire
- prikuplja i proizvodi mjerenja i metriku za interno istraživanje i razvoj
- unaprjeđuje i poboljšava znanje o ponašanju proizvoda i tehnologija kojima se koristi.

Parametri nefunkcionalnog testiranja su:

- sigurnost - definira kako je sustav zaštićen od namjernih napada iz unutarnjih i vanjskih izvora
- pouzdanost - mjera u kojoj bilo koji softverski sustav kontinuirano obavlja navedene funkcije (funkcije navedene u zahtjevu) bez greške

- mogućnost preživljavanja - provjerava nastavlja li softverski sustav funkcionirati i oporavlja li se u slučaju kvara sustava
- upotrebljivost - lakoća kojom korisnik može učiti, raditi, pripremati ulaze i izlaze kroz interakciju sa sustavom
- skalabilnost - stupanj u kojem bilo koja softverska aplikacija može proširiti svoj kapacitet obrade kako bi mogla određivati veću količinu posla u istom vremenu
- interoperabilnost - nefunkcionalni parametar koji provjerava sučelja softverskog sustava s drugim softverskim sustavima
- učinkovitost - razmjer u kojem bilo koji softverski sustav može rukovati kapacitetom, količinom i vremenom odziva
- fleksibilnost - lakoća kojom aplikacija može raditi u različitim hardverskim i softverskim konfiguracijama. Odnosi se na definiranje minimalne količine RAM-a, broja jezgara procesora i slično.
- prenosivost - fleksibilnost softvera za prijenos iz trenutnog hardverskog ili softverskog okruženja.

Nefunkcionalna testiranja koje ćemo navesti u nastavku priručnika neka su od najvažnijih metoda takvih testiranja i onih s kojima ćete se najviše susretati u svom radu.

2.4.1. Testiranje performansi

Testiranje performansi je vrsta testiranja softvera koje osigurava da će softverske aplikacije dobro raditi pod očekivanim radnim opterećenjem. Izvedba softverske aplikacije poput vremena odziva, pouzdanosti, upotrebe resursa i skalabilnosti je bitna.

Fokus testiranja performansi je provjera sljedećih karakteristika softvera:

- **brzina** - određuje hoće li aplikacija brzo reagirati.
- **skalabilnost** - određuje maksimalno korisničko opterećenje koje softverska aplikacija može podnijeti.
- **stabilnost** - određuje je li aplikacija stabilna pod različitim opterećenjima.

Testiranje performansi provodi se kako bi se pružile informacije o brzini, stabilnosti i skalabilnosti softverskog proizvoda. Testiranje performansi također otkriva što treba poboljšati prije nego što proizvod izađe na tržište. Bez testiranja performansi softver će vjerojatno patiti od problema kao što su usporen rad dok ga nekoliko korisnika koristi istovremeno, nedosljednosti u različitim operacijskim sustavima i loša upotrebljivost.

Vrste testiranja performansi

- **testiranje opterećenja** - provjerava sposobnost aplikacije da radi pod očekivanim korisničkim opterećenjem. Cilj je identificirati uska grla u izvedbi prije nego što li korisnik pokrene softversku aplikaciju.
- **testiranje otpornosti na stres** - uključuje testiranje aplikacije pod ekstremnim radnim opterećenjem da se vidi kako se nosi s velikim prometom ili obradom podataka. Cilj je identificirati prijelomnu točku aplikacije.
- **testiranje izdržljivosti** - provodi se kako bi se osiguralo da softver može podnijeti očekivano opterećenje tijekom dugog razdoblja
- **testiranje volumena** - provodi se testiranje velikog broja podataka. Podaci se popunjavaju u bazi podataka i nadzire se cjelokupno ponašanje softverskog sustava. Cilj je provjeriti performanse softverske aplikacije pod različitim volumenima baze podataka.
- **testiranje skalabilnosti** - cilj je utvrditi učinkovitost softverske aplikacije kako bi se podržalo povećanje korisničkog opterećenja. Pomaže u planiranju dodavanja kapaciteta softverskom sustavu. Ovdje se radi o dodatnim poslužiteljima kod horizontalnog skaliranja i više procesora i jezgara u slučaju vertikalnog skaliranja.

Najčešći problemi kod izvedbe testiranja

Većina problema vezana je uz brzinu izvršavanja aplikacije (proces pokretanja aplikacije) i njezinog vremena odaziva (koliko je sekundi potrebno da bi se aplikacija pokrenula), vremena učitavanja i neadekvatne mogućnosti skaliranja. Brzina je često jedan od

najvažnijih atributa aplikacije. Aplikacija koja radi sporo izgubit će korisnike. U nastavku su opisani najčešći problemi s izvedbom testa performansi:

- **dugo vrijeme učitavanja** - idealno bi bilo da je vrijeme potrebno za pokretanje aplikacije svedeno na minimum. Iako je neke aplikacije nemoguće učitati za manje od jedne minute, vrijeme učitavanja trebalo bi biti manje od nekoliko sekundi ako je moguće.
- **loše vrijeme odgovora** - vrijeme odgovora odnosi se na vrijeme od trenutka kada korisnik unese podatke u aplikaciju do trenutka kada aplikacija pošalje odgovor na taj unos. To bi trebalo biti vrlo brzo jer ako korisnik mora predugo čekati, gubi interes.
- **loša skalabilnost -** softverski proizvod pati od slabe skalabilnosti kada ne može podnijeti očekivani broj korisnika ili kada se ne prilagođava dovoljno širokom rasponu korisnika. Zato treba provoditi testiranje opterećenja kako bi se provjerilo može li aplikacija podnijeti predviđeni broj korisnika.
- **uska grla** - prepreke u sustavu koje degradiraju ukupne performanse sustava. Riječ je o smanjenju propusnosti pod određenim opterećenjima koje mogu uzrokovati pogreške u kodiranju ili problemi s hardverom. Ključ za rješavanje problema s uskim grlom je pronaći dio koda koji uzrokuje usporavanje i pokušati ga popraviti. Uska grla se adresiraju popravljanjem loše pokrenutih procesa ili dodavanjem dodatnog hardvera. Neka uobičajena uska grla u radu su:
    - iskoristivost CPU-a
    - iskoristivost memorije
    - iskoristivost mreže
    - ograničenja operacijskog sustava
    - brzina medija za pohranu.

Metodologija za testiranje performansi može uvelike varirati, ali cilj ostaje isti - pokazati **zadovoljava li softverski sustav unaprijed definirane kriterije izvedbe**. Također može poslužiti za usporedbu performansi dvaju softverskih sustava, kao i u prepoznavanju dijelova softverskog sustava koji smanjuju njegovu izvedbu.

2.4.2. Testiranje pod naprezanjem (engl. *Load testing*)

Testiranje opterećenja je vrsta testiranja performansi koje određuje performanse sustava pod stvarnim uvjetima opterećenja. Ovo testiranje pomaže odrediti kako se aplikacija ponaša kada joj više korisnika istovremeno pristupa.

Ovo testiranje obično ukazuje na:

- maksimalni radni kapacitet aplikacije
- je li trenutna infrastruktura dovoljna za pokretanje aplikacije
- održivost aplikacije s obzirom na korisničko opterećenje
- broj istodobnih korisnika koje aplikacija može podržati i skalabilnost kako bi se omogućilo većem broju korisnika da joj pristupe.

Testiranje opterećenja provodi se kako bi se simuliralo veće opterećenje aplikacije te kako bi se adekvatno planirala skalabilnost sustava u budućnosti. Primjerice, neke popularne *web*-stranice pretrpjele su zastoje kada bi se povećao broj korisnika aplikacije. *Web*-stranice za e-trgovinu puno ulažu u reklamne kampanje, ali ne i u testiranje opterećenja kako bi se osigurala optimalna izvedba sustava, kada taj marketing donosi promet. Encyclopedia Britannica proglasila je besplatni pristup 2007. godine svojoj online bazi podataka promotivnom ponudom. Tjednima nisu mogli držati korak s navalom prometa. Mnoge *web*-stranice pate od odgođenog učitavanja kada naiđu na veliki promet, a većina korisnika klikne na izlaz nakon 8 sekundi kašnjenja u učitavanju stranice.

Ciljevi testiranja opterećenja

Testiranje opterećenja identificira sljedeće probleme prije plasiranja aplikacije na tržište ili proizvodnju:

- vrijeme odgovora za svaku transakciju
- performanse komponenti sustava pod različitim opterećenjima
- performanse komponenti baze podataka pod različitim opterećenjima
- kašnjenje mreže između klijenta i poslužitelja
- problemi s dizajnom softvera

- problemi s konfiguracijom poslužitelja poput *web*-poslužitelja, aplikacijskog poslužitelja, poslužitelja baze podataka itd.
- problemi s hardverskim ograničenjima poput maksimiziranja CPU-a, ograničenja memorije, mrežno usko grlo itd.

2.4.3. Testiranje sigurnosti

Testiranje sigurnosti vrsta je testiranja softvera koja otkriva ranjivosti, prijetnje, rizike u softverskoj aplikaciji i sprječava zlonamjerne napade uljeza. Svrha sigurnosnih testova je identificirati sve moguće slabosti softverskog sustava koje bi mogle rezultirati gubitkom informacija, prihoda, ugleda zaposlenika ili vanjskih suradnika povezanih s organizacijom.

Zašto je sigurnosno testiranje važno?

Glavni cilj sigurnosnog testiranja je identificirati prijetnje u sustavu i izmjeriti njegove potencijalne slabosti kako bi se prijetnje mogle susresti i sustav ne bi prestao funkcionirati ili se ne bi mogao iskoristiti. Takvo testiranje pomaže u otkrivanju mogućih sigurnosnih rizika u sustavu i pomaže programerima da riješe probleme kodiranjem.

Sedam glavnih vrsta testiranja sigurnosti:

- **skeniranje ranjivosti:** ovo se radi putem automatiziranog softvera za skeniranje sustava protiv poznatih potpisa ranjivosti sustava. Jedan od najpoznatijih alata za takvo testiranje je alat Wireshark.
- **sigurnosno skeniranje:** uključuje prepoznavanje slabosti mreže i sustava, a kasnije nudi rješenja za smanjenje tih rizika. Ovo skeniranje može se izvesti i za ručno i za automatsko skeniranje. Neki od alata kojima se može postići takvo testiranje su NetSparker,Google Nogotofail, Wapiti.

- **testiranje prodora** (engl*. Penetration testing*)**:** ova vrsta testiranja simulira hakerski napad. Ovo testiranje uključuje analizu određenog sustava kako bi se provjerile potencijalne ranjivosti na vanjski pokušaj hakiranja.
- **procjena rizika:** ovo testiranje uključuje analizu sigurnosnih rizika uočenih u organizaciji. Rizici su klasificirani kao niski (kozmetičke promjene na samoj aplikaciji), srednji (testiranje nove funkcionalnosti koje se obavlja u razvojnom okruženju) i visoki (rizik testiranja nove funkcionalnosti koja zahvaća veliki broj korisnika). Ovo testiranje preporučuje kontrole i mjere za smanjenje rizika.
- **sigurnosna revizija:** ovo je interna inspekcija aplikacija i operacijskih sustava radi sigurnosnih nedostataka. Revizija koda se također može provesti pregledom koda red po red.
- **etičko hakiranje:** hakiranje sustava organizacijskog softvera. Organizacijski softver je vrsta softvera koji je dizajniran za pomoć organizacijama u upravljanju različitim aspektima njihovog poslovanja, kao što su upravljanje projektima, upravljanje odnosima s klijentima, upravljanje ljudskim resursima i financijsko upravljanje. Za razliku od zlonamjernih hakera koji kradu podatke radi vlastite dobiti, namjera je razotkriti sigurnosne propuste u sustavu.
- **procjena stanja:** ovo kombinira sigurnosno skeniranje, etičko hakiranje i procjenu rizika kako bi se prikazao ukupni status sigurnosti organizacije.

2.4.4. Testiranje pouzdanosti

Testiranje pouzdanosti je proces testiranja softvera koji provjerava funkcionira li softver bez greške u određenom okruženju tijekom određenog razdoblja. Svrha testiranja pouzdanosti je osigurati da softverski proizvod nema grešaka i da je dovoljno pouzdan za svoju očekivanu svrhu.

Čimbenici koji utječu na pouzdanost softvera:

- broj grešaka prisutnih u softveru
- kako na koji korisnici upravljaju sustavom.

Testiranje pouzdanosti jedan je od preduvjeta za kvalitetan softver. Ono pomaže u otkrivanju mnogih problema u dizajnu i funkcionalnosti softvera i provodi se na nekoliko razina. Složeni sustavi će se testirati na **razini jedinice** (funkcija koja izvršava određenu logiku i vraća rezultat)**, sklopa** (npr. registracijski sklop koji obuhvaća sve jedinice potrebne za izvršavanje procesa registracije korisnika)**, podsustava i sustava** (npr. sustav za e-trgovinu koji obuhvaća sve sklopove i jedinice potrebne za funkcioniranje *online* trgovine).

Zašto testirati pouzdanost?

Testiranje pouzdanosti provodi se kako bi se testirala izvedba softvera u danim uvjetima. Ono se izvodi kako bi se utvrdilo koliko je neki proizvod ili sistem pouzdan, odnosno koliko će dugo raditi bez grešaka ili prekida u radu. Testiranje pouzdanosti omogućava identifikaciju problema i nedostataka u proizvodima, što pomaže u njihovom otklanjanju i poboljšanju performansi. Uključuje **testiranje značajki, testiranje opterećenja i regresijsko testiranje**.

2.4.5. Testiranje upotrebljivosti i pristupačnosti

Web-pristupačnost znači da osobe s invaliditetom mogu jednako percipirati, razumjeti, kretati se i komunicirati s *web*-stranicama i alatima. U **testiranju pristupačnosti** testiramo koliko se lako može pristupiti sadržaju *web*-stranice/aplikacije pomoću navigacije samo pomoću tipkovnice i pomoćnih uređaja kako bi osobe s privremenim ili trajnim invaliditetom mogle imati jednak pristup sadržaju *web*-stranice i uspješno izvršiti zadatke.

Svrha testiranja digitalne pristupačnosti je razumjeti kako korisnici s invaliditetom stupaju u interakciju s *web*-stranicom pomoću navigacije samo pomoću tipkovnice i pomoćnih uređaja. Ovo pruža uvide koji pomažu u tome da *web*-stranice budu pristupačnije svima.

Testiranje upotrebljivosti identificira korisničke probleme kod korištenja (engl*. Usability problems*), što pomaže u zadržavanju korisnika.

Testiranje upotrebljivosti, koje se naziva i testiranje korisničkog iskustva, mjeri sveukupno zadovoljstvo korisnika *web*-stranice/aplikacije, prvenstveno se fokusirajući na interakcije korisničkog sučelja i uspješno izvršavanje zadataka. Svrha testiranja upotrebljivosti je razumjeti kako stvarni korisnici stupaju u interakciju s *web*-stranicom i odrediti koje značajke dodati ili poboljšati.

Testiranje upotrebljivosti pomaže u otkrivanju problema i dobivanju povratnih informacija od publike koja koristi sustav tijekom testiranja. Smanjuje rizik od izgradnje nezgrapnih proizvoda teških za korištenje i štedi dragocjene resurse.

Uključivanje UI/UX[2] najboljih praksi i smjernica za digitalnu pristupačnost u ranoj fazi razvoja proizvoda štedi vrijeme i novac u izradi *web*-stranica, što smanjuje rizik od skupih popravaka u kasnijoj fazi.

Za testiranje upotrebljivosti *web*-stranice, tester će proći kroz scenarije iz stvarnog života i izvršiti različite zadatke kako bi identificirao nedostatke kroz značajku upotrebljivosti *web*-stranice. Primjerice, korisniku se mora omogućiti da je opcija pretraživanja dostupna i da daje točne rezultate kako bi korisnik mogao dobiti traženi rezultat samog pretraživanja.

Ključne značajke testiranja pristupačnosti

Za testiranje pristupačnosti *web*-stranice možemo uzeti primjer iz stvarnog života. Primjerice, korisniku su dostupne opcije podnatpisa i prijepisa za audio i video sadržaj.

[2] UI (User Interface, ili korisnički interfejs) je ono što korisnik vidi i s čime može da interaktuje kada koristi neki proizvod ili sistem. To uključuje elemente kao što su prozori, meniji, polja za unos teksta i slično. UX (User Experience, ili iskustvo korisnika) se odnosi na kako se korisnik osjeća dok koristi neki proizvod ili sistem. To uključuje ne samo vizualni izgled, već i to koliko je proizvod ili sistem lako za korištenje, koliko je intuitivan i slično.

Bolja dostupnost postala je ključni aspekt modernih tehnologija. Stoga bi *web*-stranice i aplikacije trebale težiti da budu digitalno dostupne svim korisnicima.

2.4.6. Testiranje dokumentacije

Testiranje dokumentacije uključuje testiranje dokumentiranih artefakata koji se obično razvijaju prije ili tijekom testiranja softvera. Dokumentacija za testiranje softvera pomaže u procjeni potrebnog napora testiranja, pokrivenosti testom, praćenja zahtjeva itd. Često korišteni dokumentirani artefakata koji se odnose na razvoj i testiranje softvera su:

- plan testiranja
- zahtjevi
- testni slučajevi
- matrica sljedivosti.

2.5. Piramida testova

Korištenje **piramide testova** važan je dio strategije testiranja softvera. Piramida testova koncept je koji softverske testove grupira u tri različite kategorije. To pomaže razvojnim programerima i stručnjacima za osiguranje kvalitete da osiguraju višu kvalitetu softvera, skrate vrijeme potrebno za pronalaženje uzroka grešaka i izgrade pouzdaniji testni paket. Piramida testova strateški grupira softverske testove u različite kategorije. **Od vrha do dna, glavni slojevi piramide testiranja uključuju:**

- UI/istraživački testovi
- integracijski testovi
- jedinični testovi.

Donji dio piramide je automatiziran, što čini testove bržima. Vrh piramide je za sporije i manualne scenarije testiranja. Drugim riječima, testeri bi trebali imati mnogo jediničnih

testova, koji su kraći i izolirani, a manje istraživačkih testova koji su dugotrajni i ručni. Testovi s početnim učitavanjem na razini jedinice čine popravljanje grešaka bržim i lakšim, za razliku od nedostataka koji se otkrivaju mnogo kasnije u procesu.

2.5.1. Testabilnost

Testabilnost pokazuje koliko je praktično i lako testirati pojedinačnu komponentu i cijeli sustav. Recimo da se radi o aplikaciji za rezervaciju hotelskih soba. Da bi se ta aplikacija mogla testirati, treba imati određene karakteristike koje je čine testabilnom. To može uključivati:

- jedinstvenu **identifikaciju** svake sobe, kako bi se lako moglo testirati je li soba rezervirana ili nije
- funkciju za **poništavanje** rezervacije, kako bi se lako moglo testirati poništava li se rezervacija ispravno
- funkciju za **prikaz slobodnih soba**, kako bi se lako moglo testirati označavaju li se sobe pravilno kao rezervirane ili slobodne
- funkciju za **pretragu po različitim kriterijima**, kako bi se moglo testirati prikazuju li se rezultati pretrage pravilno.

Ovi elementi pomažu da se aplikacija lako testira i da se utvrdi funkcioniraju li ti elementi ispravno. Pitanja kojima se možemo voditi kod testabilnosti aplikacije:

- Koliko je lako kontrolirati unutarnje stanje komponenti tijekom testiranja (hoće li testiranje dovesti do prekida rada)?
- Je li teško promatrati sustav i njegove komponente tijekom testiranja (može li tester vidjeti ponašanja u sustavu tijekom testiranja)?
- Je li moguće testirati komponente u izolaciji (može li se testiranje omogućiti odvojeno od produkcijskog okruženja)?
- Je li komponenta potpuno dokumentirana? (provjeriti svu dokumentaciju vezanu uz komponentu)?

- Je li moguće pisati automatizirane testove za komponentu? (može li se automatizirati testiranje za komponentu kako bi se ubrzao postupak samog testiranja)?

2.5.2. Automatizirano testiranje korisničkog sučelja

Testiranje korisničkog sučelja (engl. *User Interface*, UI) je proces koji se koristi za testiranje ispravnosti aplikacije. Testiranje korisničkog sučelja može ručno izvršiti tester ili se može izvesti automatski pomoću softverskog programa.

Automatizirano testiranje korisničkog sučelja je automatizacija manualnih testnih slučajeva. Budući da manualni testni slučajevi mogu biti dugotrajni i skloni pogreškama, moguće je implementirati automatizirano testiranje korisničkog sučelja kao **točniju, učinkovitiju i pouzdaniju metodu**. Tijekom duljeg razdoblja, automatizirano testiranje korisničkog sučelja postaje isplativa zamjena za manualno testiranje. Međutim, treba ga implementirati nakon što je neki dio funkcionalnosti već završen (npr. rezultat sprinta).

Osim toga, automatizirani testovi korisničkog sučelja mogu se izvoditi istovremeno na različitim platformama i u različitim preglednicima. Automatizirani testovi korisničkog sučelja postaju puno važniji kada više ljudi radi na istom dijelu koda, iz očitih razloga. Bez dobre pokrivenosti automatiziranim testiranjem, kompliciranim dijelovi koda brzo postaju domena jedne osobe.

2.5.3. Automatizirano testiranje *web*-servisa i Rest API

Kod razvijanja programskih proizvoda kao što su internetske aplikacije, važan segment funkcioniranja su **internetski servisi,** poznati i kao **RESTful** servisi, REST API, SOAP, Web API itd. Oni predstavljaju programske jedinice koje se pokreću i izvršavaju na *web*-poslužiteljima i odgovaraju na HTTP zahtjeve klijenata. Osnovni uvjet za testiranje

internetskih servisa je dubinsko poznavanja rada internetskih tehnologija. Osnovna arhitektura interneta je ista, tako da su osnove na kojima počivaju internetski servisi poprilično standardizirani i učestali u svojoj primjeni. Potrebno je poznavati osnove mreža računala, a kroz njih specifično rad aplikacijskog, prezentacijskog, transportnog i mrežnog sloja internetske mreže. Pod osnovne HTTP zahtjeve ubrajamo **zahtjev primanja** (GET), **zahtjev slanja** (POST), **ažuriranja** (PUT) i **brisanja** (DELETE).

2.5.4. Automatizirano testiranje sustava pod naprezanjem

Testiranje sustava pod naprezanjem podrazumijeva izlaganje softverskog proizvoda mnogo težim uvjetima i upotrebi od one koja mu je predviđena u stvarnosti. Cilj je preopteretiti proizvod na različite načine kako bi se brzo ispravile pogreške.

Pitanja za ponavljanje:

1. Što je to piramida testova?
2. Nabrojite neke osnovne pojmove vezane uz softverskom testiranju.
3. Objasnite metodu sive kutije.
4. Koji su aspekti kvalitete softverskog proizvoda?
5. Što je testiranje pouzdanosti?
6. Što je specifično kod testiranja opterećenosti?
7. Kada se koristi metoda crne kutije i koji je njezin izlazni kriterij?
8. Što su Restful servisi?

3. Pristupi testiranju softvera

U OVOM POGLAVLJU NAUČIT ĆETE:

- što je testiranje upravljano ponašanjem
- kako kreirati scenarij pisan u jeziku Gherkin
- analizirati probleme strukture u testiranju
- što je to istraživačko testiranje
- izvoditi testove kako bi se otkrili rizici.

Dva su čimbenika važna za testiranja softvera: **strategija testiranja** te **ulaganje vremena i resursa u testiranje.** Strategija je na prvom mjestu. Nemoguće je odrediti ukupnu investiciju projekta bez znanja koje su tehnike i alati potrebni za testiranje *web*-stranice ili aplikacije. Prvi korak u određivanju strategije je jasna ideja o najučinkovitijim strategijama testiranja softvera. Različite strategije testiranja zahtijevaju specifične razine tehničkih vještina, znanja i alata. U nastavku poglavlja upoznat ćemo se s Testiranjem upravljanim ponašanjem kroz ciklus razvoja softvera. Prikazat ćemo kako se kreiraju testni scenariji koristeći poseban jezik za to i kako se takvo testiranje s istraživačkim kombinira u praksi.

3.1. Testiranje upravljano ponašanjem

Testiranje upravljano ponašanjem (engl*. Behaviour Driven Testing/Development,* **BDD**) je proces razvoja softvera koji je izvorno nastao iz metodologije Test vođen razvojem (engl*. Test Driven Development,* TDD).

3.1.1. BDD u ciklusu razvoja softvera

BDD koristi scenarije za prikaz ponašanja sustava. Ti scenariji su napisani čitljivim i razumljivim jezikom za sve uključene u razvoj softverskog proizvoda. Ovi primjeri uključuju:

- kreiranje scenarija na temelju specifikacija koje su napisane za razvoj softvera
- scenariji se koriste kao test prihvaćanja.

BDD se fokusira na:

- pružanje zajedničkog procesa i zajedničkih alata koji promiču komunikaciju sa softverom, programerima, poslovnim dijelom tima i vlasnicima proizvoda za suradnju na razvoju, s ciljem isporuke proizvoda koji ima poslovnu vrijednost, tj. zadovoljava specifikaciju koju je vlasnik proizvoda postavio prije početka razvoja aplikacije.
- ono što bi sustav trebao raditi, a ne kako bi se trebao implementirati
- pružanje bolje čitljivosti samih testnih scenarija
- rad softvera, ali i ispunjava li korisnikova očekivanja.

Cilj BDD-a je uočiti kvar prije plasiranja aplikacije na tržište jer se trošak popravka kvara višestruko povećava ako se kvar ne otkrije u pravo vrijeme. Dvije glavne prakse BDD-a su:

- specifikacija primjerom (engl*. Specification by example, SbE*)
- test vođen razvojem (TDD).

Specifikacija primjerom

Specifikacija primjerom (SbE) koristi primjere u razgovorima o softverskom proizvodu za ilustraciju poslovnih pravila i ponašanja softvera koji će biti izgrađen. Specifikacija primjerom omogućuje vlasnicima proizvoda, poslovnim analitičarima, testerima i programerima uklanjanje nesporazuma o poslovnim zahtjevima.

Test vođen razvojem

Test vođen razvojem, u kontekstu BDD-a, pretvara primjere scenarija u čitljive, izvršne specifikacije. Programer koristi ove specifikacije kao vodič za implementaciju novih funkcionalnosti. To rezultira jednostavnom bazom koda i paketom automatiziranih regresijskih testova koji održavaju niske troškove održavanja tijekom životnog vijeka softvera.

3.1.2. Gherkin jezik kao oblikovanje BDD scenarija

Gherkin je programski jezik koji se koristi za opisivanje funkcionalnosti softvera u okviru pristupa BDD-a. To je jednostavan format koji omogućava brzu komunikaciju između timova, programera i vlasnika proizvoda o očekivanim funkcionalnostima aplikacije. Svaki scenarij u Gherkinu opisan je korištenjem ključnih riječi poput *Given, When* i *Then* kako bi se opisao kontekst funkcionalnosti, akcije i rezultat. Ovi se scenariji koriste kao temelj za automatsko testiranje funkcionalnosti softvera.

Osnovne komponente Gherkina su:

- **značajka** (engl*. Feature*) - opisuje funkcionalnost softvera kroz jedan ili više scenarija
- **scenarij** (engl*. Scenario*) - sadrži korake koji definiraju neko ponašanje
- **korak** (engl*. Steps*) - jedan korak nekog ponašanja.

Ključne riječi koje se koriste u Gherkinu:

- značajka (engl. *Feature*)
- pravilo (engl. *Rule*)
- primjer (ili scenarij) (engl. *Example* ili *Scenario*)
- s obzirom, kada, tada, i, ali (*engl. Given, When, Then, And, But*)
- pozadina (engl*. Background*)
- nacrt scenarija (ili predložak scenarija) (engl*. Scenario Templates*).

3.1.3. Definiranje korisničkih zahtjeva korištenjem korisničkih priča

Glavni cilj testera je identificirati što više relevantnih rizika za neki proizvod, a zatim kroz njegovo testiranje skupiti informacije vezane uz te rizike. Definiranje korisničkih zahtjeva je izvrstan pristup za otkrivanje različitih rizika jer tester pritom analizira različite značajke na različite načine.

Scenarij: Uspješna prijava
S obzirom *da je korisnik na stranici za prijavu*
Kada *korisnik unese ispravno korisničko ime i lozinku*
I *korisnik klikne na gumb za prijavu*
Tada *će korisnik biti preusmjeren na početnu stranicu*
I *poruka dobrodošlice bit će prikazana*

Scenarij: Neuspješna prijava
S obzirom *da je korisnik na stranici za prijavu*
Kada *korisnik unese neispravno korisničko ime i lozinku*
I *kada klikne na gumb za prijavu*
Tada *bi trebala biti prikazana poruka o neuspješnoj prijavi*

3.1.4. BDD scenariji u testiranju

BDD scenarij je opis ponašanja proizvoda iz jedne ili više perspektiva korisnika. Scenariji su osmišljeni kako bi smanjili troškove i olakšali programerima razumijevanje zahtjeva te da se timu za testiranje olakša testiranje.

Korištenje BDD scenarija znači da se zahtjevi i testovi mogu kombinirati u jednu specifikaciju. U nekim slučajevima, napisani scenariji mogu se lako pretvoriti u automatizirane testove. BDD scenariji obično slijede određen format. Format je prilično jasan i nije težak za shvatiti, kao što ste vidjeli na prethodno opisanoj korisničkoj priči kod uspješnog i neuspješnog prijavljivanja.

3.1.5. Prednosti i ograničenja BDD-a

Kad govorimo o prednostima BDD-a, u prvi plan dolazi otkrivanje dodatnih mogućnosti i složenosti projekta. Budući da su se identificirali scenariji na početku, manje je prerade same funkcije ili logike nekog koda na kraju projekta. BDD je specifičan jer se fokusira na poslovni dio razvojnog procesa, a također je važan za definiranje specifikacija projekta, čime s izbjegavaju nesporazumi i poboljšava komunikacija.

Prednost BDD-a je u stvaranju zajedničkog jezika između razvojnog i testnog tima. Korištenjem istih fraza koje se koriste za specifikaciju zahtjeva, ubrzava se razvoj testova, a samim time se osigurava da su svi mogući scenariji testirani. Testiranje većih i složenijih aplikacija lakše je jer je stvoren jasan sistem koji je dovoljno razumljiv i nije dvosmislen kada se pristupa testiranju.

Uključivanje testera počinje od početka životnog ciklusa projekta, tj. faze specifikacije. Testiranje kreće samo kada postoji fizički proizvod za testiranje. Vlasnik proizvoda zna točno što želi dobiti isporukom određene funkcionalnosti. Funkcionalnost se tada rastavlja

na dijelove koji su puno lakši za implementaciju i testiranje. To znatno olakšava razvoj projekta i uklanja složenost projekta u ranijim fazama njegova razvoja.

Pomak ulijevo je popularan izraz za rano testiranje u razvojnom postupku. Rano testiranje znači manje grešaka kasnije. U BDD-u, definicija testnog slučaja se ispostavlja kao nužna faza (kod modela vodopada) ili pripreme (za agilni model). Kada su situacije ponašanja sastavljene, testiranje i automatizacija mogu započeti.

Ograničenja i nedostaci BDD pristupa:

- nije pogodan za projekte koje je potrebno završiti u kratkom vremenu
- zahtijeva dobru komunikaciju između osobe koja razvija kod za automatizaciju i osobe koja piše značajku ili funkcionalnost. Osoba koja piše automatizaciju treba datoteke i scenarije za razvoj skripte automatizacije. Ako nemaju međusobno razumijevanje datoteka, onda je teško razviti projekt.
- definira testne podatke koji olakšavaju stvaranje automatiziranih testnih slučajeva, ali kada se koristi za izvršavanje testnih slučajeva, stvara problem kada je testna okolina iz nekog razloga promijenjena. Testni slučajevi ovise o vanjskim podacima koji često uzrokuju problem kada izvršavamo testne slučajeve.
- vrlo je teško pretvoriti *Given* naredbe u upute za postavljanje i skripte koje stavljaju sustav u poznato stanje prije nego što se izvedu naredbe *When*
- potrebno je izraditi dokument za BDD projekt jer postoje mnoge datoteke i scenariji koje treba razumjeti kako bismo izradili dokumentaciju.

BDD pristup nudi velike prednosti: bolju komunikaciju, lakšu automatizaciju testiranja i veću kvalitetu koda. Postoji mnogo načina na kojem timovi mogu početi raditi BDD, a naravno, usvajanje BDD-a zahtijeva trud i prilagodbu.

Kad govorimo o nedostatku BDD-a, potrebno je puno vremena da se stvori specifikacija zahtjeva u obliku posebnih scenarija. Postavljanje scenarija zahtijeva precizno znanje o funkcionalnosti aplikacije. Isto tako prilikom testiranja manjih aplikacija mogu se stvoriti nepotrebni scenariji.

3.1.6. Suradničko definiranje BDD scenarija Tri amigos

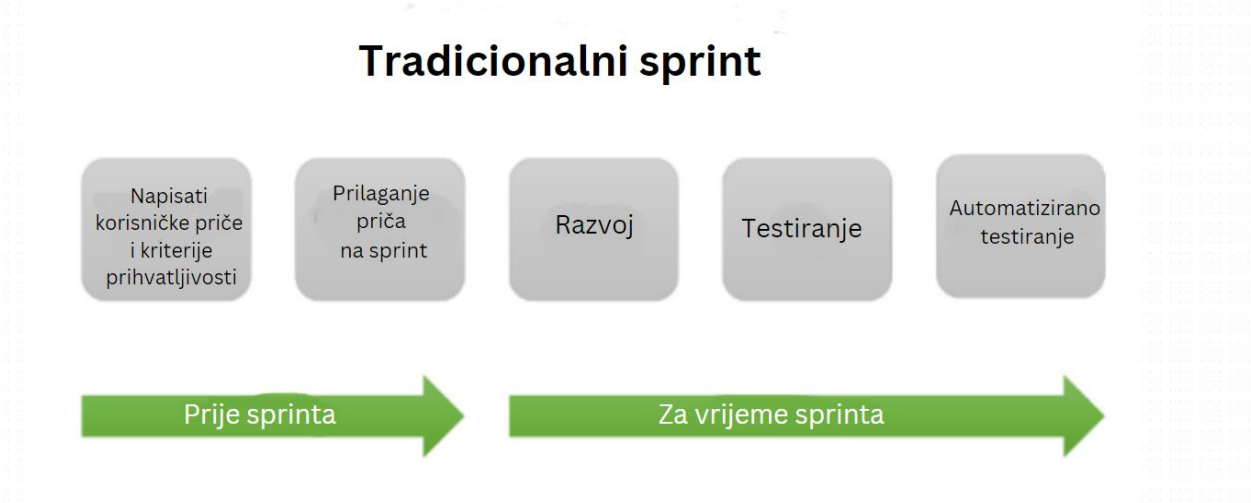
Tri Amigosa su sastanci koji se odnose na rad s tri strane (tri sudionika) koje sudjeluju u projektu. Na tim sastancima sudjeluju predstavnici iz različitih sektora koji se bave istim projektom, a cilj im je razmjenjivati informacije, dogovarati suradnju i planirati daljnji rad. Obično su to sastanci između tima razvoja softvera, tima proizvodnje i tima poslovanja.

U ovom ćemo dijelu predstaviti tri moguća načina za vođenje usvajanja BDD-a, svaki iz jedne od uloga Tri Amigosa. Bilo koja uloga može voditi, ali svaka će imati svoje jedinstvene prednosti i nedostatke.

Pristupi u nastavku pretpostavljaju da je temeljni proces razvoja softvera **Agile Scrum**. Unatoč tome, mogu se prilagoditi i primijeniti na druge procese, poput vodopada ili Kanbana.

Polazna točka

Polazna točka za sva tri pristupa u nastavku je "tradicionalni" Agilni sprint - onaj koji (još) nije vođen ponašanjem. Vlasnici proizvoda pišu korisničke priče, programeri implementiraju rješenja, a testeri testiraju rezultate. Donji dijagram prikazuje glavni tijek sprinterskog rada u ovoj vrsti *sprinta* i poslužit će kao osnova za ilustraciju usvajanja BDD-a:



Slika 3.2. Prikaz tradicionalnog sprinta

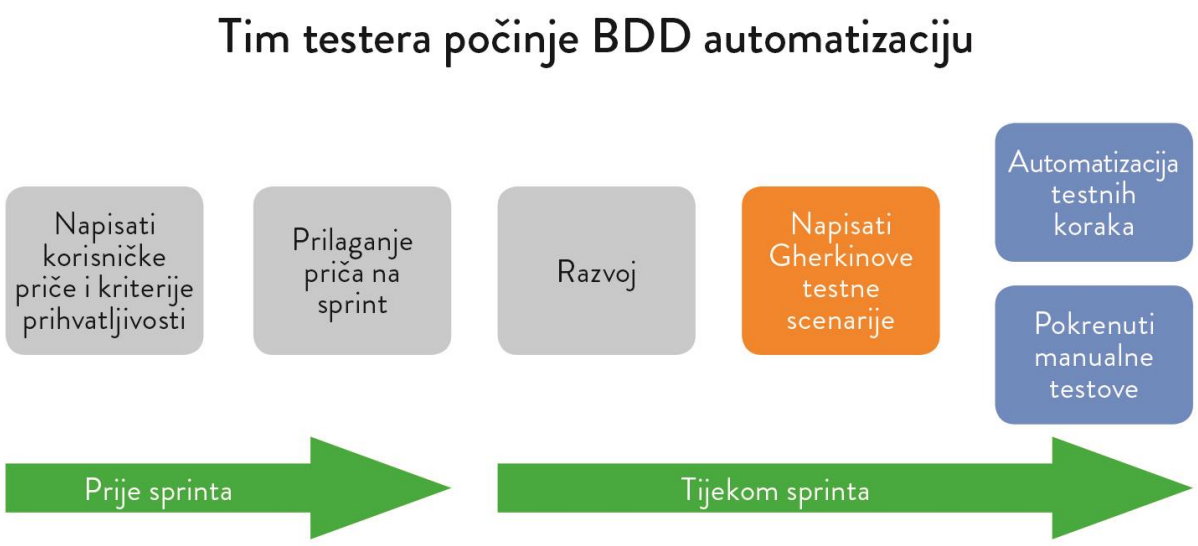
BDD pristup koji vode tim testera, programeri i poslovna strana znači da su svi ti timovi uključeni u razvoj projekta i da se fokusiraju na kvalitetu i ispravnost aplikacije. **Tim za testiranje** koristi BDD pristup da definira očekivano korištenje i funkcionalnosti aplikacije, dok **programeri** koriste te definicije kako bi razvili i testirali aplikaciju. **Poslovna strana** se fokusira na korištenje aplikacije u poslovne svrhe i na ispunjavanje poslovnih ciljeva. Ovaj pristup omogućuje da se različite perspektive uključe u razvoj projekta, što dovodi do boljeg razumijevanja potreba korištenja aplikacije i njenih funkcionalnosti. To također omogućuje bržu identifikaciju i brže rješavanje problema, što rezultira kvalitetnijom aplikacijom koja odgovara potrebama korisnika.

Kada glavnu ulogu kod vođenja razvoja ima tester

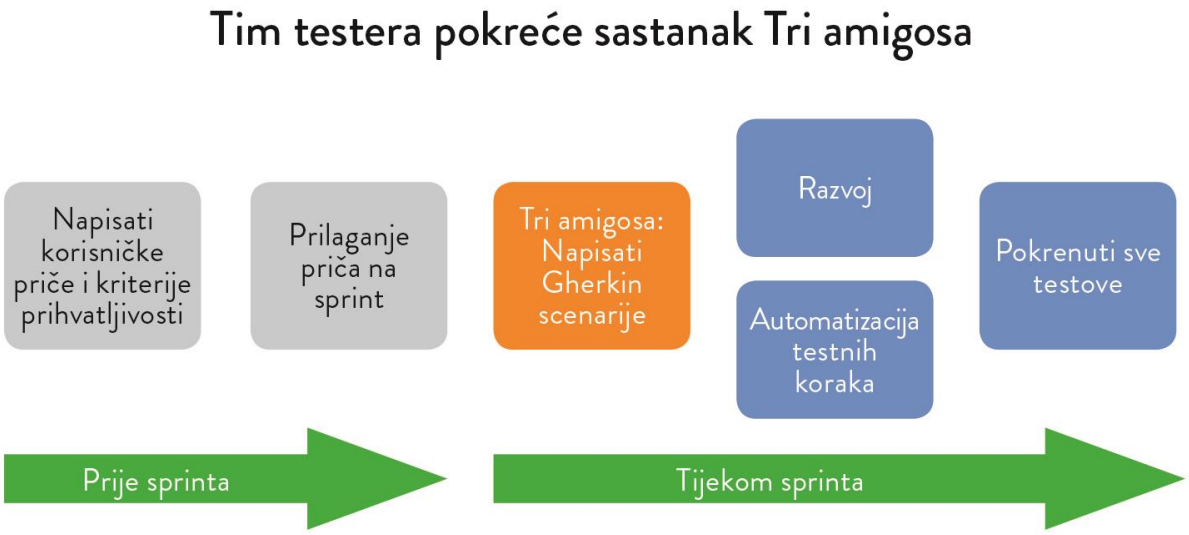
Tester započinje skroz ulijevo i postupno pomicati udesno, što je prikazano u slici dolje (slika 1.10.). To znači da bi početna točka bila automatizacija testiranja. Započeta je izgradnja čvrste baze kodova za automatizaciju. Odabere se dobro podržan BDD okvir kao što je Cucumber, SpecFlow ili neki drugi i počinju se dodavati scenariji i definicije koraka.

Nakon što kod za automatizaciju dosegne "kritičnu masu" za ponovnu upotrebu koraka, tester tada može proaktivno klasificirati nove testne scenarije kao automatizirane ili ručne. Automatizirani testovi postaju sve lakši za pisanje, dajući testeru više vremena za istraživanje uz ručno testiranje. U idealnom slučaju, sva ručna testiranja postala bi istraživačka, što će biti kasnije opisano u priručniku.

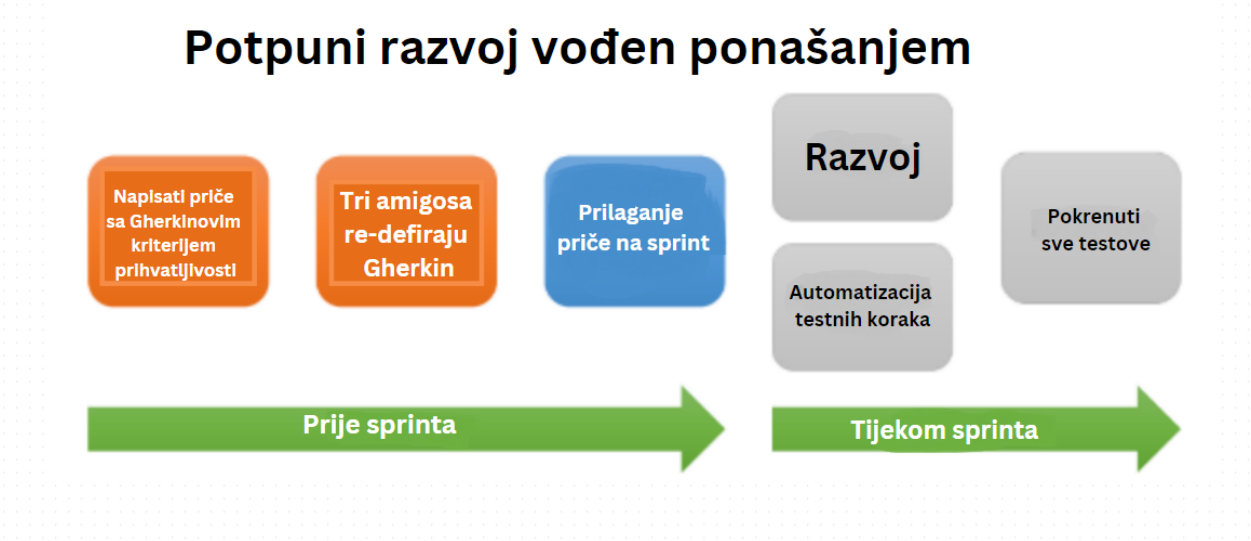
Zatim, vrijeme je da se počne pomicanje ulijevo. U ovom trenutku, svi koraci Gherkina bili bi samo u kodu za automatizaciju, stoga je važno postavljanje alata poput Cucumbera (alata koji omogućuje pisanje testnih slučajeva koje svatko može lako razumjeti bez obzira na tehničko znanje). Cucumber se koristi da bi se svim članovima tima koraci izložili kao dokumentacija. Tester bi tada trebao zakazati sastanak Tri Amigosa s poslovnim dijelom tima i programerima kako bi se definirala očekivanja korisničkih priča. Na tim sastancima tester bi trebao početi demonstrirati kako napisati kriterije prihvaćanja u Gherkinu, što zatim ubrzava testiranje. Idealno je kada tester može napisati novi scenarij koristeći samo postojeće, unaprijed automatizirane korake i zatim ga uspješno pokrenuti na licu mjesta.



Slika 3.3. Tim testera kreće s BDD automatizacijom od lijeve strane



Slika 3.4. QA tim vodi sastanak Tri Amigosa



Slika 3.5. Maksimalno prebacivanje ulijevo

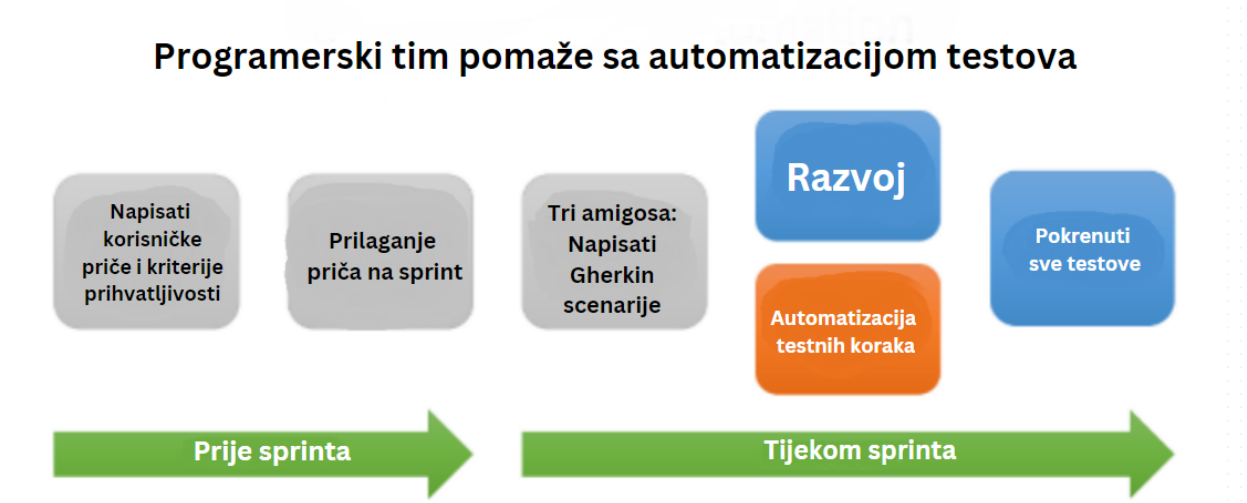
Postoji nekoliko razloga koji bi mogli potaknuti **programere** da vode BDD. U nekim agilnim timovima nema razlike između uloga programera i testera: svi članovi tima su softverski inženjeri odgovorni i za razvoj i za testiranje softvera. U takvim slučajevima

programeri možda nisu zadovoljni uloženim testiranjem. Bez obzira na okolnosti, programeri su više nego sposobni voditi razvoj softvera s BDD-om.

Kada glavnu ulogu kod vođenja razvoja kada ulogu ima programer

Najbolji način da programeri počnu je da sazovu sastanke Tri Amigosa kako bi komunikacija između poslovnog i razvojnog dijela tima bila što preciznija. Na tim sastancima počinju se prevoditi kriteriji prihvaćanja na Gherkin jezik. Zatim se počinje pomagati testerskom timu s automatizacijom testiranja.

Slika 3.6. Programer vodi sastanak Tri Amigosa



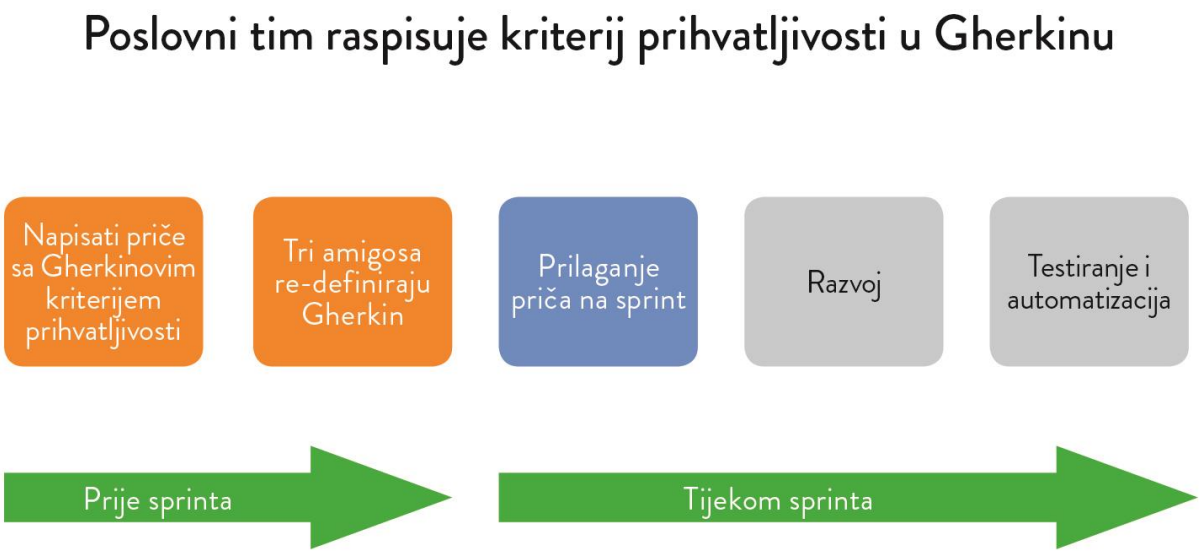
Slika 3.7. Pomak udesno: s nekom inicijalnom pripremom, programerski tim može pomoći testerskom timu u automatizaciji testnih koraka

Slika 3.8. Slika prikazuje sprint koji je vođen programerskim dijelom tima

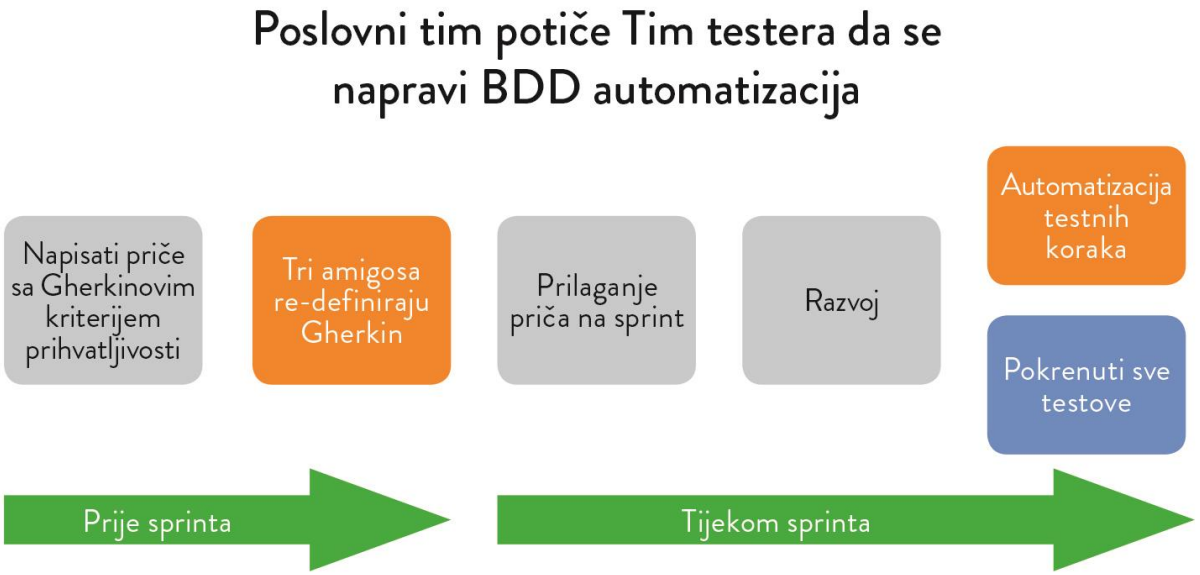
Kada glavnu ulogu kod vođenja razvoja ima poslovni dio tima

Za poslovne uloge, "pomak ulijevo" započinje pisanjem korisničkih priča i kriterija prihvaćanja. Usredotočenost se svodi na dobar Gherkin jezik koji je čitljiv i za višekratnu upotrebu. Zatim se predstavlja BDD kao usavršavanje Agilnog procesa, a kojim se ističu njegove prednosti. Pokreće se sastanak Tri Amigosa, a nakon što suradnja krene dobro,

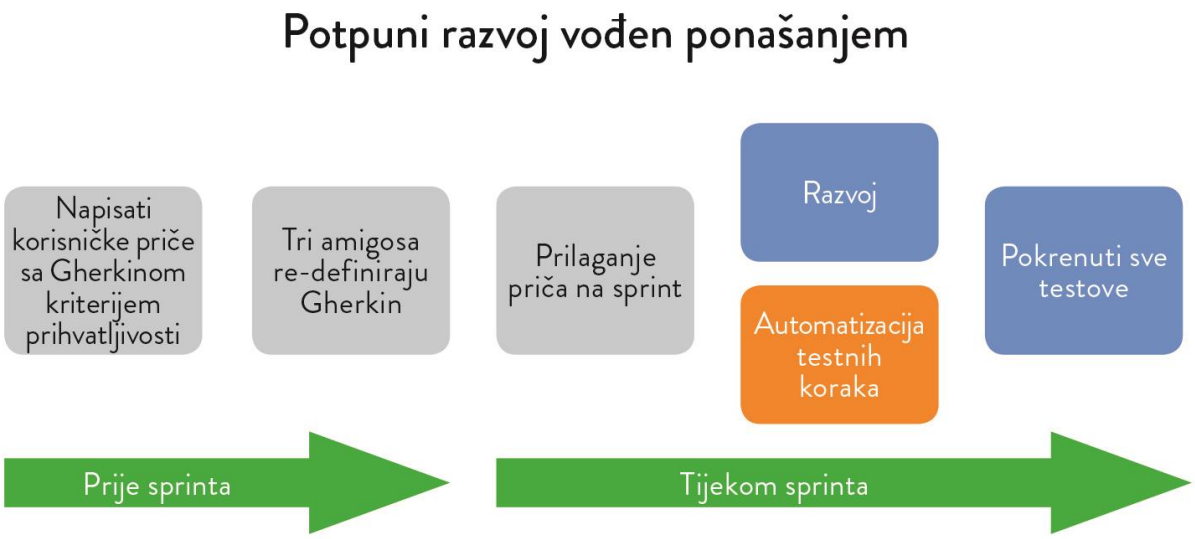
predlaže se automatizacija BDD-a kao način da se ubrza razvojni i testni rad. Ako su svi kriteriji prihvaćeni, razvoj BDD automatizacije testova bio bi logični nastavak.



Slika 3.9. Poslovni dio tima počinje s razvojnim procesom tako što piše korisničke zahtjeve u Gherkinu.



Slika 3.10. Poslovni dio tima potiče tim testera da se odradi BDD automatizacija



Slika 3.11. Prikaz sprinta vođen poslovnim dijelom tima

3.2. Istraživačko testiranje (engl. *Exploratory testing*)

Istraživačko testiranje je vrsta testiranja softvera u kojem se testni slučajevi ne kreiraju unaprijed, već testeri provjeravaju sustav u hodu. Mogu zabilježiti ideje o tome što testirati prije izvođenja testa. Fokus istraživačkog testiranja je više na testiranju kao "misaonoj" aktivnosti. Istraživačko testiranje uveliko se koristi u agilnim modelima i zasniva se o otkrivanju, istraživanju i učenju. Naglašava osobnu slobodu i odgovornost pojedinog testera.

Prednosti istraživačkog testiranja:

- ubrzava razvoj testova jer se mogu napraviti brži testovi
- prilagođava se prema uvjetima u trenutku kada se testira
- omogućava brže testiranje i učinkovitiju upotrebu vremena
- pomaže u pronalaženju unaprijed nepoznatih problema
- omogućava bržu detekciju pogrešaka u usporedbi s automatiziranim testovima.

Nedostaci istraživačkog testiranja:

- ne postoji jasan plan rada i moguće je da se nešto propusti
- postoji veći rizik da se neki scenariji ne ispitaju
- zbog različitih interpretacija istog testa, moguće je da se propuste pogreške
- nije moguće testirati sve scenarije.

3.2.1. Istraživačko testiranje kao konkurentno dizajniranje i izvršavanje testova u svrhu otkrivanja rizika

Istraživačko testiranje je pristup testiranja softvera koji se odnosi na otkrivanje i istraživanje aplikacije. Oslanja se na **smjernice pojedinačnog testera** kako bi otkrio nedostatke koji nisu obuhvaćeni drugim testovima. Istraživačko testiranje spaja procese učenja (o programu) i testiranja. Istraživačko testiranje osigurava da:

- tester ima sveobuhvatno razumijevanje o tome kako softver treba raditi
- da je tester dobro opremljen za izradu učinkovitijih testova koji brzo identificiraju glavne nedostatke u sustavu
- da se identificiraju nedostaci koji možda nisu inače pronađeni prilikom strukturiranog testiranja.

U nedostatku formalnih testnih slučajeva ili scenarija, sve ovisi o testeru. Ovisno o projektu na kojem se radi, mogu se zabilježiti ideje prije testa ili testirati u hodu. Oba pristupa dobro funkcioniraju, budući da je fokus istraživačkog testiranja na misaonoj aktivnosti umjesto na unaprijed propisanoj definiciji.

3.2.2. Formalnost i neformalnost u testiranju

Testiranje softvera može se obaviti na dva načina - **manualno (ručno)** ili **automatizirano**. Metodu manualnog testiranja možemo koristiti i za formalno i za

neformalno testiranje, a automatsko testiranje ne preporučuje se kod neformalnog testiranja.

Formalno testiranje softvera odnosi se na testiranje koje se provodi s planom, dokumentiranim skupom testnih slučajeva itd. Testna dokumentacija može se razviti iz zahtjeva, dizajna, podjele ekvivalencije, pokrivenosti domene itd. Razina formalnosti i temeljitosti testnih slučajeva ovisit će o potrebama projekta. Formalno testiranje slijedi sustavni proces u kojem postoje različite faze kao što su analiza zahtjeva, planiranje testa, razvoj testnog slučaja, postavljanje testnog okruženja, izvođenje testa i zatvaranje testnog ciklusa.

Neformalno testiranje softvera ili *ad hoc* testiranje provodi se bez dokumentiranog skupa ciljeva ili planova i oslanja se na intuiciju i vještine testera koji provodi testiranje. Iskusni testeri provode neformalno testiranje koristeći svoje prethodno iskustvo i tehnike testiranja temeljene na iskustvu kao što su istraživačko testiranje i nagađanje pogreške itd. Imaju dobro znanje o softveru koji se testira i kako radi. To je neformalna ili nestrukturirana vrsta testiranja softvera koja se provodi nasumično i obično je neplanirana aktivnost koja ne prati nikakvu dokumentaciju i tehnike dizajna testa za izradu testnih slučajeva.

3.2.3. Problem strukture u testiranju

Problemi strukture u testiranju mogu dovesti do problema s kvalitetom proizvoda, nedovoljne pouzdanosti i niskih performansi što može uzrokovati nezadovoljstvo korisnika. Stoga je važno osigurati da se testovi organiziraju i planiraju na način koji omogućuje efikasno i učinkovito testiranje proizvoda.

Današnji testni sustavi pružaju raznolike testne platforme koje se lako prilagođavaju testnoj okolini kod različitih timova koji testiraju aplikacije. Potreba za rješavanjem sve veće raznolikosti izazova testiranja proizvoda dovela je do vrlo velikog broja takvih

platformi. Odluka o odabiru testne platforme isključivo je na samom timu testera koji će na takvim platformama raditi.

Problem strukture u testiranju nastaje ako su testovi:

- nepotpuni - ako testovi ne obuhvaćaju sve slučajeve i scenarije
- netočni - testovi koji imaju pogrešne pretpostavke dovode do problema strukture u testiranju
- nedosljedni - neusklađenost testova i njegovih standarda jedan su od glavnih problema kod strukture u testiranju
- neefikasni - dolazi do problema u identificiranju problema
- nedovoljno automatizirani testovi - testovi kod manualnog testiranja zahtijevaju više vremena i truda
- nedostatak pokrivenosti testova - funkcionalnost ili kod nisu pokriveni testom pa neće biti testiran.

3.2.4. Razlika između provjeravanja i testiranja

Testiranje i provjera koriste se kako bi se procijenilo je li proizvod u skladu sa specifikacijama i radi li prema očekivanjima. Oba postupka izvode se prije isporuke proizvoda kako bi se osiguralo da je proizvod ispravan i originalan.

	Testiranje	**Provjeravanje**
Definicija	Proces provjere valjanosti proizvoda i kako ga poboljšati tijekom razvojnog procesa	Proces u kojem se provjerava određena funkcionalnost za koju se očekuje da će biti prisutna u proizvodu, radi li dobro ili ne. To se općenito radi za provjeru značajki proizvoda koji dobro radi prije promjene koda.
Područje pokriveno testom	Fokusira se na područje funkcionalnosti koje je novorazvijeno u skladu s novim zahtjevima ili poboljšanjima u proizvodu.	Usmjerena na osiguravanje očekivanog rada postojeće funkcionalnosti
Tko izvodi testove/ provjeru?	Testeri.	Osoba zadužena za provjeru, kontrolor.
Ishod	Predstavljen je kao očekivan ili ne, ishod se uspoređuje s unaprijed definiranim očekivanim rezultatom.	Predstavljen je kao DA ili NE, tj. radi li određena funkcija dobro ili ne.
Ažuriranje	Tijekom testiranja testeri ne moraju ažurirati što se	Kontrolori moraju biti obaviješteni o ažuriranju

	događa u drugim područjima proizvoda ili koja su nova ažuriranja koja su se nedavno dogodila.	softvera jer trebaju provjeriti funkcionalnost nakon ažuriranja, odnosno radi li dobro ili ne.
Osiguranje kvalitete	Testiranje proizvoda pruža osiguranje kvalitete koje pomaže programerima i voditeljima projekata da odluče koliko nedostataka ili grešaka njihov proizvod trenutno ima.	Provjera je praksa osiguranja kvalitete koju provode programeri.

3.2.5. Čarter (povelja) u istraživačkom testiranju

Čarter povelja u istraživačkom testiranju je dokument koji definira ciljeve, zadatke i odgovornosti svih uključenih strana u procesu istraživanja i testiranja. To uključuje programere, testere, korisnike i druge zainteresirane strane. Čarter povelja također može sadržavati informacije o načinu na koji će se provesti testiranje, kriterije za uspješnost i planove za praćenje i izvještavanje. Svrha ovog dokumenta je osigurati jasne smjernice i osigurati da svi uključeni razumiju svoje uloge i obveze.

Istraživačko testiranje temelji se na umijeću odlučivanja o testiranju koje će se obaviti, a tester ovisno o njegovom prethodnom i stečenom iskustvu zna najbolje kako pristupiti istraživačkom testiranju kako bi se pokrio što veći broj testnih slučajeva. Ove odluke testeri temelje na saznanjima i iskustvu testiranja prijašnjih aplikacija. Može se temeljiti na poznatim slabostima u bazi koda, prethodnim greškama i karakteristikama te

integracijskim točkama koje su nedavno implementirane. Povelje određuju svrhu određenog istraživanja, kao i opseg.

Istražite <cilj>
S <resursima>
Da bi se otkrila <informacija>

Kod ovakvog predloška, timovi koji se bave testiranjem aplikacije ili softvera ovlašteni su tretirati svoje povelje o istraživačkom testiranju kao prvorazredne priče. Povelje se mogu poredati po prioritetu među značajkama, poslovima i greškama prema vrijednosti koju pružaju.

Istražite <cilj>:
Cilj može biti određena značajka, područje koda, novi paket koji je upravo uveden u sustav, točka integracije itd.

S <resursima>:
Ovo je način na koji se definiraju alati koje bi tester trebao koristiti, kao i karakteristike ciljanih ulaznih podataka, kao što su neuobičajena kodiranja znakova. Odjeljak s resursima također se može koristiti za definiranje ograničenja ili uvjeta kako bi se izbjeglo pretjerano testiranje ili kako bi se testiranje vremenski uokvirilo.

Otkriti <informacije>:
Ovo je svrha povelje. Želimo li saznati radi li aplikacija ispravno, je li pouzdana, sigurna, jednostavna za korištenje i ima li tijek rada smisla.

3.2.6. Heuristike za istraživačko testiranje u sesijama

Heuristike za istraživačko testiranje u sesijama su pravila i naznake koje se koriste prilikom testiranja softvera kako bi se identificirale i ispravile potencijalne greške. Ovaj

pristup pridonosi učinkovitom i efikasnom testiranju softvera. Budući da se može lako primijeniti u više područja, uključujući i testiranje softvera, heuristička tehnika može se primijeniti kroz čitav ciklus životnog razvoja softvera. Glavna prednost ove metode svakako je mogućnost upotrebe u svim fazama razvoja aplikacije, intuitivna i jeftina primjena te brza i djelotvorna identifikacija glavnih i sporednih problema upotrebljivosti. S druge strane, ova metoda ima i nedostataka, a najveći je njena ovisnost o iskustvu i vještinama testera.

3.2.7. Prednosti i ograničenja istraživačkog pristupa testiranju

Ovakvo testiranje je korisno kada dokument o zahtjevima aplikacije nisu dostupni ili su dostupni samo djelomično. Istraživačko testiranje uključuje proces istraživanja koji pomaže u pronalaženju više grešaka od uobičajenog testiranja:

- otkriva greške koje druge tehnike testiranja obično zanemaruju
- potiče maštu testera izvođenjem sve više i više testnih slučajeva što konačno poboljšava i produktivnost.

Istraživačko testiranje pokriva sve vrste testiranja, kao i različite scenarije i slučajeve, potiče kreativnost i intuiciju te generira nove ideje tijekom samog izvođenja testa. Nedostatak ovakvog testiranja jest što ovisi isključivo o vještini testera i njegovom poznavanju testiranja.

Istraživačko testiranje ima sljedeće prednosti:

- bolje razumijevanje korisničkog iskustva: istraživanjem korisničkog iskustva stječe se jasnije razumijevanje potreba i navika korisnika, što omogućuje da kvalitetniji razvoj proizvoda
- povećanje kvalitete proizvoda: razumijevanje korisničkog iskustva i potreba pridonosi rješavanju problema prije nego što dođu do korisnika, što povećava kvalitetu proizvoda

- povećanje pouzdanosti: istraživački pristup testiranju omogućuje ispitivanje funkcionalnosti i performansi proizvoda u realnim uvjetima, što povećava njegovu pouzdanost
- razvijanje kreativnih rješenja: ispitivanje korisničkog iskustva i potreba može pomoći u razvijanju kreativnih rješenja koja odgovaraju potrebama korisnika.

Međutim, postoje i neka ograničenja istraživačkog pristupa testiranju, a to mogu biti:

- visoki troškovi: istraživački pristup testiranju može biti skup, osobito ako se obuhvaća veliki broj korisnika ili ako se provodi u različitim geografskim područjima
- vrijeme potrebno za prikupljanje i analizu podataka: istraživanje korisničkog iskustva može potrajati duže vrijeme, što može dovesti do odgode u razvoju proizvoda
- nepredvidivi rezultati: moguće je da istraživanje ne daje očekivane rezultate, što može dovesti do potrebe za ponovnim istraživanjem.

3.3. Kombiniranje BDD i istraživačkog testiranja

Kombiniranje BDD i istraživačkog testiranja je pristup testiranju koji kombinira testiranje temeljeno na hipotezama BDD-a i istraživanju. Kod kombiniranog testiranja možemo zaključiti da će tester imati pretpostavku prema kojoj će prilikom istraživačkog testiranja unaprijed pretpostaviti rezultat svojeg testa. Primjer: tester će kod funkcionalnosti za prijavu (*login*) pretpostaviti da će sljedeći korak biti uspješna/neuspješna prijava, a ne korak za registraciju. Ovaj pristup omogućuje testerima da koriste svoju intuiciju, iskustvo i znanje kako bi kreirali testne scenarije koji će pokriti što veći broj mogućih slučajeva.

BDD se može primijeniti na testiranje aplikacija kako bi se osiguralo da se svi scenariji testiraju i da se sve nove funkcionalnosti aplikacije isprobaju. BDD će pomoći razvojnom i testnom timu da stvore specifikaciju zahtjeva uključujući i sve moguće scenarije te da isprobaju aplikaciju u svim mogućim situacijama, dok se istraživačko testiranje može primijeniti na testiranje aplikacije kako bi se osiguralo da su sve funkcionalnosti ispravne.

Istraživačko testiranje pomoći će u pronalaženju unaprijed nepoznatih problema i u detekciji pogrešaka brže u usporedbi s automatiziranim testovima.

3.3.1. Defekti odstupanja od specifikacije i defekti rizika

Kako je već navedeno, defekt je greška ili nedostatak u softverskom proizvodu koji se otkriva tijekom testiranja. Ako se defekti ne otkriju i ne isprave, oni mogu uzrokovati problemu primjeni softvera te tako umanjiti kvalitetu proizvoda. Defekti se dijele u dvije skupine - **defekti odstupanja od specifikacije** i **defekti rizika**, a mogu se pojaviti tijekom razvoja i testiranja sustava.

Defekti odstupanja od specifikacije su problemi koji se javljaju kada se isporučeni proizvod ne podudara sa specifikacijom koja je napisana kod planiranja i kreiranja samog proizvoda. To može biti bilo koji problem u kvaliteti proizvoda ili pružanja usluge. To su nedostaci koji nastaju kada softver ne ispunjava zahtjeve i specifikacije navedene u dokumentu dizajna. Ovi nedostaci mogu rezultirati neispravnim ili neočekivanim ponašanjem u sustavu

Defekti rizika su problemi koji se javljaju kada dođe do neželjenih ponašanja u proizvodu ili usluzi. To mogu biti neke nesreće, nedostatak predviđanja ponašanja softvera ili aplikacije, nedovoljne zaštitne mjere i slično. Ovi se defekti mogu pojaviti zbog nedostatka znanja ili nedostatka iskustva. To su nedostaci koji se identificiraju na temelju potencijalnog utjecaja kvara. Defekti rizika imaju prioritet na temelju vjerojatnosti i ozbiljnosti utjecaja, s ciljem da se testiranje prvo usmjeri na visokorizična područja.

3.3.2. Istraživačko testiranje u sesijama koristeći BDD scenarije kao izvor informacija za pronalaženje rizika

BDD scenariji posebno su korisni za istraživačko testiranje jer omogućavaju testerima da poštuju oblik i sadržaj koji je potreban za razumijevanje prirodnog govornog jezika. BDD scenariji obično su oblikovani kao "ako/onda" izjave, što znači da se određeni scenariji mogu lako usporediti sa specifikacijom za softver i identificirati bilo kakva neslaganja. Kao takvi, oni se mogu koristiti kao sredstvo za pronalaženje rizika i ranjivosti. U istraživanju, BDD scenariji se mogu koristiti kao izvor informacija ili za simulaciju stvarnih i mogućih scenarija koji bi se mogli dogoditi kako bi se identificirali potencijalni problemi i ranjivosti.

Istraživačko testiranje u sesijama koje koriste BDD scenarije kao izvor informacija za otkrivanje rizika i generiranje testnih slučajeva može biti vrlo učinkovita kombinacija. BDD scenariji pružaju osnovu za razumijevanje sustava i svih potencijalnih rizika povezanih s njim. Ovo se znanje zatim može koristiti za kreiranje istraživačkih sesija testiranja koje su usmjerene na otkrivanje rizika i generiranje testnih slučajeva na temelju tih rizika.

Sesije se mogu strukturirati oko određenih područja sustava i koristiti BDD kao izvor informacija za generiranje testnih slučajeva i provjeru valjanosti rezultata. Korištenjem BDD scenarija kao temelja za istraživačko testiranje, testeri mogu brzo identificirati područja rizika i izraditi testne slučajeve na temelju tih rizika. To pomaže osigurati da su testovi sveobuhvatni i da mogu otkriti sve potencijalne probleme. Nadalje, pomaže osigurati da su testovi ponovljivi i da se mogu koristiti za provjeru bilo kakvih promjena napravljenih na sustavu.

3.3.3. Heurističke tehnike za kreiranje testova

Heuristika testiranja je pristup testiranju koji koristi pravila, savjete i iskustvo kako bi se identificirali i testirali mogući problemi u softveru. Ne bazira se na formalnom postupku, već se temelji na iskustvu i intuiciji testera. Heurističko testiranje koristi se za brzo

identificiranje i ispravljanje problema u softveru, pomaže u smanjivanju vremena i troškova u testiranju i povećava kvalitetu softvera.

Za izradu vlastite heuristike testiranja softvera potrebno je pratiti ova tri koraka:

1. Prepoznavanje uzoraka

Uzimajući u obzir da nam heuristika testiranja softvera pomaže zapamtiti stvari koje bismo trebali testirati i stoga ćemo ih ponavljati, prepoznavanje uzoraka u sesijama testiranja najsigurniji je način za stvaranje nove heuristike. Možemo razmisliti o tome zašto se pokrenula neka vrsta testova, zašto su radnje izvršene određenim redoslijedom, a zatim pokušati identificirati ponašanja koje se događaju u tim radnjama.

2. Stvaranje lako pamtljivih riječi ili rečenica

Korištenjem identificiranih uzoraka, potrebno je stvoriti pojmove (riječi ili rečenice) koji će omogućiti da se zapamte ciljevi testa i da netko tko nije stručna osoba može izvršiti test praćenjem koraka.

3. Stvaranje mnemotehnike

Napraviti akronim ili riječ koja predstavlja načela heuristike jer će biti lakše zapamćen.

Pitanja za ponavljanje:

1. Što je BDD i koja mu je funkcija u procesu testiranja?
2. Kakva je Gherkin struktura pisanja scenarija?
3. Objasnite čarter povelju.
4. Koje su prednosti i nedostatci BDD-a?
5. Objasnite Heurističke tehnike za kreiranje testova.

4. Proces testiranja softvera na primjeru projektnog zadatka

U OVOM POGLAVLJU NAUČIT ĆETE:

- kako osmisliti plan testiranja
- kako primijeniti metode testiranja bijele i crne kutije
- izvršavati različite metode kod primjera projektnog zadatka
- analizirati izlazne kriterije za test
- analizirati praćenje statusa testa.

4.1. Proces testiranja

U ovoj cjelini predstavit ćemo proces testiranja softvera na konkretnom primjeru. Kako bi proces bio uspješan, potrebno je okruženje za samo testiranje **JDK 17 (engl*. Java Development Kit*).** Ako ga nemate na vlastitim računalima, potrebno ga je preuzeti s interneta.

Maven je alat za automatizaciju izgradnje primarno za Java projekte. Definira strukturu projekta, ovisnosti, redoslijed izgradnje i druge procese izgradnje. Maven pomaže u upravljanju izgradnjom, izvještavanjem i dokumentacijom projekta iz jednog središnjeg izvora informacija.

IntelliJ je integrirano okruženje za razvoj kod Jave i drugih programskih jezika. Pruža niz funkcija za razvoj, uključujući dopunjavanje koda, isticanje pogrešaka, preuređivanje i ispravljanje pogrešaka, kao i sučelje koje se jednostavno koristi.

Browser driveri su posebni programi koji omogućavaju automatizaciju interakcije s *web*-preglednicima, kao što su Google Chrome, Mozilla Firefox ili Safari. Oni omogućavaju testiranje *web*-aplikacija i simulaciju radnji korisnika, kao što su klikanje na poveznice, unos teksta ili prebacivanje između stranica. *Browser driveri* često su korišteni u kombinaciji s različitim programskim jezicima i alatima za automatizaciju, kao što su Selenium, Cucumber ili JUnit, kako bi se osiguralo da *web*-aplikacije funkcioniraju ispravno na različitim preglednicima i platformama.

4.1.1. Testiranje u fazi planiranja - prve verzije BDD

U ovom dijelu pozabavit ćemo se značajkama i prvim verzijama BDD scenarija. Prva verzija BDD scenarija izgledat će ovako, a koristit će se za funkcionalnost prijave (*login*).

Značajka: Značajka prijavljivanja
Scenarij: Korisnik je uspješno prijavljen
S obzirom da je korisnik na stranici za prijavu
Kada korisnik unese ispravne podatke
Tada će korisnik uspješno biti prijavljen

4.1.2. Testiranje u fazi dizajna

U ovoj fazi redefiniramo sami scenarij značajke prijavljivanja kako bi se pokrile veće značajke samog testiranja.

Značajka: Značajka prijavljivanja
Scenarij: Korisnik je uspješno prijavljen
S obzirom da je korisnik na stranici za prijavu
Kada korisnik unese ispravne podatke
Tada će korisniku biti odobren pristup web-stranici

4.1.3. Struktura sastanka Tri Amigosa

Kako smo opisali uloge svakog od Tri Amigosa u trećem poglavlju (Pristupi testiranju softvera), znamo što koja uloga donosi kod takvog pristupa. Mi ćemo se postaviti u ulogu tima testera, što znači da ćemo kod strukture sastanaka krenuti s testiranjem u najranijoj fazi razvoja funkcionalnosti.

1. **Početni sastanak** - tijekom ovog sastanka tim će raspravljati o opsegu projekta, vremenskom okviru i resursima potrebnim za završetak projekta.
2. **Sastanak o dizajnu** - tijekom ovog sastanka tim će raspravljati o dizajnu značajke prijave, uključujući korisničko iskustvo, korisničko sučelje i tehničke zahtjeve.
3. **Razvojni sastanak** - tijekom ovog sastanka tim će raspravljati o razvoju značajke prijave, uključujući kodiranje, testiranje i otklanjanje pogrešaka.
4. **Završni sastanak** - tijekom ovog sastanka, tim će pregledati značajku prijave, raspraviti sva neriješena pitanja i finalizirati projekt.

4.1.4. Primjena metoda bijele kutije za testiranje koda

U ovom primjeru primijenit ćemo metodu bijele kutije za testiranje koda te će se naš testni scenarij sastojati od pozitivnog i negativnog slučaja, a izgledat će ovako:

//Testni slučaj 1 - Pozitivan proces
//Ulazni kriterij: // korisničko ime: testKorisnik
//lozinka : testLozinka
//Očekivani rezultat :// uspješna prijava
//Aktualni rezultat: // uspješna prijava

//Testni slučaj 2 - Negativan proces
//Ulazni kriterij: // korisničko ime: testKorisnik
//lozinka : pogrešnaLozinka
//Očekivani rezultat: // neuspješna prijava
//Aktualni rezultat: // neuspješna prijava

Primjena metode bijele kutije u ovom slučaju daje nam očekivani i aktualni rezultat kod testnog slučaja gdje se testira pozitivan proces (uspješna prijava) i negativan proces (neuspješna prijava).

4.1.5. Primjena metoda crne kutije za testiranje softverskog proizvoda za vrijeme razvoja

Kod primjene ove metode moramo imati na umu naše ulazne kriterije za samo testiranje (u ovom slučaju to je funkcija prijave). Naši ulazni kriteriji pomoći će nam da odredimo izlazni kriterij (uspješna prijava) i moći ćemo usporediti i analizirati ako nastane problem (kada se korisnik ne može prijaviti jer nije unio ispravne podatke).

Značajka: Prijava
Scenarij: Uspješna prijava
S obzirom da sam na stranici za prijavu
Kada unesem ispravne podatke
Tada moram biti odveden na početnu stranicu

Značajka: Neuspješna prijava
Scenarij: Neuspješna prijava
S obzirom da sam na stranici za prijavu
Kada unesem neispravne podatke
Tada moram dobiti poruku da su podaci neispravni

Značajka: Prijava
Nacrt scenarija: Uspješna prijava
S obzirom sam na stranici <primjer stranice>
Kada unesem ispravne <podatke>
Tada moram biti odveden na <stranicu>
Primjer:
| stranica | podaci | stranica |
| stranica za prijavu | username/password | Početna stranica |

Primjenom metode crne kutije, gdje smo koristili **ulazne kriterije** koji su morali biti pokriveni našim testom (uspješna/neuspješna prijava), kao rezultat dobili smo **konačne izlazne kriterije** s točnim rezultatima koji su u ovom slučaju prikazani u obliku puta kojim će korisnik biti preusmjeren na željenu početnu stranicu.

4.1.6. Primjena heurističkih tehnika za istraživačko testiranje

Primjena heurističkih tehnika omogućava testerima da u kratkom roku identificiraju što više problema u aplikaciji. U ovom primjeru naše bi heurističke tehnike za istraživačko testiranje ove funkcionalnosti bile:

- pokušajte se prijaviti s nevažećim podacima kako biste bili sigurni da će sustav izbaciti pogrešku
- pokušajte unijeti prazne podatke za prijavu i provjerite javlja li sustav pogrešku
- pokušajte unijeti korisničko ime i lozinku obrnutim redoslijedom kako biste provjerili javlja li sustav pogrešku
- pokušajte koristiti posebne znakove u polju korisničkog imena i lozinke kako biste bili sigurni da će sustav izbaciti pogrešku
- pokušajte se prijaviti s prethodno korištenim korisničkim imenom kako biste bili sigurni da vas sustav prijavljuje
- provjerite dopušta li sustav više istovremenih prijava s istim korisničkim imenom.
- prije zaključavanja korisnika, provjerite dopušta li sustav više neuspjelih pokušaja prijave
- provjerite dopušta li sustav korisniku ponovno postavljanje lozinke ako je zaboravi.
- provjerite postoji li vrijeme čekanja neaktivne sesije
- pokušajte se prijaviti s onemogućenim računom kako biste bili sigurni da će sustav izbaciti pogrešku.

4.1.7. Izlazni kriteriji za test

Izlazni kriteriji su standardi koji moraju biti zadovoljeni da bi se zaključilo da je test uspješan. Kako bi testiranje imalo izlazne kriterije, moraju biti potvrđene ove stavke:

1. svi testovi trebaju biti uspješni.
2. svi važeći korisnički podaci trebali bi voditi uspješnoj prijavi.
3. svi nevažeći korisnički podaci ne bi trebali uspješno proći prijavu.
4. sve poruke o pogreškama trebaju biti ispravno prikazane.

5. svi elementi na stranici za prijavu trebaju biti ispravno prikazani.
6. sve funkcije povezane sa stranicom za prijavu trebale bi ispravno raditi.
7. stranicu treba zaštititi od bilo kakvih zlonamjernih napada.

4.1.8. Regresijsko testiranje

Kod **regresijskog testiranja** za značajku prijavljivanja bit će uključeni sljedeći testni scenariji:

1. provjerite može li se korisnik prijaviti s valjanim korisničkim podacima
2. testirajte da se korisnik ne može prijaviti s nevažećim podacima
3. provjerite može li korisnik poništiti svoju lozinku
4. provjerite može li korisnik poništiti svoja sigurnosna pitanja
5. provjerite može li se korisnik prijaviti s ispravnim razlikovanjem velikih i malih slova
6. provjerite može li se korisnik prijaviti s posebnim znakovima u lozinki
7. provjerite može li se korisnik prijaviti s isteklom lozinkom
8. testirajte može li se korisnik prijaviti s neaktivnim računom
9. testirajte može li se korisnik prijaviti s računom koji je zaključan
10. provjerite može li se korisnik prijaviti s privremenom lozinkom.

4.1.9. BDD scenariji kao dokumentacija kod testiranja korisničkog prihvaćanja

BDD scenariji su izvrsna forma dokumentacije za **testiranje korisničkog prihvaćanja**. BDD je metodologija razvoja softvera koja stavlja naglasak na zajedničko razumijevanje ponašanja softvera između tehničkih i nefunkcionalnih timova. U BDD-u, korisničko prihvaćanje modelira se pomoću scenarija koji se fokusiraju na ponašanje sustava u odgovoru na korisničke akcije. Scenariji su napisani na prirodnom jeziku (npr. engleskom) i trebali bi biti lako razumljivi za sve članove tima, bez obzira na njihovu tehničku

stručnost. Ovi scenariji opisuju očekivano ponašanje sustava i služe kao dokumentacija za sve korake koje korisnik može poduzeti u interakciji sa sustavom.

Izgled BDD scenarija za dokumentaciju kod testiranja korisničkog prihvaćanja, koja će biti predstavljena za značajku prijavljivanja, izgledat će ovako:

Scenarij: Uspješna prijava
S obzirom da sam na stranici za prijavu
Kada unesem ispravno korisničku ime i lozinku

Scenarij: Neuspješna prijava
S obzirom da sam na stranici za prijavu
Kada unesem neispravno korisničko ime i ispravnu lozinku
Tada bi poruka o neuspješnoj prijavi trebala biti prikazana

4.1.10. Održavanje softvera popraćeno održavanjem BDD scenarija

Za održavanje softvera popraćenim održavanjem BDD scenarija, a za značajku prijavljivanja, scenarij će izgledati ovako:

1. Dodati novi scenarij prilikom prijavljivanja
   Scenarij: Korisnik je uspješno prijavljen s ispravnim podacima
   S obzirom na korisnika s valjanim podacima
   Kada se korisnik pokuša prijaviti
   Tada bi korisnik morao uspješno biti prijavljen i preusmjeren na početnu stranicu

2. Ažurirati postojeći scenarij za prijavu
   Scenarij: Korisnik se ne uspijeva prijaviti s pogrešnim podacima
   S obzirom na korisnika s nevažećim podacima
   Kada korisnik pokuša da se prijavi

Tada bi korisniku trebao biti odbijen pristup i poruka o neuspješnoj prijavi

4.1.11. Alati za održavanje testnih scenarija i praćenje statusa testa

Kako bi pravilno održavali testne scenarije i pratili statuse naših testova, koristit ćemo Jiru. Jira je softver koji koriste timovi za planiranje, praćenje i upravljanje projektima. *Jira* kao jedan od najkorištenijih softvera nudi veliki niz dodataka koji pomaže različitim dijelovima tima da što lakše obave svoj posao. Kroz integraciju Jire sa *Zephyr Scaleom* moći ćemo klonirati ili raditi nove testne planove u budućem testiranju neke nove funkcije. *Zephyr Scale* je jedan od dodataka koji se nalaze u Jiri, a koje najčešće koriste testeri za planiranje, izvođenje i izvještavanje o testovima.

Rezultat našeg testiranja može se prikazati u *Zephyr Scaleu,* gdje grafički možemo prikazati koliko smo napredak kod testiranja ostvarili, kakvi su rezultati nakon testiranja i koliko smo nedostataka prijavili tijekom testiranja.

Slika 4.1. Rezultati prikazani u obliku grafikona u Jiri

5. Upravljanje procesom testiranja softvera

U OVOM POGLAVLJU NAUČIT ĆETE:

- analizirati zahtjeve testnih aktivnosti
- izvještavati o statusu testiranja
- kako odabrati testnu strategiju i pristup kod testiranja
- što je to ciklus testiranja
- kako se planira testiranje kod životnog ciklusa razvoja softvera
- kako i zašto planirati automatizaciju testiranja.

Upravljanje testiranjem je proces u kojem se pravilno implementiranim aktivnostima testiranja osigurava kvalitetna softverska aplikacija. Sastoji se od praćenja procesa testiranja, organizacije testiranja, kontrole procesa testiranja i provjeravanja procesa testiranja kako bi se isporučila kvalitetna softverska aplikacija. Upravljanjem procesom testiranja softvera zadužen je testni tim koji ćemo objasniti u nastavku cjeline.

5.1. Testni tim

5.1.1. Vrste testnih timova

Tipični tim za testiranje ima članove koji obavljaju različite aktivnosti na temelju svojih vještina i iskustva. Te se uloge i odgovornosti mogu razlikovati ovisno o organizaciji u kojoj testni tim djeluje te veličini i složenosti projekta.

- menadžer testnog tima
- voditelj testiranja
- stariji tester
- tester funkcionalnosti
- inženjer za testiranje automatizacije
- analitičar nefunkcionalnih testova

Menadžer testnog tima preuzima cjelokupnu odgovornost za testni proces i vodstvo testnih aktivnosti. Njegovi tipični zadaci uključuju:

- upravljanje cijelim timom za testiranje za projekt ili program.
- koordiniranje s naručiteljima softverskog proizvoda, od faze pokretanja projekta kako bi se razumjeli ciljevi testa.
- sudjelovanje u sastancima analize rizika s dioničarima i predlaže kako će testiranje ublažiti rizike proizvoda.
- razvijanje testne strategije za projekt ili program.
- planiranje aktivnosti testiranja i potvrda procjene testa i plan testiranja. Odlučivanje o metrici koja se koristi za mjerenje napretka testa i zakazivanje različitih aktivnosti testiranja.

Voditelj testiranja upravlja testiranjem za projekt ili fazom testiranja. Njegovi tipični zadaci uključuju:

- procjene testa i izradu plana testiranja za projekt, dodjeljuje zadatke timu za testiranje i upravlja njihovim dnevnim aktivnostima
- pripremanje izvješće o statusu testiranja i izvješće o sažetku testiranja za dionike.

Stariji tester je iskusan tester u projektu koji dobro poznaje procese testiranja. Njegovi tipični zadaci uključuju:

- pomaganje timu u analizi zahtjeva i pripremi testnih slučajeva visoke razine
- pregledavanje testova koje su pripremili drugi testeri
- obučavanje novih testera u timu
- obavljanje neke aktivnosti upravljanja testiranjem u odsutnosti voditelja testiranja.

Funkcionalni tester odgovoran je za funkcionalno testiranje sustava. Njegovi tipični zadaci uključuju:

- analiziranje zahtjeva za stvaranje testnih slučajeva visoke razine (testni uvjeti ili testni scenariji) i testnih slučajeva niske razine (detaljni testni slučajevi)
- identificiranje i pronalaženje/pripremanje testnih podataka za testiranje
- izvršavanje testnih slučajeva i bilježenje rezultata testa tijekom izvođenja testa
- izvještavanje o nedostacima kada se kvarovi pronađu tijekom testiranja i njihovo ponovno testiranje nakon što ih programeri poprave.

Inženjer za testiranje automatizacije odgovoran je za automatizaciju ručnih testnih slučajeva. Dobro poznaje alate za automatizaciju i skriptni jezik koji se koristi za automatizaciju. Njegovi tipični zadaci uključuju:

- razumijevanje ručnih testnih slučajeva i njihovu automatizaciju pomoću alata za automatizaciju testiranja
- pomaganje timu za testiranje u odabiru pravih testnih slučajeva za automatizaciju
- analizu testnih slučajeva kako bi se vidjelo je li ih moguće automatizirati
- održavanje skripti za automatizaciju kada dođe do promjena u sustavu.

Analitičar nefunkcionalnih testova odgovoran je za testiranje performansi sustava. Njegovi tipični zadaci uključuju:

- analiziranje nefunkcionalnih zahtjeva za stvaranje testnih scenarija/skripti za performanse ili testove opterećenja

- pokretanje skripte i praćenje performansi sustava, na primjer provjera performansi sustava kada se 100 korisnika istovremeno prijavilo u sustav.

Analitičar nefunkcionalnih testova koristi dokumente o dizajnu sustava ili se savjetuje s programerima/administratorima baze podataka/arhitektima prije izrade i pokretanja testnih skripti za testiranje performansi. Alati se uglavnom koriste za izvođenje takvog testiranja, stoga je dobro poznavanje alata ključno za ovu poziciju.

5.2. Planiranje testnih aktivnosti

5.2.1. Analiza zahtjeva

Analiza zahtjeva uključuje proces identifikacije, definiranja i validacije zahtjeva za aplikaciju. Sastoji se od sljedećih koraka:

- **identifikacija** svih funkcionalnih i nefunkcionalnih zahtjeve za aplikaciju
- **definiranje zahtjeva**, uključujući funkcionalnost, performanse, sigurnost i druge kriterije
- **validacija zahtjeva**, potvrda jesu li svi zahtjevi precizno definirani i u skladu s potrebama korisnika i poslovnom strategijom
- **prioritizacija zahtjeva**, ocjena važnosti svakog zahtjeva i određivanje prioritetnosti
- **dokumentiranje zahtjeva**, uključujući njihove definicije i prioritete.
- **održavanje zahtjeva**, redovita provjera i ažuriranje zahtjeva kako se poslovne potrebe i tehnologije mijenjaju.

Analiza zahtjeva ključna je za uspješno testiranje jer testeru omogućava da se fokusira na najvažnije zahtjeve i osigura da su svi zahtjevi uključeni u testiranje. Uobičajeni dokumenti za upravljanje testiranjem su:

- Politika testiranja – opisuje ciljeve projekta i ciljeve testiranja.
- Strategija testiranja – opisuje cjelokupni pristup testiranju za projekt ili program.
- Plan testiranja – opisuje provedbu strategije testiranja za određeni projekt i definira visoku razinu planiranih aktivnosti testiranja. Za velike projekte može postojati glavni plan testiranja za cijeli projekt i više planova testiranja specifičnih za svaku razinu testiranja.

Strategija testiranja opisuje pristup testiranju projekta. Dokument strategije testiranja detaljno opisuje cjelokupni pristup testiranju i daje smjernice za testiranje. Ovaj dokument opisuje identificirani rizik proizvoda i kako će ti rizici biti ublaženi testiranjem, podjelu testiranja na razine i aktivnosti visoke razine povezane s testiranjem. Strategija testiranja te procesi i aktivnosti opisani u njemu moraju biti u skladu s politikom testiranja. Pruža generičke ulazne i izlazne kriterije testa za jedan ili više projekata ili programa. Napravljena je za informiranje svih uključenih u projekt o ciljevima testiranja, ključnim pitanjima procesa testiranja, metodama testiranja i testnim podacima te zahtjevima okoline.

Stavke koje bi trebale biti detaljno obrađene u dokumentu strategije testiranja:

- **standardi testiranja** koje treba slijediti u projektu (ovisno o domeni sustava koji se testira)
- **razine testiranja** koje će se koristiti (npr. test sustava, test integracije sustava, korisničko prihvaćanje)
- **vrste testova** koje će se koristiti (funkcionalni, nefunkcionalni testovi ili testovi koje će tester odabrati s obzirom na njegovo iskustvo)
- **tehnike projektiranja testa** koje će se koristiti (primjerice, podjela ekvivalencije, analiza graničnih vrijednosti itd.
- Pristup **ponovnom testiranju** i **regresijskom testiranju**
- **ulazni i izlazni kriteriji** koji će se koristiti za svaku razinu testiranja (primjerice, nepostojanje otvorenih kritičnih nedostataka može biti izlazni kriterij za razine testiranja sustava)
- pojedinosti o **testnim okruženjima** koja će se koristiti za testiranje

- **alati koji se koriste za podršku** aktivnostima testiranja.

Strategija testiranja razlikuje se ovisno o razvojnim modelima koji se slijede u projektu, a različite strategije testiranja prikladne su za različite organizacije i projekte. Uobičajene strategije testiranja:

Analitičke strategije (testiranje temeljeno na riziku) imaju testovi koji su dizajnirani na temelju razine rizika. (U slučaju kada rizik predstavlja prekid rada aplikacije iz nekog razloga, tada će tim morati odrediti hoće li će i kako sastaviti strategiju testiranja).

Primjerice, u testiranju temeljenom na zahtjevima, testovi visoke razine izvode se iz zahtjeva, testovi niske razine se zatim dizajniraju i implementiraju kako bi pokrili te zahtjeve. Testovi se naknadno izvršavaju, često koristeći prioritet zahtjeva koji pokriva svaki test kako bi se odredio redoslijed kojim će se testovi izvoditi. Rezultati testiranja također se objavljuju u smislu statusa zahtjeva, npr. zahtjev je testiran i prošao, zahtjev je testiran i nije prošao, itd.

Strategije temeljene na modelu, kao što je operativno profiliranje, podrazumijeva testove dizajnirane na temelju nekog modela. (npr. Model crne ili bijele kutije ovisno o tome da li poznajemo unutarnju strukturu koda same aplikacije).Model se može izgraditi na temelju određenog potrebnog aspekta proizvoda, kao što je funkcija, poslovni proces, unutarnja struktura ili nefunkcionalna karakteristika (npr. pouzdanost) ili na stvarnim ili očekivanim situacijama okruženja u kojem proizvod postoji.

Primjerice, u testiranju izvedbe na temelju modela rastuće stranice društvenih mreža, testeri mogu razviti modele za prikaz aktivnih i neaktivnih korisnika i rezultirajućeg opterećenja obrade, na temelju trenutne upotrebe i rasta projekta tijekom vremena. Osim toga, modeli se mogu razviti uzimajući u obzir hardver, softver, podatkovni kapacitet, mrežu i infrastrukturu trenutne proizvodne okoline. Modeli se također mogu razviti za idealne, očekivane i minimalne brzine protoka podataka, vremena odziva i raspodjelu resursa.

Metodičke strategije koriste testove temeljene na karakteristikama kvalitete. Ako testni tim koristi unaprijed određene skupine testnih uvjeta, primjerice utemeljenih na standardu kvalitete (npr. ISO 25000[3]), kontrolnom popisu ili zbirci logičnih testnih uvjeta koji se mogu odnositi na određenu domenu, aplikaciju ili vrstu testiranja (npr. sigurnosno testiranje), dobit će se prikaz uobičajenih ili vjerojatnih vrsta nedostataka i kvarova, kao i popis karakteristika kvalitete ili standarda softvera.

Primjerice, tijekom testiranja održavanja jednostavne *web*-stranice za *online* kupovinu, testeri mogu koristiti kontrolni popis koji identificira ključne funkcije (nova narudžba, izmjena narudžbe, brisanje narudžbi), atribute i veze za svaku stranicu.

Reaktivne strategije ne podrazumijevaju planirano testiranje, nego je ono reaktivno na komponentu ili sustav koji se testira i događaje tijekom izvođenja testa. Testovi su osmišljeni i implementirani i mogu se odmah izvršiti kao odgovor na znanje stečeno iz prethodnih rezultata testa (istraživačko testiranje). Svakom testeru dodijeljen je niz testnih povelja koje zatim koristi za strukturiranje svojih istraživačkih sesija testiranja, pri čemu se koristi svojom intuicijom kako bi rezultat testa trebao izgledati. Tako testiranje aplikacije razdvoji na sesije kako bi se pokrila čitava aplikacija ili softver koji se testira. Testeri povremeno voditelja testiranja izvještavaju o rezultatima testiranja, a on može revidirati testne povelje na temelju izvještaja.

Konzultativne strategije, poput testiranja usmjerenog prema korisniku, u kojem se testni tim oslanja na savjete, smjernice ili upute naručitelja softvera, stručnjaka za poslovnu domenu ili tehnoloških stručnjaka koji mogu biti izvan testnog tima ili izvan same organizacije. Na primjer, u vanjskom testiranju kompatibilnosti za *web*-aplikaciju, tvrtka

[3] Serija standarda ISO/IEC 25000, također poznata kao SQuaRE (System and Software Quality Requirements and Evaluation), ima za cilj stvoriti okvir za ocjenu kvalitete softverskih proizvoda.

ISO/IEC 25000 je rezultat evolucije nekoliko drugih standarda; posebno iz ISO/IEC 9126, koji definira model kvalitete za evaluaciju softverskog proizvoda, i ISO/IEC 14598, koji definira proces za evaluaciju softverskog proizvoda. Niz standarda ISO/IEC 25000 sastoji se od pet odjeljaka.

može dati vanjskom pružatelju usluge testiranja popis prioriteta različitih preglednika i njihovih verzija prema kojima želi ocijeniti svoju aplikaciju.

Strategija testiranja nesklona regresiji odnosi se na strategiju kojom se nastoji izbjeći regresija postojećih sposobnosti. Ova strategija testiranja uključuje ponovnu upotrebu postojećeg softvera za testiranje (osobito testnih slučajeva i testnih podataka), opsežnu automatizaciju regresijskih testova i standardne pakete testova. Na primjer, pri regresijskom testiranju aplikacije temeljene na *webu*, testeri mogu koristiti alat za automatizaciju testiranja temeljen na korisničkom interfejsu za automatizaciju aplikacije. Ti se testovi zatim izvršavaju u bilo kojem trenutku izmjene aplikacije. Različite strategije mogu se kombinirati kako bi se dobila odgovarajuća testna strategija.

Na primjer, testiranje temeljeno na riziku (analitička strategija) može se kombinirati s istraživačkim testiranjem (reaktivna strategija) jer se međusobno nadopunjuju i mogu postići učinkovitije testiranje kada se koriste zajedno.

Za **testiranje temeljeno na riziku**, tim testera dizajnirat će testove na temelju analize identificiranih rizika i zahtjeva. Testovima će se odrediti prioriteti na temelju razine rizika i prioriteta zahtjeva, tako da se visokorizične stavke testiraju rano u ciklusu.

Rizik proizvoda uključuje mogućnost da radni proizvod (npr. specifikacija, komponenta, sustav ili test) ne uspije zadovoljiti legitimne potrebe svojih korisnika i/ili dionika. Rizici proizvoda također se nazivaju **rizicima kvalitete** kada su povezani sa specifičnim karakteristikama kvalitete proizvoda (npr. funkcionalna prikladnost, pouzdanost, učinkovitost izvedbe, upotrebljivost, sigurnost, kompatibilnost, mogućnost održavanja i prenosivost).

Uobičajeni primjeri rizika proizvoda uključuju:

- softver možda neće obavljati predviđene funkcije u skladu sa specifikacijom
- softver možda neće obavljati predviđene funkcije u skladu s potrebama korisnika, naručitelja softvera i/ili ostalih uključenih u njegovu izradu

- arhitektura sustava možda neće adekvatno podržavati neke nefunkcionalne zahtjeve
- određeni izračun može biti netočno izveden u nekim okolnostima.

Rizik projekta podrazumijeva mogućnost situacija koje mogu imati negativan učinak na postizanje cilja tog projekta. Uobičajeni primjeri projektnih rizika uključuju:

- kašnjenja u ukupnoj isporuci ili dovršetak pojedinačnog zadatka
- netočne procjene kod razvoja aplikacije
- zakašnjele promjene mogu rezultirati značajnim utrošenim vremenom i troškovima
- projektni tim možda nema dovoljno članova ili članovi tima nisu kvalificirani za izvršavanje zadataka
- korisnici, poslovno osoblje ili stručnjaci za razvijanje softvera ili aplikacije možda neće biti dostupni zbog sukobljenih poslovnih prioriteta
- zahtjevi možda nisu dovoljno dobro definirani
- testna okolina možda neće biti spremna na vrijeme
- treća strana (vanjski partner koji je zadužen za ispostavu aplikacije ili softvera na tržište) možda neće uspjeti isporučiti potreban proizvod ili uslugu na vrijeme.

U pristupu testiranja temeljenom na riziku testiranje, priprema i aktivnosti izvršenja testa temelje se na riziku povezanom sa zahtjevima. Ovaj pristup zahtijeva procjenu rizika na temelju sljedećih kriterija:

- **poslovna kritičnost** odnosi se na potencijalni/očekivani učinak na poslovanje ako funkcija ne radi kako se očekuje
- **vjerojatnost kvara** – usredotočuje se na vjerojatnost kvara na temelju funkcionalne i tehničke složenosti.

Određivanje prioriteta rizika – na temelju kombinacije poslovne kritičnosti i postotaka vjerojatnosti neuspjeha da se neće ispuniti zahtjevi klijenta, priprema se matrica prioriteta za testiranje. Ako uzmemo za primjer sustav internetskog bankarstva, identificiramo neke od sljedećih rizika kvalitete:

- rizik 1: netočni izračuni godišnjih kamata u izvješćima (funkcionalni rizik vezan uz točnost)
- rizik 2: spor odgovor na korisnički unos (nefunkcionalni rizik povezan s učinkovitošću i vremenom odgovora).

Kod prvog primjera rizika, poslovni rizik je visok jer će pogrešan izračun godišnjih kamata utjecati na cijelo poslovanje za tu godinu. Kod drugog primjera, poslovni rizik je nizak jer korisnik još uvijek može izvršiti radnju, ali će takav rizik utjecati na veliki broj korisnika koji pristupa sustavu možda i više puta dnevno. Nakon što se napravi određivanje prioriteta na temelju rizika, tim za testiranje izradit će odgovarajuće testne slučajeve za njihovo pokrivanje. Bit će potrebno više testova za rizične stavke visokog prioriteta, a manje za rizične stavke niskog prioriteta. Testiranje temeljeno na riziku ima prednost u tome što se prvo testiraju najkritičnija područja sustava. To će smanjiti rizik čak i ako postoje bilo kakva kašnjenja ili problemi koji rezultiraju ograničenjem testiranja.

5.2.2. Planiranje testnih aktivnosti

Planiranje testa odnosi se na planiranje aktivnosti koje moraju biti izvedene tijekom testiranja kako bi se postigli ciljevi testa. Planiranje testiranja za svaku razinu testiranja projekta počinje na početku procesa testiranja te razine i nastavlja se do kraja završnih aktivnosti razine.

Proces planiranja testa uključuje sljedeće aktivnosti:

- utvrđivanje ciljeva testa
- nabrajanje testnih aktivnosti
- identificiranje potrebnih resursa
- određivanje metrike testa
- identificiranje postupaka za prikupljanje i praćenje testnih metrika
- utvrđivanje mjerila za procjenu pridržavanja plana testiranja
- identificiranje parametara za mjerenje postignuća ciljeva

- usmjeravanje na metriku testiranja softvera tijekom planiranja omogućuje testnom timu da odabere odgovarajuće alate za testiranje, osmisli rasporede obuke i postavi smjernice za dokumentaciju.

Odabir testne strategije pomaže u popisivanju ispitnih zadataka u samoj fazi planiranja. Na primjer, u slučaju testiranja temeljenog na riziku, fokus je na identificiranju aktivnosti koje mogu smanjiti rizike i izradi plana za nepredviđene situacije. Podaci analize rizika proizvoda ovdje služe kao vodič za određivanje prioriteta aktivnosti testiranja. Slično tome, ako je vjerojatno da će proizvod imati sigurnosne rizike, testiranje se usredotočuje na sigurnosno testiranje, a tijekom faze planiranja osmišljeno je i razvijeno više sigurnosnih testova.

Ako je vjerojatno da proizvod ima nedostatke u dizajnu, testiranje se usredotočuje na specifikacije dizajna, a dodatni pregledi specifikacija dizajna (statičko testiranje) planiraju se i ugrađuju u proces testiranja. Mnogo je aktivnosti testiranja koje treba provesti i zato je nužno odrediti prioritete. Podaci dobiveni analizom rizika igraju važnu ulogu u određivanju ove liste prioriteta. Na primjer, ako postoji visok rizik od problema s performansama sustava ili je izvedba sustava kritična, testiranje performansi ima najveći prioritet kada se kod dostavlja na testiranje. U slučaju reaktivnog testiranja, planiranje izrade testnih povelja i razvoj metoda dinamičkog testiranja kao što je istraživačko testiranje zadaci su najvišeg prioriteta.

5.2.3. Testni scenariji koji se dostavljaju korisnicima za testiranje korisničke prihvatljivosti

Testni scenariji isporučeni korisnicima za testiranje korisničke prihvatljivosti odnose se na skup testova osmišljenih kako bi se osiguralo da softverski proizvod zadovoljava zahtjeve korisnika prije nego što ih zaista počnu koristiti. Ove testove obično provode krajnji korisnici koji su reprezentativni za ciljanu publiku i uključuju izvršavanje softvera kako bi se procijenila njegova učinkovitost, upotrebljivost i stabilnost.

Testni scenariji obično se temelje na funkcionalnim zahtjevima proizvoda i mogu se koristiti za identifikaciju potencijalnih problema koji bi se mogli pojaviti tijekom korištenja proizvoda.

Primjer testnog scenarija koji se dostavlja korisnicima u svrhu testiranja korisničkog prihvaćanja:

- prijavite se u sustav s važećim korisničkim imenom i lozinkom
- idite na stranicu nadzorne ploče i provjerite jesu li prisutni svi očekivani elementi
- pokušajte izvršiti pretraživanje pomoću ključne riječi i provjerite rezultate
- napravite novi račun i provjerite jesu li podaci ispravno pohranjeni
- izvršite kupnju i provjerite je li plaćanje uspješno
- ažurirajte profil i provjerite jesu li promjene ispravno spremljene
- pokušajte ponovno postaviti lozinku i provjerite je li postupak uspješan
- pokušajte pristupiti ograničenoj stranici i provjerite je li pristup odbijen
- pokušajte izbrisati račun i provjerite je li uspješno uklonjen
- odjavite se iz sustava i provjerite je li sesija prekinuta.

5.2.4. Ulazni kriteriji

Kako bi se izvršila učinkovita kontrola kvalitete softvera i testiranja, preporučljivo je imati kriterije koji definiraju kada određena testna aktivnost treba započeti i kada je aktivnost dovršena. **Ulazni kriteriji** (uglavnom nazvani definicijom **spremnosti** u agilnom razvoju) definiraju preduvjete za poduzimanje određene testne aktivnosti. Ako kriteriji za ulazak nisu ispunjeni, vjerojatno će aktivnost biti teža, dugotrajnija, skuplja i riskantnija.

Tipični ulazni kriteriji uključuju:

- dostupnost zahtjeva koji se mogu testirati, korisničkih priča i/ili modela (npr. kada se slijedi strategija testiranja temeljena na modelu)

- dostupnost ispitnih stavki koje su zadovoljile izlazne kriterije za bilo koje prethodne razine testiranja
- dostupnost testnog okruženja
- dostupnost potrebnih alata za testiranje
- dostupnost testnih podataka i drugih potrebnih izvora.

5.3. Izvještavanje o statusu testiranja

Nakon što izvršimo testove, slijedi **analiza dobivenih rezultata**. Potrebno je pažljivo proučiti rezultate testa koje smo zabilježili u prethodnim procesima implementacije i izvršenja testova jer rezultati iste značajke ili funkcionalnosti u konačnici mogu davati različite rezultate (možda je poslovna logika neke značajke promijenjena). Svaki trag i rezultat potrebno je usporediti i ocijeniti s predviđenim kriterijima koje smo si zadali tijekom planiranja. Konačnim ocjenama možemo odrediti je li provedeno testiranje uspješno ili neuspješno odrađeno. Ako testiranje ocijenimo uspješnim, krećemo s kreiranjem izvještaja koji prikazuju koje rezultate smo dobili, koliko grešaka smo uočili, gdje se nalaze tj. u kojim testovima smo ih našli, koji utjecaj imaju na daljnji razvoj i slično. Također se u ovoj fazi u razmatranje uzimaju vremenski i financijski resursi te se na temelju te dvije stvari mogu definirati grafovi koji će prikazati troškove.

5.3.1. Izvještavanje o statusu testa

Izvještavanje o statusu testa važan je dio procesa testiranja. Važno je osigurati da svi uključeni u proces testiranja budu u toku s napretkom procesa testiranja i svim potencijalnim problemima ili uspjesima. Izvještavanje o statusu obično se sastoji od redovitih ažuriranja koja uključuju napredak, rezultate i sve promjene u planu testiranja. Svrha izvještavanja o statusu je osigurati da su svi uključeni u projekt informirani i da se sva pitanja mogu pravodobno riješiti. Izvještavanje o statusu treba biti prilagođeno specifičnim potrebama projekta i uključuje sljedeće elemente:

- ciljevi testa
- status izvršenja testa
- broj izvršenih testnih slučajeva
- broj utvrđenih nedostataka
- popis otvorenih nedostataka
- popis testnih slučajeva koji nisu uspjeli
- popis testnih slučajeva koji su prošli
- popis zadataka koje je potrebno izvršiti
- procjene i rokovi
- plan akcije za rješavanje svih problema
- sažetak rezultata testiranja
- sažetak napretka.

5.3.2. Izlazni kriterij

Kako smo i ranije naveli, **izlazni kriteriji** definiraju koji uvjeti moraju biti zadovoljeni da bi se proglasila testna razina ili kada je set testova završen. **Izlazni kriteriji** definiraju koji uvjeti moraju biti postignuti da bi se proglasila testna razina ili kada je set testova završen. Ulazni i izlazni kriteriji trebaju biti definirani za svaku razinu i vrstu testiranja, te se razlikuju ovisno o ciljevima testa.

Tipični **izlazni kriteriji** uključuju:

- planirana testiranja su izvršena
- definirana razina pokrivenosti (npr. zahtjeva, korisničkih priča, kriterija prihvaćanja, rizika, koda) je postignuta
- broj neriješenih nedostataka je unutar dogovorenog ograničenja
- broj procijenjenih preostalih nedostataka je dovoljno nizak
- procijenjene razine pouzdanosti, učinkovitosti izvedbe, upotrebljivosti, sigurnosti i drugog relevantnog karakteristike kvalitete su dovoljne.

Čak i bez zadovoljenja izlaznih kriterija, također je uobičajeno da se testne aktivnosti ograniče zbog financijskih ili vremenskih resursa i/ili pritiska da se proizvod plasira na tržište.

Nakon što se proizvedu razni testni slučajevi i testni postupci (s nekim potencijalnim testnim postupcima automatizacije) koji su sastavljeni u testne pakete, testni paketi raspoređuju se u raspored izvođenja testa koji definira redoslijed njihova izvođenja. Kod rasporeda izvođenja testa treba uzeti u obzir čimbenike poput određivanje prioriteta, ovisnosti, potvrdnih testova, regresijskih testova i najučinkovitijeg slijeda za izvođenje testova.

U idealnom slučaju, testni slučajevi trebali bi se pokrenuti na temelju njihovih razina prioriteta, obično izvršavanjem prvo testnih slučajeva s najvišim prioritetom. Međutim, ova praksa možda neće funkcionirati ako testni slučajevi ili značajke koje se testiraju imaju ovisnosti. Drugim riječima, ako je testni slučaj s višim prioritetom ovisan o testnom slučaju nižeg prioriteta, prvi se mora izvršiti testni slučaj nižeg prioriteta. Slično tome, ako postoje ovisnosti između testnih slučajeva, one se moraju pravilno poredati bez obzira na njihove relativne prioritete.

5.4. Analize i procjene

Analize i procjene su procesi koji se koriste u testiranju softvera kako bi se ocijenila kvaliteta i ispravnost aplikacije. Analiza uključuje testiranje specifikacija i zahtjeva za aplikaciju kako bi se identificirali mogući problemi i slabosti. Cilj analize je identificirati i definirati probleme u ranoj fazi razvoja, kako bi se spriječile buduće poteškoće.

Procjena kod testiranja se koristi za određivanje vjerojatnosti i stupnja utjecaja potencijalnih problema. To se obično obavlja nakon analize, a cilj je odrediti prioritet problema i planirati kako ih riješiti. Analize i procjene su važni koraci u procesu testiranja kako bi se osiguralo da aplikacija funkcionira ispravno i zadovoljava zahtjeve korisnika.

5.4.1. Trošak i isplativost

Trošak testiranja je cijena resursa i sredstava koja se troše za provođenje testiranja softvera. Ovi resursi uključuju vrijeme i napor testera, hardver i opremu za testiranje, softver za testiranje i druge materijalne troškove. Isplativost testiranja odnosi se na mjere koje se koriste za određivanje koristi i vrijednosti testiranja u odnosu na njegove troškove. Ova se mjera obično izražava kroz financijske kriterije, poput povećanja produktivnosti, smanjenja troškova povezanih s greškama i problemima, povećanja zadovoljstva korisnika i slično.

Cilj svakog razvojnog tima je da maksimizira isplativost testiranja, što znači da treba postići najveću moguću korist od testiranja u odnosu na njegove troškove. Da bi se postigla optimalna isplativost, potrebno je uravnotežiti troškove i koristi testiranja, koristiti najbolje prakse i alate za testiranje i osigurati da su testovi dizajnirani tako da se softver adekvatno i efikasno testira. Ukratko, trošak testiranja i isplativost su važni aspekti u procesu testiranja softvera, a razvojni timovi trebaju ravnopravno razmatrati oba faktora kako bi postigle optimalnu isplativost testiranja.

Ulaganje u testiranje povećava zadovoljstvo korisnika. Konačni cilj mnogih razvojnih timova je da njihov softver što je moguće brže stigne do što većeg broja korisnika, što zapravo nije održiv poslovni model jer brzina često dovodi do pogrešaka i lošeg korisničkog iskustva. Plasiranje softvera s greškama ili softvera koji pruža loše korisničko iskustvo na tržište vjerojatno će navesti krajnje korisnike da potraže proizvode sličnih značajki kod konkurencije.

Ulaganje u testiranje ubrzava inovacije. Kako bi ostali konkurentni u današnjem digitalnom svijetu, razvojni timovi moraju kontinuirano razvijati nove tehnologije kako bi privukli veliki broj korisnika, a samim time i povećali svoje prihode.

Ulaganje u testiranje povećava produktivnost. Uključivanje testera i provjere softvera u razvojni proces povećava produktivnost cijelog tima. Naravno, potrebno je manje vremena za brzo izbacivanje softvera nego za njegovo temeljito testiranje prije slanja, međutim vrijeme koje se kao takvo uštedi, kompenzira se mnogim satima kada tim mora popravljati te greške. Takav problem utječe na samo plasiranje aplikacije na tržište jer je potrebno puno više vremena za ispravljanje grešaka u fazi proizvodnje nego u fazi dizajna. Kada se uključe testiranje softvera i provjere kvalitete kao dio tijeka rada, tim uštedi više vremena, povećavajući produktivnost i štedeći novac.

Ulaganje u testiranje izbjegava skupe prerade. Važno je da testeri testiraju proizvod tijekom cijelog razvojnog ciklusa jer što kasnije u procesu pronađu nedostatak, to će više koštati njegovo popravljanje. Puno je jednostavnije unošenje promjena u aplikaciju u ranim fazama životnog ciklusa razvoja nego u završnim fazama.

Ulaganje u testiranje sprječava katastrofe. Ograničavanje testiranja kako bi se proizvodi brže plasirali na tržište predstavlja značajan rizik za tvrtke jer bi softver mogao raditi loše, pokvariti se, ugroziti sigurnost — ili možda čak koštati života.

Ulaganje u testiranje poboljšava ugled i prepoznatljivost robne marke. Danas je, više nego ikad, *online* prisutnost nekog proizvoda ili tvrtke sinonim za robnu marku. Dakle, plasiranje softvera s greškama negativno utječe na kratkoročni prihod, a potencijalno može izazvati i ogroman gubitak povjerenja u neki proizvod. Izlazak na tržište sa softverom bez grešaka, s druge strane, povećat će svijest o proizvodu i poboljšati ugled.

5.4.2. Procjena napora

Procjena napora testiranja uključuje predviđanje količine rada povezanog s testiranjem koji će biti potreban da bi se zadovoljili ciljevi testiranja za određeni projekt, **izdanje** ili **iteraciju**. Čimbenici koji utječu na testne napore uključuju karakteristike proizvoda,

karakteristike procesa razvoja, karakteristike ljudi i rezultate testa, kao što je prikazano u nastavku.

Karakteristike proizvoda:

- rizici povezani s proizvodom - možemo li isporučiti proizvod prema zahtjevima krajnjih korisnika
- kvaliteta testne baze - testna baza mora biti kvalitetna i sastojati se od kvalitetno izrađenih testova kako bi isporučeni proizvod zadovoljio očekivanja naručitelja softvera, ali i krajnjih korisnika
- veličina proizvoda - veličina proizvoda može utjecati na sami uređaj korisnika (ako korisnik posjeduje stariji uređaj, upotreba softvera ili aplikacije za njega može biti otežana i neučinkovita, što može dovesti do gubitaka korisnika)
- složenost domene proizvoda - domena proizvoda koristi certifikate koja dopušta korisnicima pristup proizvodu
- zahtjevi za karakteristike kvalitete (npr. sigurnost, pouzdanost) - svakom korisniku je bitna sigurnost ili pouzdanost (posebice kod, na primjer, bankovnih aplikacija), stoga je važno osigurati kvalitetu proizvoda
- potrebna razina detalja za dokumentaciju o testiranju - svi rezultati testiranja moraju biti navedeni u dokumentaciji za testiranje koja dokazuje da je isporučeni proizvod spreman za upotrebu
- zahtjevi za zakonsku i regulatornu usklađenost - vrlo je bitno voditi se zakonima i regulativama za pojedine sfere u koje aplikacija ili softver spadaju (medicina, bankarstvo…).

Karakteristike procesa razvoja:

- stabilnost i zrelost razvojnog tima - kako bi proizvod bio isporučen krajnjim korisnicima proizvoda, stabilnost razvojnog tima koji ima iskustva u razvoju i isporuci proizvoda jedna je od glavnih stavki kod procesa razvoja
- razvojni model u uporabi - odabrati pravi razvojni pristup za određeni proizvod dovodi do lakšeg, bržeg i pouzdanijeg procesa razvoja

- testni pristup i proces - pokazuje da je kod procesa razvoja bio zahvaćen najveći mogući postotak testa (da je aplikacija testirana u cijelosti i sigurna za korištenje)
- korišteni alati - kod procesa razvoja važno je odabrati alate koji će se koristiti unutar tima, što može dovesti do lakšeg, bržeg i pouzdanijeg procesa razvoja
- vremenski pritisak - pritisak da se proizvod isporuči u određenom vremenu može dovesti do negativnog utjecaja na sami proces razvoja, stoga je bitno postaviti prihvatljive i dostižne rokove.

Karakteristike ljudi:

- vještine i iskustvo uključenih ljudi, posebno sa sličnim projektima i proizvodima (npr. znanje domene)
- Timska kohezija i vodstvo.

Rezultati testiranja:

- broj i ozbiljnost pronađenih nedostataka
- potrebna dorada funkcionalnosti koja nije prošla testiranje.

5.4.3. Odabir testne strategije i pristupa

Odabir testne strategije i pristupa ključan je korak u procesu testiranja softvera. To podrazumijeva da je potrebno odrediti koje testove treba provesti te kada i kako. Ova odluka utječe na efikasnost i isplativost testiranja, kao i na kvalitetu i sigurnost softvera. Odabir testne strategije i pristupa zavisi od mnogih faktora, uključujući zahtjeve korisnika, vrstu i složenost softvera, vrijeme i budžet za testiranje, kao i specifične ciljeve i očekivanja naručitelja softvera. Kombinacija više testnih strategija i pristupa može se koristiti kako bi se osiguralo da softver bude što je moguće pažljivije testiran i da se svi važni aspekti testiraju. Bitno je ravnopravno razmatrati sve faktore i pažljivo odabrati testne strategije i pristupe kako bi se postigli ciljevi testiranja i osiguralo da softver zadovoljava zahtjeve korisnika.

Dvije najčešće korištene tehnike pristupa testiranju su:

1. **Tehnika temeljena na metrici**

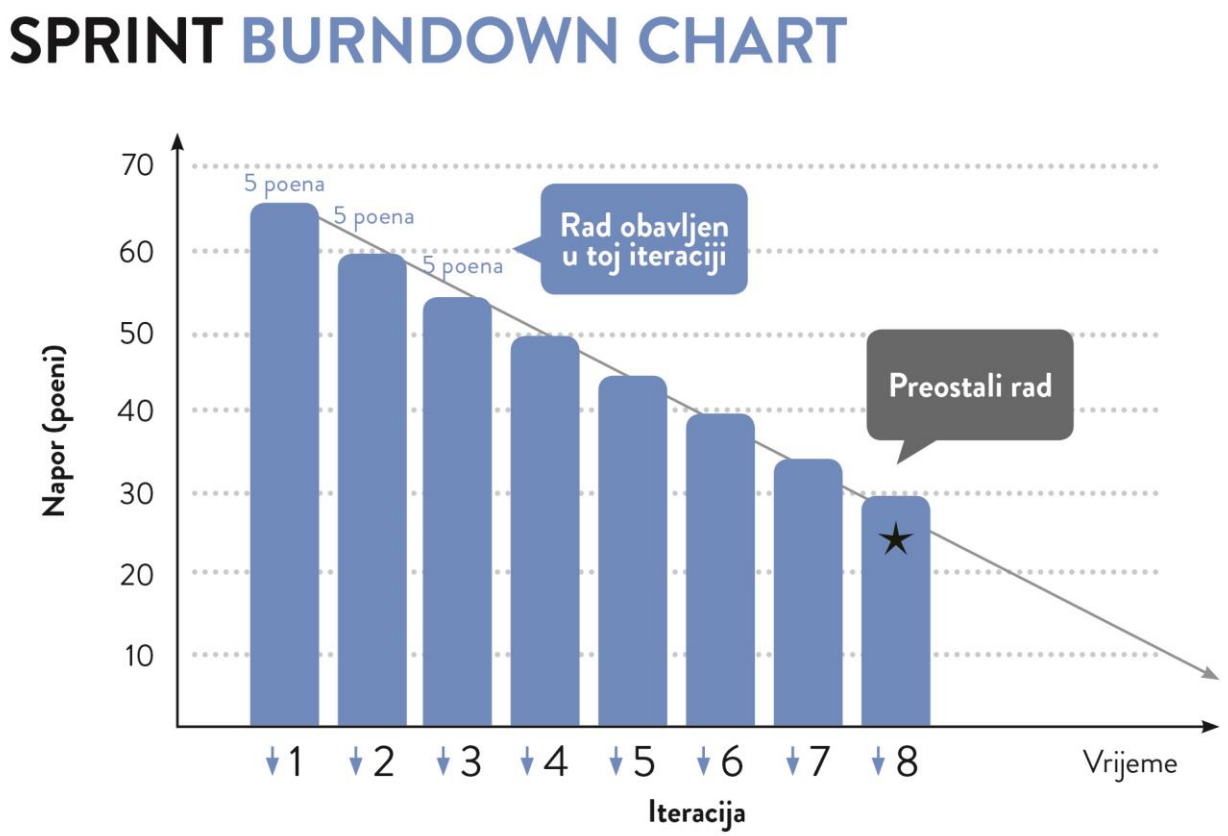
Tehnika temeljena na metrici pristup je testiranju koji se temelji na korištenju mjera (metrika) za procjenu i poboljšanje kvalitete softvera. Ova tehnika koristi različite mjere, poput broja grešaka, vremena za izvršenje, broja funkcionalnih jedinica itd., za procjenu i usporedbu kvalitete softvera s ciljem identifikacije problema i njegovog rješavanja. Cilj tehnike temeljene na metrici je objektivno procjenjivanje kvalitete softvera i identifikacije problema koji utječu na kvalitetu softvera. Ovaj pristup također pomaže u planiranju i kontroli procesa testiranja, a također može pomoći u identifikaciji područja kojoj su potrebna poboljšanja. Međutim, korištenje tehnike temeljene na metrici zahtijeva precizne i pouzdane metrike, pa je važno odabrati one odgovarajuće koje će biti korisne u procjeni kvalitete softvera. Također, potrebno je pažljivo razmatrati i tumačiti rezultate korištenjem ovih metrika, kako bi se osiguralo dobivanje pouzdanih i relevantnih podataka.

2. **Tehnika utemeljena na stručnjacima:**

Tehnika utemeljena na stručnjacima pristup je testiranju softvera u kojem se koriste iskustvo i znanje stručnjaka za testiranje, odnosno da se identificiraju i isprave problemi u softveru. Ova se tehnika često koristi za testiranje kompleksnih i složenih softverskih sustava jer stručnjaci za testiranje imaju značajno iskustvo u identificiranju i rješavanju problema u softveru. U ovom pristupu stručnjak za testiranje koristi svoje iskustvo i znanje kako bi identificirao potencijalne probleme u softveru, a zatim izvodi testove kako bi potvrdio ili odbacio te probleme. Stručnjak također može koristiti razne tehnike, poput analize i modeliranja, kako bi razvio i proveo testove. Prednost ovog pristupa je u tome što se koristi iskustvo i znanje stručnjaka, što često rezultira u identificiranju i ispravljanju problema u softveru brže i efikasnije nego što bi to bilo moguće s automatiziranim testovima.

Na primjer, agilnom modelu, *burndown* grafikon primjer je pristupa koji se temelji na metrikama. Dolje prikazan grafikon ima prikazane dvije metrike. Na X osi se nalazi broj iteracije (sprinta), a Y osi čitamo količinu preostalog rada za gledanu iteraciju, a na Y osi

prikazujemo vrijeme određeno za razvoj i određivanje količine posla koju razvojni tim može obaviti u sljedećoj iteraciji.



Slika 5.1. *Burdown* grafikon u Agilnom razvoju

Unutar sekvencijalnih projekata, modeli uklanjanja nedostataka primjeri su pristupa temeljenog na metrici, gdje se bilježe broj nedostataka i vrijeme potrebno za njihovo uklanjanje, što onda predstavlja osnovu za procjenu sličnih budućih projekata.

5.4.4. Analiza rizika

Rizik uključuje mogućnost realiziranja nekog događaja u budućnosti koji će imati negativne posljedice. Razina rizika određena je vjerojatnošću realiziranja tog događaja i njegovim utjecajem (štetom).

Ranije smo spomenuli da **rizik proizvoda** uključuje mogućnost da radni proizvod (npr. specifikacija, komponenta, sustav ili test) možda neće uspjeti zadovoljiti potrebe korisnika i/ili vlasnika proizvoda. Kada su rizici proizvoda povezani s određenim karakteristikama kvalitete proizvoda (npr. funkcionalna prikladnost, pouzdanost, učinkovitost izvedbe, upotrebljivost, sigurnost, kompatibilnost, mogućnost održavanja i prenosivost), rizici proizvoda tada se nazivaju **rizicima kvalitete**.

Primjeri rizika proizvoda uključuju:

- softver možda neće obavljati predviđene funkcije prema specifikaciji
- softver možda neće obavljati predviđene funkcije prema krajnjem korisniku
- arhitektura sustava možda neće adekvatno podržavati neke nefunkcionalne zahtjeve
- određeni izračun može biti netočno izveden u nekim okolnostima
- struktura kontrole petlje može biti neispravno kodirana
- vremena odgovora mogu biti neadekvatna za sustav obrade transakcija visokih performansi
- povratne informacije o korisničkom iskustvu (UX) možda neće ispuniti očekivanja proizvoda.

Projektni rizik uključuje situacije koje, ako se dogode, mogu imati negativan učinak na ostvarenje ciljeva projekta.

Rizici na projektu mogu biti :

1. projektni:
   - kašnjenje u isporuci, dovršetku zadatka ili nezadovoljavanju izlaznih kriterija
   - netočne procjene, preraspodjela sredstava na projekte višeg prioriteta ili opće smanjenje troškova u organizaciji mogu rezultirati neadekvatnim financiranjem

2. organizacijski:
   - tim nema znanja potrebna kod razvoja i isporuke

- testeri ne priopćuje rezultate testa na odgovarajući način
- neprikladan stav prema testiranju ili očekivanja od njega.

3. tehnički:
   - zahtjevi nisu dovoljno dobro definirani
   - zahtjevi nisu ispunjeni, s obzirom na postojeća ograničenja
   - testna okolina nije spremna na vrijeme
   - pretvorba podataka, planiranje migracije i njihova podrška za alate mogu kasniti.

Projektni rizici mogu utjecati i na razvojne i na testne aktivnosti. U nekim slučajevima voditelji projekata odgovorni su za rukovanje svim projektnim rizicima, ali nije neobično da voditelji testiranja imaju odgovornost za rizike projekta povezanih s testiranjem.

5.5. Planiranje automatizacije i održavanje testova

Automatsko testiranje znači korištenje alata za automatizaciju za izvođenje skupa testnih slučajeva. Softver za automatizaciju također može unijeti testne podatke u **System Under Test**[4], usporediti očekivane i stvarne rezultate i generirati detaljna testna izvješća. Uzastopni razvojni ciklusi zahtijevat će ponavljanje izvođenja istog skupa testova. Pomoću alata za automatizaciju testiranja moguće je snimiti ovaj testni paket i ponovno ga reproducirati prema potrebi. Nakon što se testni paket automatizira, nije potrebna ljudska intervencija. Ovo je poboljšalo ROI[5] automatizacije testiranja. Cilj automatizacije je smanjiti broj testnih slučajeva koji se ručno izvode, a ne eliminirati ručno testiranje.

[4] Sustav pod testiranjem (SUT) odnosi se na sustav koji se testira na ispravan rad. Prema ISTQB-u to je ispitni objekt.Iz perspektive testiranja jedinice, SUT predstavlja sve klase u testu koje nisu unaprijed definirani dijelovi koda poput završnica ili čak ismijavanja. Svaki od njih može imati vlastitu konfiguraciju (ime i verziju), što ga čini skalabilnim za niz testova kako bi postali sve precizniji, u skladu s količinom kvalitete sustava u testu.

[5] Povrat ulaganja (ROI) mjera je izvedbe koja se koristi za procjenu učinkovitosti ili profitabilnosti ulaganja ili usporedbu učinkovitosti više različitih ulaganja. ROI pokušava izravno izmjeriti iznos povrata na određeno ulaganje, u odnosu na trošak ulaganja.

Automatizirano testiranje softvera važno je iz sljedećih razloga:

- ručno testiranje svih tijekova rada, svih polja, svih negativnih scenarija oduzima vrijeme i novac.
- teško je ručno testirati višejezične stranice
- automatizacija ne zahtijeva ljudsku intervenciju
- automatizirani test može biti pokrenut bez nadzora (preko noći)
- automatizacija povećava brzinu izvođenja testa
- automatizacija pomaže u povećanju pokrivenosti testom
- ručno testiranje može postati zamorno i stoga podložno pogreškama.

5.5.1. Ciklus testiranja i planiranje testnog ciklusa u životnom ciklusu razvoja

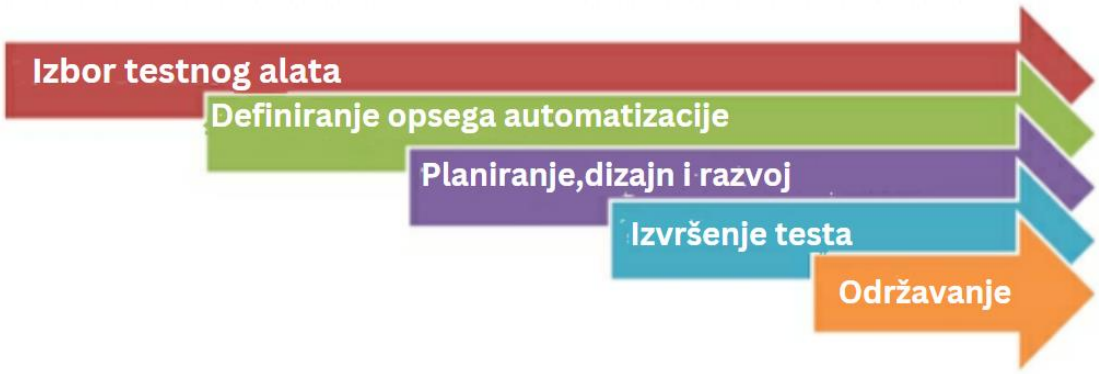
Testni slučajevi koji će se automatizirati mogu se odabrati prema sljedećem kriteriju za povećanje automatizacije ROI-a:

- visoki rizik - poslovni kritični testni slučajevi
- testni slučajevi koji se opetovano izvode
- testni slučajevi koji su vrlo zamorni ili ih je teško izvesti ručno.

Testni slučajevi koji oduzimaju puno vremena su kategorija testnih slučajeva koja nije prikladna za automatizaciju:

- testni slučajevi koji su novo dizajnirani i nisu ručno izvedeni barem jednom
- testni slučajevi za koje se zahtjevi često mijenjaju
- testni slučajevi koji se izvršavaju na *ad-hoc* bazi.

Na prikazanoj slici su nazivi koraka koji se koriste u procesu automatizacijskog testiranja.



Slika 5.2. Procesi automatizacije

1. **Izbor testnog alata** (engl. *Test Tool Selection*) - odabir alata za testiranje uvelike ovisi o tehnologiji na kojoj je izgrađena aplikacija koja se testira.
2. **Definiranje opsega automatizacije** - opseg automatizacije je područje aplikacije pod testom koje će biti automatizirano. Sljedeće točke pomažu u određivanju opsega:
    - značajke koje su važne za poslovanje
    - scenariji koji imaju veliku količinu podataka
    - zajedničke funkcionalnosti u svim aplikacijama
    - tehnička izvedivost
    - razmjer do kojeg se poslovne komponente ponovno koriste
    - složenost testnih slučajeva
    - mogućnost korištenja istih testnih slučajeva za unakrsno testiranje.
3. **Planiranje, dizajn i razvoj** - tijekom ove faze stvara se strategija i plan automatizacije koji sadrži sljedeće pojedinosti:
    - ddabrani alati automatizacije
    - dizajn okvira i njegove značajke
    - stavke unutar i izvan opsega automatizacije
    - priprema testne platforme za automatizaciju
    - raspored i vremenski okvir skriptiranja i izvođenja
    - isporučivi rezultati testiranja automatizacije.

4. **Izvršavanje testa** - skripte za automatizaciju izvođenja testa izvršavaju se tijekom ove faze. Skriptama su potrebni ulazni testni podaci prije nego što se postave za izvođenje, a nakon izvršenja daju detaljna izvješća o testiranju. Izvršenje se može izvesti pomoću alata za automatizaciju izravno ili putem alata za upravljanje testiranjem koji će preusmjeriti na alat za automatizaciju.
5. **Održavanje** - dodavanjem novih funkcionalnosti koje se uzastopno testiraju kroz nove cikluse potrebno je ažuriranje već napisanih automatiziranih skripti kako bi se poboljšala njihova učinkovitost.

Kod ciklusa testiranja vrlo je bitan okvir za automatizaciju. **Okvir za automatizaciju testiranja softvera predstavlja skup alata, procesa i tehnika koje se koriste za automatiziranje testiranja softvera.** Cilj ovog okvira je povećati efikasnost i efektivnost testiranja, smanjiti troškove i ubrzati vrijeme testiranja. Korištenjem automatizacijskog okvira, testni timovi mogu automatizirati veliki broj testova, smanjiti vrijeme i troškove testiranja te osigurati visoku kvalitetu i pouzdanost softvera. Međutim, važno je napomenuti da automatizacijski okvir zahtijeva visoku razinu planiranja i investiranja u resurse i alate kako bi se postigli željeni rezultati.

Postoje **četiri vrste okvira** koji se koriste u testiranju softvera za automatizaciju:

1. **okvir za automatizaciju vođen podacima** (engl*. Data Driven Automation Framework*)
2. **okvir za automatizaciju vođen ključnim riječima** (engl*. Keyword Driven Automation Framework*)
3. **modularni okvir za automatizaciju** (engl*. Modular Automation Framework*)
4. **hibridni automatizacijski okvir** (engl*. Hybrid Automation Framework*).

Kako bi efikasnost automatizacije bila maksimalna, potrebno je pridržavati se dobrih praksi koje su prihvaćene u svijetu testiranja:

- opseg automatizacije potrebno je detaljno odrediti prije početka projekta
- odabrati pravi alat za automatizaciju: alat se ne smije odabrati na temelju njegove popularnosti, već mora odgovarati zahtjevima automatizacije

- odabrati odgovarajući okvir
- mjerenje metrike: uspjeh automatizacije ne može se odrediti usporedbom ručnog rada s naporom automatizacije, već također bilježenjem sljedećih metrika:
    - postotak pronađenih nedostataka
    - vrijeme potrebno za testiranje automatizacije za svaki pojedini ciklus izdavanja
    - minimalno vrijeme potrebno za izdavanje
    - indeks zadovoljstva krajnjih korisnika
    - poboljšanje produktivnosti
- standardi skriptiranja: standardi se moraju poštovati tijekom pisanja skripti za automatizaciju:
    - stvaranje jedinstvene skripte i komentare
    - adekvatno rukovanje iznimkama: kako se postupa s greškom u slučaju kvara sustava ili neočekivanog ponašanja aplikacije.
    - korisnički definirane poruke trebaju biti kodirane ili standardizirane za evidentiranje pogrešaka kako bi ih testeri razumjeli.

Prednosti automatizacijskog testiranja:

- 70% brže od ručnog testiranja
- šira pokrivenost značajki aplikacije testom
- pouzdani rezultati
- osiguranje dosljednosti
- štedi vrijeme i troškove
- poboljšava točnost
- ljudska intervencija nije potrebna tijekom izvođenja
- povećava učinkovitost
- bolja brzina u izvođenju testova
- višekratno upotrebljive testne skripte
- testiranje može biti često i temeljito
- više ciklusa izvođenja može se postići automatizacijom
- rani izlazak na tržište.

Različite vrste testiranja softvera koje je moguće automatizirati su:

- *smoke test*
- testiranje jedinica
- integracijsko testiranje
- funkcionalno testiranje
- testiranje ključnih riječi
- regresijsko testiranje
- testiranje temeljeno na podacima
- testiranje crne kutije.

Odabir pravog alata može biti težak zadatak. Veliki broj kriterija utječe na njegov odabir. U obzir moramo uzeti okruženje, jednostavnost upotrebe, bazu podataka, korištenje skriptnih jezika itd. Odabir alata jedan je od najvećih izazova s kojim se treba pozabaviti prije automatizacije. Prvo je potrebno identificirati zahtjeve, istražiti razne alate i njihove mogućnosti i postaviti očekivanja od alata. Pravi odabir alata za automatizaciju, postupak testiranja i tim važni su za uspjeh automatizacije. Ručne i automatizirane metode idu ruku pod ruku za uspješno testiranje.

Pitanja za ponavljanje:

1. Tko sve spada u testni tim?
2. Kako pravilno analizirati zahtjeve?
3. Što spada u procjenu napora?
4. Što je izlazni kriterij?
5. Koje su četiri vrste *frameworka* koje se koriste u testiranju softvera za automatizaciju?

6. Izvještavanje o nedostacima softvera

U OVOM POGLAVLJU NAUČIT ĆETE:

- što su to izvještaji o nedostacima i koje elemente sadržavaju
- kako kreirati anotirani snimak zaslona
- kako odrediti stupanj težine nedostatka
- kako se dodjeljuje statusi kod životnog ciklusa defekta od objavljivanja do zatvaranja
- što je to Jira.

6.1. Izvještaji o nedostacima

Izvješće o nedostacima je dokument koji sadrži pojedinosti o tome koji su nedostaci identificirani, koji se koraci poduzimaju da se nedostaci pojave (odnosno da se reproduciraju kako bi se utvrdilo zašto se pojavljuju) i koji su očekivani rezultati da aplikacija prikazuje pogrešku (defekt) dok poduzima određene radnje, korak po korak. Izvješća o nedostacima obično izrađuje tim za osiguranje kvalitete, a također i krajnji korisnici. Korisnici često otkriju više nedostataka i prijave ih timu za podršku razvoja softvera.

Izvješća o nedostacima kreiraju se kako bi se pomoglo programerima da otkriju nedostatke i poprave ih. Tim testera obično dodjeljuje izvješće o nedostacima programeru koji zatim čita izvješće i reproducira nedostatke na softverskom proizvodu, slijedeći radnje navedene u izvješću. Nakon toga programer popravlja nedostatke kako bi se dobio željeni ishod naveden u izvješću. Zato su izvješća o nedostacima važna i pažljivo izrađena.

6.1.1. Format i elementi izvješća o nedostacima

Izvješća o nedostacima trebaju biti kratka, organizirana, izravna i trebaju pokrivati sve informacije koje programer treba otkriti. Uobičajeno je da timovi testera od klijenata dobiju izvješća o nedostacima koja su ili prekratka da bi se reproducirala i ispravila ili preduga da bi se shvatilo što je zapravo pošlo po zlu.

Primjer: Opis kvara: Aplikacija ne radi prema očekivanjima.

Razvojni programer ili tester vidi da se aplikacija ponaša drugačije od očekivanog, ali u takvom formatu nije ništa drugo opisano i reprodukcija takvog nedostatka možda neće biti moguća. U takvom slučaju, programer obavještava testera da nije mogao pronaći nikakav problem ili da je možda popravio bilo koju drugu pogrešku, ali ne i nedostatak koji je otkrio korisnik. Zato je jako važno izraditi sažeto izvješće o kvarovima kako bi se oni ispravili.

Tipično izvješće o kvaru sadrži informacije u Excel tablici i sadrži:

1. **ID kvara**: serijski broj nedostatka u izvješću.
2. **opis kvara:** Kratak i jasan opis otkrivenog kvara.
3. **koraci akcije:** Radnje koje se poduzimaju korak po korak
4. **očekivani rezultat:** Koji se rezultati očekuju prema zahtjevima prilikom izvođenja navedenih radnji.

5. **stvarni rezultat:** Koji se rezultati zapravo prikazuju prilikom izvođenja navedenih radnji.
6. **ozbiljnost** (engl. *Severity*):

- **trivijalan** - mali *bug* koji ne utječe na korištenje softverskog proizvoda
- **nisko** - mali *bug* koji treba popraviti i opet neće utjecati na performanse softvera
- **srednji** - greška koja utječe na performanse, primjerice prepreka za poduzimanje određene radnje. Ipak, postoji drugi način da učinite istu stvar.
- **visoko** - jako utječe na softver, iako postoji način da softverski proizvod uspješno učini ono što *bug* narušava.
- **kritično** - pogreške koje uvelike utječu na performanse aplikacije, poput rušenja sustava, zamrzavanja sustava ili zahtijevanja da se sustav ponovno pokrene kako bi ispravno radio.

7. **prilozi:** Niz snimaka zaslona izvođenja radnji korak po korak i dobivanja neočekivanog rezultata. Također se može priložiti kratka snimka zaslona izvođenja koraka i nailaženja na nedostatke. Kratki videozapisi pomažu programerima da lako i brzo razumiju greške.
8. **dodatne informacije**: Platforma koju ste koristili, operacijski sustav i verzija, kao i druge informacije koje detaljno opisuju nedostatke za pomoć programeru u razumijevanju problema i popravljanju koda za postizanje željenih rezultata.

6.1.2. Anotirani snimak zaslona

Snimak zaslona s komentarima u testiranju softvera je snimak zaslona računalne aplikacije ili *web*-stranice koja je označena komentarima i/ili strelicama za označavanje različitih elemenata softvera. Koristi se za ilustraciju problema sa softverom ili za pružanje objašnjenja značajke ili ponašanja. Snimka zaslona s komentarima također se može koristiti za pružanje vizualne dokumentacije softvera drugim korisnicima ili programerima.

6.1.3. *Logovi* (defekti u Gherkinu)

Gherkin je moguće koristiti kao pomoć pri pisanju kratkih, ali informacijama bogatih izvješća. Umjesto nabrajanja koraka za reprodukciju, dodaju se ključne riječi, kao u ovom primjeru:

Given I logged in with my business account
When I click on the "foo" tab in the left-hand navigation
And I select "bar" from the "quux" drop-down list
Then the main document view should show the "quux" data
And the main document view should not show an endless loading animation

- ***Given*** se koristi za popis preduvjeta
- ***When*** za popis svih radnji koje poduzmete
- ***Then*** za popis svojih očekivanja i ishoda
- ***And*** kad god se preduvjet, radnja ili očekivanje/ishod sastoji od više dijelova.

6.1.4. Određivanje stupnja težine

Vrste prioriteta greške/nedostatka mogu se kategorizirati u tri dijela:

- **niska:** kvar je iritantan, ali popravak se može obaviti tek nakon što se ozbiljniji kvar popravi
- **srednji:** tijekom normalnog tijeka razvojnih aktivnosti nedostatak bi trebao biti riješen, može pričekati dok se ne izradi nova verzija.
- **visoko:** kvar se mora riješiti što je prije moguće jer ozbiljno utječe na sustav i ne može se koristiti dok se ne popravi.

6.1.5. Težina nedostatka

Težina nedostataka je mjera razine utjecaja koji nedostatak ima na razvoj ili rad softvera. Razine nedostatka obično se klasificiraju kao **kritične**, **veće**, **manje** i **trivijalne**.

Kritični nedostatak je nedostatak koji dovodi do potpunog kvara softverskog sustava. Ne postoje zaobilazna rješenja. Kritične pogreške treba riješiti što je prije moguće jer bez popravka krajnji korisnici neće moći koristiti aplikaciju. Na primjer, na *web*-stranici za kupnju sljedeće pogreške bit će klasificirane kao kritične:

- korisnik se ne može prijaviti na svoj račun
- korisnik ne može dovršiti plaćanje
- cijena proizvoda nije prikazana.

Veći nedostatak je onaj koji dovodi do kvara ključnog dijela aplikacije. Zaobilazna rješenja ili alternative mogu postojati, ali nisu idealne. Na primjer, na *web*-stranici za kupnju sljedeće pogreške bit će kategorizirane kao velike:

- rezultati pretraživanja ne odgovaraju upitu pretraživanja
- ne možete koristiti debitne kartice tijekom naplate. (Iako se mogu koristiti kreditne kartice i druge opcije plaćanja)
- recenzije proizvoda nisu prikazane.

Manji nedostatak je nedostatak koji uzrokuje probleme u nekoj nevažnoj ili nižoj funkcionalnosti sustava. Na primjer, na *web*-stranici za kupnju sljedeće greške će se smatrati manjim:

- ne mogu se pretraživati prošle narudžbe starije od godinu dana
- ne možete uspoređivati više od tri proizvoda odjednom
- fotografija proizvoda koje su učitali korisnici nisu jasne.

Trivijalni nedostatak obično nastaje u estetskom dijelu aplikacije poput neporavnatih korisničkih elemenata, teksta koji se preklapa i poveznica koje ne rade.

6.1.6. Uloga testera u određivanju prioriteta ispravljanja i zagovaranje grešaka (engl. *Bug Advocacy*)

Zagovaranje grešaka je proces koji podrazumijeva zagovaranje ispravljanja specifične greške ili problema u softveru. Ova tehnika se koristi kada testni tim želi da ispravljanje određene greške dobije visoki prioritet u razvojnom ciklusu softvera.

U takvom pristupu predstavljamo pet pitanja - **Što? Gdje? Tko? Zašto? Kada? Kako?**

Što?

Pitanjem "što" podrazumijeva se problem sa specifičnom greškom ili problemom u softveru. Ova tehnika se koristi kada testni tim želi saznati više o specifičnom problemu i kako se on može ispraviti. Glavni cilj ovog pristupa je osigurati da se razumiju uzroci problema i da se zagovara njegovo ispravljanje na najbolji mogući način.

Proces zagovaranja grešaka pitanjem **"što"** uključuje sljedeće korake:

- Identifikacija greške: testni tim mora identificirati specifičnu grešku ili problem koji žele istražiti.
- Pitanje: testni tim mora postaviti pitanje "što" je problem sa specifičnom greškom.
- Istraživanje: testni tim mora istražiti uzroke problema i razumjeti ga.
- Prikupljanje dokaza: testni tim mora prikupiti relevantne informacije o problemu, uključujući izvješća o testiranju, snimke zaslona i druge dokaze.
- Zagovaranje: testni tim mora zagovarati ispravljanje problema prezentiranjem informacija i dokaza razvojnom timu.
- Dogovaranje: testni tim i razvojni tim moraju dogovoriti prioritet ispravljanja problema i vrijeme kada će se ispraviti.
- Praćenje: testni tim mora pratiti napredak ispravljanja problema i osigurati da se ispravi prema dogovoru.

Zagovaranje greške pitanjem **"što"** može pomoći u stvaranju preciznijeg razumijevanja problema i u ispravljanju grešaka na najbolji mogući način. Ovaj pristup zahtijeva visoku razinu istraživanja i komunikacije između testnog i razvojnog tima.

Gdje?

Zagovaranje greške pitanjem **"gdje"** otkriva gdje se problem pojavljuje u softveru. Ova se tehnika koristi kada testni tim želi locirati problem u softveru i razumjeti njegovu specifičnost. Glavni cilj ovog pristupa je osigurati da se problem identificira na najpreciznji mogući način i da se ispravi u što kraćem roku.

Proces zagovaranja grešaka pitanjem **"gdje"** uključuje sljedeće korake:

- identifikacija greške: testni tim mora identificirati specifičnu grešku ili problem koji žele istražiti
- pitanje: testni tim mora postaviti pitanje **"gdje"** se problem pojavljuje u softveru
- istraživanje: testni tim mora istražiti lokaciju problema i njegove specifičnosti
- prikupljanje dokaza: testni tim mora prikupiti relevantne informacije o problemu, uključujući snimke zaslona, podatke o bazi podataka i druge dokaze
- zagovaranje: testni tim mora zagovarati ispravljanje problema prezentiranjem informacija i dokaza razvojnom timu
- dogovaranje: testni i razvojni timovi moraju dogovoriti prioritet ispravljanja problema i vrijeme kada će se ispraviti
- praćenje: testni tim mora pratiti napredak u ispravljanju problema i osigurati da se ispravi prema dogovoru.

Zagovaranje greške pitanjem "gdje" može pomoći u brzom identificiranju i ispravljanju problema u softveru. Ovaj pristup zahtijeva visoku razinu istraživanja i preciznosti u identificiranju problema.

Tko?

Zagovaranje greške pitanjem **"tko"** istražuje tko je odgovoran za ovaj problem u softveru. Ova se tehnika koristi kada testni tim želi identificirati odgovornost za problem u softveru

i utvrditi tko će ga ispraviti. Glavni cilj ovog pristupa je osigurati da se problem identificira na najpreciznji mogući način i da se ispravi u što kraćem roku.

Proces zagovaranja grešaka pitanjem **"tko"** uključuje sljedeće korake:

- identifikacija greške: testni tim mora identificirati specifičnu grešku ili problem koji žele istražiti
- pitanje: testni tim mora postaviti pitanje "tko je odgovoran za ovaj problem u softveru?"
- analiza: testni tim mora analizirati sastav tima i njegove odgovornosti
- prikupljanje dokaza: testni tim mora prikupiti relevantne informacije o problemu, uključujući snimke zaslona, podatke o bazi podataka i druge dokaze
- zagovaranje: testni tim mora zagovarati ispravljanje problema prezentiranjem informacija i dokaza odgovornoj osobi
- dogovaranje: testni tim i odgovorna osoba moraju dogovoriti prioritet ispravljanja problema i vrijeme kada će se ispraviti
- praćenje: testni tim mora pratiti napredak u ispravljanju problema i osigurati da se ispravi prema dogovoru.

Zagovaranje greške pitanjem "tko" može pomoći u brzom identificiranju i ispravljanju problema u softveru. Ovaj pristup zahtijeva preciznost u identificiranju odgovorne osobe i dobru komunikaciju između testnog tima i odgovorne osobe.

Zašto?

Zagovaranje greške pitanjem **"zašto"** istražuje zašto se ovaj problem u softveru javlja. Ova se tehnika koristi kada testni tim želi identificirati uzroke problema u softveru i utvrditi što ga uzrokuje. Glavni cilj ovog pristupa je razumijevanje problema kako bi se moglo naći njegovo trajno rješenje.

Proces zagovaranja grešaka kroz pitanje **"zašto"** uključuje sljedeće korake:

- identifikacija greške: Testni tim mora identificirati specifičnu grešku ili problem koji žele istražiti

- pitanje: testni tim mora postaviti pitanje "zašto se ovaj problem u softveru javlja?"
- analiza: testni tim mora analizirati softver i podatke kako bi identificirao uzroke problema
- prikupljanje dokaza: testni tim mora prikupiti relevantne informacije o problemu, uključujući snimke zaslona, podatke o bazi podataka i druge dokaze
- zagovaranje: testni tim mora zagovarati za ispravljanje problema prezentiranjem informacija i dokaza relevantnim članovima tima
- dogovaranje: testni tim i relevantni članovi tima moraju dogovoriti prioritet ispravljanja problema i vrijeme kada će se ispraviti
- praćenje: testni tim mora pratiti napredak u ispravljanju problema i osigurati da se ispravi prema dogovoru.

Zagovaranje greške pitanjem **"zašto"** može pomoći u identificiranju trajnog rješenja za problem u softveru. Ovaj pristup zahtijeva preciznost u identificiranju uzroka problema i dobru analitičku sposobnost testnog tima.

Kada?

Zagovaranje greške pitanjem **"kada"** sagledava kada se ovaj problem u softveru javlja. Ova se tehnika koristi za identificiranje uvjetima i situacijama kada se problem u softveru javlja, kako bi se pomoglo u identificiranju uzroka. Glavni cilj ovog pristupa je razumijevanje kada se problem javlja, što može pomoći u identificiranju uzroka.

Proces zagovaranja grešaka pitanjem **"kada"** uključuje sljedeće korake:

- identifikacija greške: testni tim mora identificirati specifičnu grešku ili problem koji žele istražiti
- pitanje: testni tim mora postaviti pitanje "kada se ovaj problem u softveru javlja?"
- analiza: testni tim mora analizirati softver i podatke kako bi identificirao situacije i uvjete kada se problem javlja
- prikupljanje dokaza: testni tim mora prikupiti relevantne informacije o problemu, uključujući snimke zaslona, podatke o bazi podataka i druge dokaze

- zagovaranje: testni tim mora zagovarati za ispravljanje problema prezentiranjem informacija i dokaza relevantnim članovima tima
- dogovaranje: testni tim i relevantni članovi tima moraju dogovoriti prioritet ispravljanja problema i vrijeme kada će se ispraviti
- praćenje: testni tim mora pratiti napredak u ispravljanju problema i osigurati da se ispravi prema dogovoru.

Zagovaranje greške pitanjem **"kada"** može pomoći u identificiranju situacija i uvjeta kada se problem javlja u softveru, što može pomoći u identificiranju uzroka. Ovaj pristup zahtijeva preciznost u identificiranju situacija i uvjeta kada se problem javlja i dobru analitičku sposobnost testnog tima.

Kako?

Najvažnije pitanje je kako započeti sa zagovaranjem grešaka. Zagovaranje greške pitanjem **"kako"** istražuje se kako se ovaj problem u softveru javlja. Ova tehnika se koristi za identificiranje koraka i procesa koji dovode do pojave problema u softveru, kako bi se pomoglo u identificiranju uzroka. Glavni cilj ovog pristupa je razumijevanje kako se problem javlja, što može pomoći u identificiranju uzroka.

Proces zagovaranja grešaka pitanjem **"kako"** uključuje sljedeće korake:

- identifikacija greške: testni tim mora identificirati specifičnu grešku ili problem koji žele istražiti
- pitanje: testni tim mora postaviti pitanje "kako se ovaj problem u softveru javlja?"
- analiza: testni tim mora analizirati softver i podatke kako bi identificirao korake i procese koji dovode do pojave problema
- prikupljanje dokaza: testni tim mora prikupiti relevantne informacije o problemu, uključujući snimke zaslona, podatke o bazi podataka i druge dokaze
- zagovaranje: testni tim mora zagovarati za ispravljanje problema prezentiranjem informacija i dokaza relevantnim članovima tima
- dogovaranje: testni tim i relevantni članovi tima moraju dogovoriti prioritet ispravljanja problema i vrijeme kada će se ispraviti

- praćenje: testni tim mora pratiti napredak u ispravljanju problema i osigurati da se ispravi prema dogovoru.

Zagovaranje greške pitanjem **"kako"** može pomoći u identificiranju koraka i procesa koji dovode do pojave problema u softveru, što može pomoći u identificiranju uzroka. Ovaj pristup zahtijeva preciznost u identificiranju koraka i procesa koji dovode do pojave problema i dobru analitičku sposobnost testnog tima.

6.1.7. Komunikacija o defektima

Komunikacija o defektima u testiranju softvera ključni je dio procesa testiranja softvera. Uključuje razmjenu informacija između testera, programera i drugih dionika uključenih u projekt. Ova komunikacija treba sadržavati točan opis kvara, njegov uzrok i moguća rješenja.

Kako bi se pravilno komuniciralo o nedostacima, važno je osigurati da svi uključeni jasno razumiju što je defekt i kako je otkriven. To se može učiniti davanjem sažetka o defektu, njegovih mogućih uzroka i mogućih rješenja. Također je važno razgovarati o svim mogućim rizicima povezanim s defektom i svim potrebnim koracima za njihovo ublažavanje.

Osim toga, važno je osigurati da komunikacija između timova bude jasna i konzistentna te pravovremena i relevantna. To pomaže smanjiti nesporazume te omogućuje brzo i točno donošenje odluka. Konačno, važno je osigurati da je priopćavanje nedostataka dokumentirano. To pomaže osigurati da su svi uključeni svjesni defekta, njegovog uzroka i mogućih rješenja. Ovo također daje zapis rasprava za buduće potrebe.

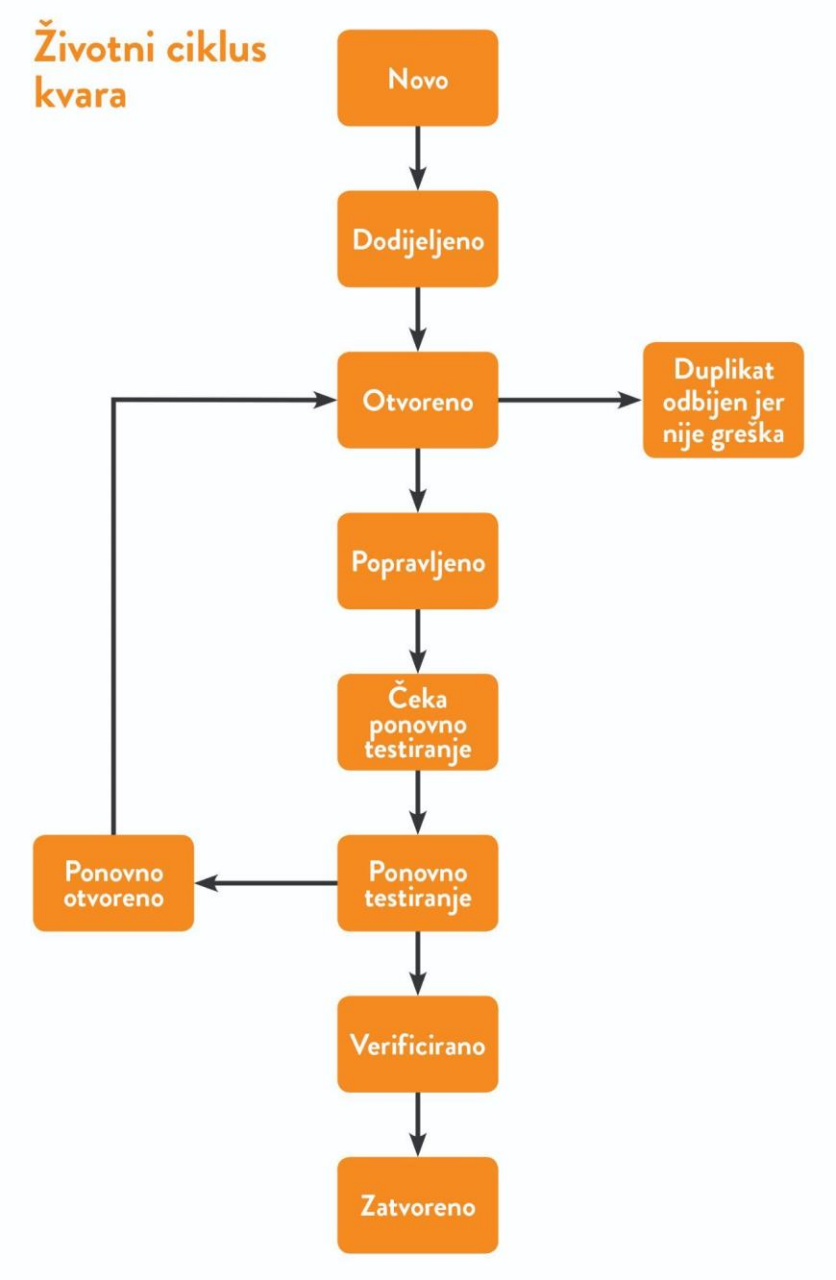
6.1.8. Životni ciklus defekta od objavljivanja do zatvaranja

Životni ciklus defekta (engl. *Bug Life Cycle*) u testiranju softvera specifičan je skup stanja kroz koja defekt prolazi tijekom svog cijelog životnog vijeka. Cilj životnog ciklusa defekta je jednostavno koordiniranje i komuniciranje trenutnog statusa kvara koji se mijenja te učiniti proces otklanjanja defekta sustavnim i učinkovitim.

Status defekta (kvara) u životnom ciklusu je trenutno stanje iz kojeg defekt trenutno proizlazi. Cilj statusa defekta jest precizno prenijeti njegovo trenutno stanje ili napredak kako bi se bolje pratio i razumio stvarni napredak životnog ciklusa defekta. Broj stanja kroz koja defekt prolazi varira od projekta do projekta.

Donji dijagram životnog ciklusa pokriva sva moguća stanja:

- **novo:** kada se prvi put zabilježi i objavi novi defekt, dodijeljen mu je status NOVO.
- **dodijeljeno**: nakon što je tester objavio defekt, voditelj tima testera ga odobrava i dodjeljuje ga timu razvojnih programera
- **otvoreno:** programer počinje analizirati i radi na ispravljanju defekta
- **popravljeno**: kada programer napravi potrebnu izmjenu koda i potvrdi promjenu, tada može postaviti status defekta kao "popravljeno"
- **ponovno testiranje na čekanju**: nakon što je defekt popravljen, programer daje određeni kod za ponovno testiranje koda testeru. Budući da testeri testiranje softvera stavljaju na čekanje, dodijeljeni status je "ponovno testiranje na čekanju"
- **ponovno testiranje**: tester ponovno testira kod u ovoj fazi kako bi provjerio je li defekt popravio programer ili nije i mijenja status u "ponovno testiraj"
- **potvrđeno**: tester ponovno testira defekt nakon što ga je programer ispravio. Ako u softveru nije otkriven defekt, znači da je defekt ispravljen i da mu se dodjeljuje status "provjereno"
- **ponovno otvori:** ako se defekt nastavi čak i nakon što ga je programer ispravio, tester mijenja status u "ponovno otvoreno". Još jednom defekt prolazi kroz životni ciklus
- **zatvoreno**: ako defekt više ne postoji, tester dodjeljuje status "zatvoreno"



Slika.6.1. Životni ciklus defekta

- **duplikat**: ako se defekt ponavlja dvaput ili kvar odgovara istom konceptu, status se mijenja u "duplikat"
- **odbijeno**: ako razvojni programer smatra da defekt nije pravi defekt, mijenja nedostatak u "odbijen"

- **odgođeno**: ako sadašnji defekt nije primarnog prioriteta i ako se očekuje da će biti ispravljen u sljedećem izdanju, tada se status "odgođeno" dodjeljuje takvim defektima
- **nije *bug***: ako ne utječe na funkcionalnost aplikacije, tada je status dodijeljen defektu "nije *bug*".

Objašnjenje o životnom ciklusu defekta:

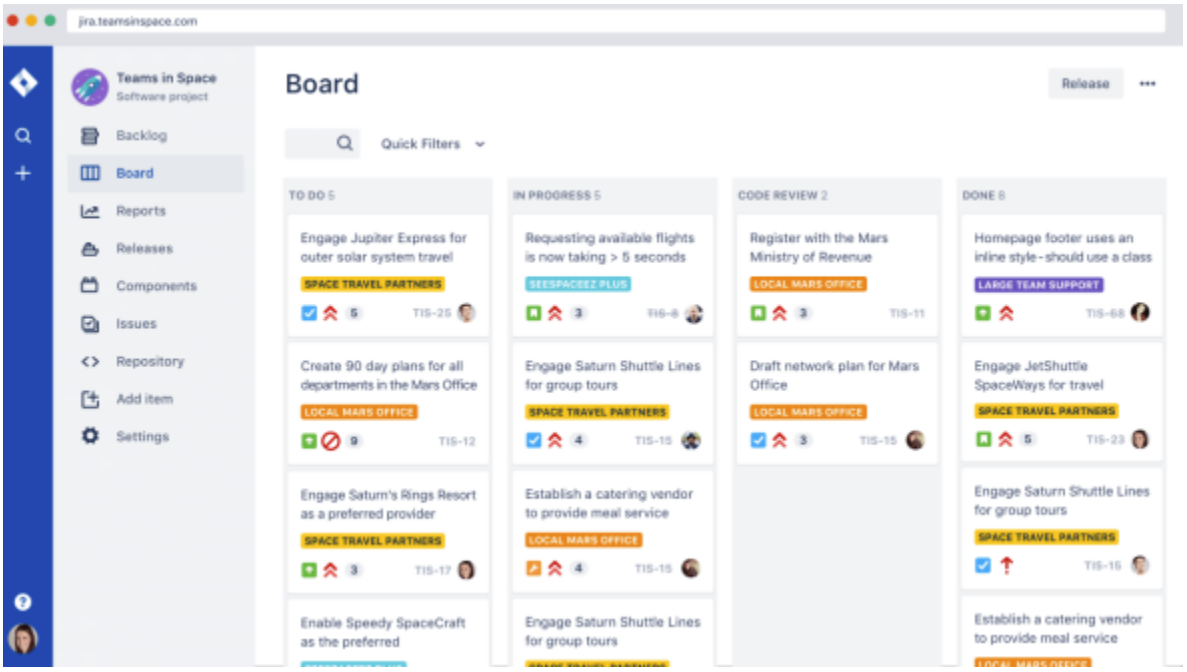
- tester pronalazi defekt
- status dodijeljen defektu **- "novo"**
- efekt se prosljeđuje voditelju projekta na analizu
- voditelj projekta odlučuje je li defekt valjan
- defekt nije valjan - daje se status **"odbijeno"**.

Dakle, voditelj projekta dodjeljuje status odbijeno. Ako defekt nije odbačen, sljedeći korak je provjeriti je li u opsegu (tj. ulazi li u popis zadataka koji će se pustiti u produkcijsko okruženje). Pretpostavimo da imamo funkciju e-pošte za aplikaciju koju razvijamo, a mi u tome pronađemo problem. Ako funkcija nije dio trenutnog izdanja (verzije aplikacije koja se trenutno razvija) tada se takvim nedostaci dodijeljuje status "odgođeni", a zatim se provjerava je li sličan defekt ranije istaknut. Ako da, defektu se dodjeljuje "**duplikat statusa"**. Ako nije, defekt se dodjeljuje programeru koji počinje popravljati kod. Tijekom ove faze, defektu se dodjeljuje status "**u tijeku".** Nakon što se šifra popravi, **defektu** se dodjeljuje status "**otklonjen".** Zatim će tester ponovno testirati kôd. U slučaju da testni slučaj prođe, defekt se **zatvara.** Ako testni slučajevi ponovno ne uspiju, defekt se **ponovno otvara** i dodjeljuje programeru.

6.1.9. Alati za prijavljivanje defekata

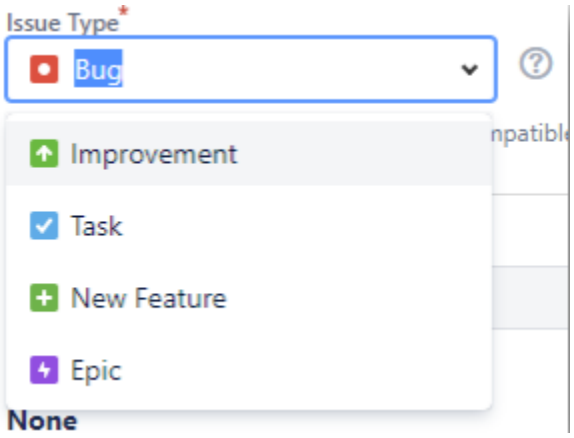
Što je Jira?

Jira je *online* alat za upravljanje softverskim projektima. Koristan je za rad u većim timovima i često se koristi u organizaciji test projekata.



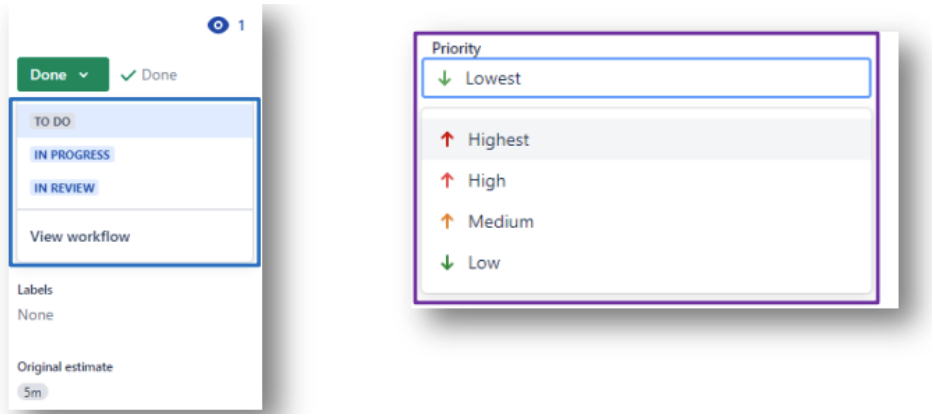
Slika. 6.2. Prikaz projekta kako u Jiri

Osnovni entitet u Jiri je neki problem (engl. *Issue*), a svi problemi raspoređeni su po projektima. Jedan *Jira* projekt sadrži mnoštvo stavki na kojima se trenutno radi. Stavke su ključna jedinica u Jiri. Procesi koji se u Jiri događaju podrazumijevaju definiranje, različitu vrstu obrade, komentiranje i logiranje stavki i slično. Stavka mora pripadati nekom projektu, imati opis i tip i korisnika koji je prijavljuje. Tip stavke mora se odrediti prilikom njenog kreiranja i ne može se naknadno mijenjati.



Slika 6.3. Prikaz stavki koje se prijavljuju u Jiri

Cilj stavke je njeno rješavanje. Jednom kreiranu stavku konstantno mijenjamo. Posebna vrsta stavki može biti vezana za glavnu stavku. Ovakve se stavke nazivaju podzadaci (*subtasks*). Korisni su ako hoćemo glavni zadatak podijeliti na više korisnika. Status stavke označava fazu u kojoj se ona trenutno nalazi dok prioritet označava važnost stavke.



Slika 6.4. Status stavki i njezina prioritetnost prikaza u Jiri

Cilj svake stavke je da bude konačno razriješena. Razrješenja ne moraju biti obavezno i rješenje nekog problema, ovisno o radnom toku (engl. W*orkflow*), razlikuju se i opcije razrješenja.

Pitanja za ponavljanje:

1. Što spada pod elemente izvještaja?
2. Koja je uloga testera u zagovaranju bugova?
3. Što spada pod trivijalni defekt?
4. Kojih pet pitanja postavljamo kod zagovaranja bugova?
5. Kako nam Jira pomaže u softverskom testiranju?

7. Automatiziranje testova softvera

U OVOM POGLAVLJU NAUČIT ĆETE:

- što je Cucumber i kako se Gherkin povezuje s programskim jezikom
- kreirati automatizacijski test za testiranje s Cucumberom i Seleniumom
- kako testirati *web*-servis i REST API-a
- kreirati i izvršiti testove u alatu JMeter
- osmisliti automatizirane testove performansi u alatu JMeter.

Automatizirano testiranje softvera je postupak korištenja specijaliziranih softverskih alata za automatsko izvršavanje testova na softverskoj aplikaciji. Cilj automatiziranog testiranja je **identificirati nedostatke, pogreške i probleme s performansama** što je ranije moguće u razvojnom ciklusu, poboljšati kvalitetu i pouzdanost softvera i smanjiti vrijeme i trud potrebne za ručno testiranje.

Automatizirano testiranje može obuhvatiti širok raspon aktivnosti testiranja, uključujući testiranje jedinica, funkcionalno testiranje, integracijsko testiranje i testiranje performansi, između ostalog. Automatizirani testovi mogu se izvoditi opetovano i dosljedno, omogućujući programerima da brzo i jednostavno potvrde promjene u kodu i primijete

probleme rano u procesu razvoja. To dovodi do bržih povratnih informacija, bolje suradnje između programera i testera te poboljšane ukupne kvalitete softvera.

7.1. Sustav za testiranje Cucumber i automatizacije izvršavanja BDD scenarija

Cucumber je alat za razvoj vođen ponašanjem (BDD) za pisanje testnih slučajeva. Služi za opisivanje ponašanja sustava prirodnim jezikom s određenim ključnim riječima. Proces počinje stvaranjem datoteke značajki, koja objašnjava značajku sustava i neke scenarije različitih testnih situacija. Budući da Cucumber ne može sam protumačiti značajke, sljedeći korak je stvaranje definicija koraka koje mu objašnjavaju što učiniti kada pronađe taj korak u jednom od scenarija. Definicije koraka napisane su u programskom jeziku **Ruby**. To daje veliku fleksibilnost u načinu na koji se testni koraci izvršavaju. Može koristiti i definicije koraka za **Webrat**, alat za testiranje prihvatljivosti za Ruby, koji simulira preglednik bez podrške za **Javascript** (programski jezik koji se koristi za dinamičnost neke *web*-stranice ili aplikacije). Također se može kombinirati s okvirom za automatizaciju *web*-stranice, kao što je Watir koji služi za implementaciju testova *web*-automatizacije temeljenih na pregledniku.

7.1.1. Cucumber – povezivanje BDD scenarija u Gherkinu s izvršnim kodom odabranog programskog jezika

Kod isporuke projekta koji se razvija prema BDD metodologiji, tim se fokusira na dostavljanje visokokvalitetnog softvera koji zadovoljava korisničke zahtjeve i očekivanja. U ovom procesu isporuke, BDD scenariji služe kao glavni alat za osiguravanje kvalitete i provjeru funkcionalnosti softvera. Kako biste zadovoljili poslovne zahtjeve, morate točno razumjeti što želite da softver postigne iz poslovne perspektive. U BDD-u to postaje vaš poslovni cilj: pokretačka snaga projekta.

Evo primjera jednostavnog opisa značajke iz datoteke značajki Cucumber:

```
Feature: Website Navigation
In order to navigate website pages
I need to be able to click on menu item
```

Značajka je objašnjenje onoga što želimo postići, a koja sama po sebi zapravo ne čini mnogo. Spomenutu značajku koju smo naveli gore testirat ćemo s nekoliko scenarija. Na primjer, korisnik može biti na početnoj *web*-stranici na početku testa i želi kliknuti na drugu poveznicu. Jedan scenarij ćemo prikazati kako bi vizualno što bolje pokazali sintaksu scenarija. Opisujemo radnju klikom na stavku izbornika na *web*-stranici:

```
Scenario: Going to page Services
Given that I am on spriteCloud Home
When I click on link Services
Then the page title should be "spriteCloud - Services"
```

Ovo je dio koji trebamo objasniti Cucumberu kako bi znao što treba učiniti. U ovom slučaju koristimo Watir[6] s opcijom *Watir-webdriver*. Također koristimo *RSpec gem*[7] kako bismo mogli koristiti operaciju.

```
require 'spec'

require "watir-webdriver"
```

Zatim definiramo neke konstante, jednu za adresu stranice i drugu za upućivanje na preglednik. Obratite pažnju na sintaksu skripte za pokretanje instance *Firefoxa* s *web-driverom*.

[6] *Watir* je open-source *Ruby* biblioteka za automatizaciju *web*-preglednika. Pokreće preglednike *Internet Explorer, Firefox, Chrome, Operu* i *Safari*, a dostupan je kao *RubyGems*.

[7] RSpec gem je meta-gem, koji ovisi o rspec-core, rspec-expectations i rspec-mocks draguljima. Svaki od njih može se instalirati odvojeno i učitati izolirano pomoću zahtjeva

```
SITE = "www.spritecloud.com"
BROWSER = Watir::Browser.start(SITE, :firefox)
PAGES = {
 "spriteCloud Home" => "http://www.spritecloud.com/"
}
```

Sada kada smo obradili dio postavljanja, moramo početi s koracima uključenim u scenarij. Ključna riječ "*Given*" odnosi se na postavljanje testa. Moramo biti na početnoj stranici na početku testa.

```
Given /^that I am on (.*)$/ do |page|
 BROWSER.goto(PAGES[page])
end
```

Primijetit ćete da je sintaksa koraka oblikovana kao regularni izraz pisan kao kod Gherkina. ***Given*** je ključna riječ, ali sve ostalo je nešto što Cucumber može prepoznati kao dio vašeg testa. Razdvojimo malo taj dio:

- rečenica mora početi ključnom riječi "*Given*", ovo je obavezno.
- ono što slijedi je razmak i riječi "*that I am on*"
- još jedan regularni izraz (.*), koji odgovara svemu od "*on*" do kraja retka "*storing*" koji će se kasnije upotrebiti za stranicu varijable

Sljedeća ključna riječ, "*When*", odnosi se na operaciju koju radimo na stranici. U ovom slučaju želimo kliknuti poveznicu na stranici. Veza se može pronaći s nekoliko različitih metoda u *Watiru*, ovdje koristimo metodu ***:text***, koja u osnovi klikne na vezu s tekstom koji navedemo u scenariju.

```
When /^I click on link (.*)$/ do |link|
BROWSER.link(:text, link).click
end
```

Posljednja ključna riječ u ovom primjeru, "*When*" odnosi se na rezultat koji očekujemo. Za potrebe ovog primjera, provjeravamo je li naslov stranice točan, tj. isti kao u definiciji testnog scenarija.

```
Then /^the page title should be "([^"]*)"$/ do |title|

 BROWSER.title.eql?(title).should == true

end
```

Ova vrsta testa može se pokrenuti ili tijekom faze implementacije, što je prvotna namjena, ili kasnije kada stranica bude potpuno spremna. Čak i ako je test kreiran tijekom faze implementacije, svejedno bi ga trebalo pokrenuti sa svakim novim izdanjem koje izađe. Može se čak koristiti kao dio samostalnog testnog skupa za *web*-stranicu.

Prednost Cucumbera je oblikovan izlaz testa koji odmah prikazuje uspješne i neuspješne testove i testne korake. Uspješni testovi označeni su zelenom bojom na terminalu, a neuspješni crvenom bojom.

Mbp-2:features marko$ cucumber
Feature: Navigacija na web-stranici
Kako bi navigirali po stranicama
Moram moći kliknuti na navigaciju

Scenarij: Odlazak na stranice servisa
Pošto se nalazim na spriteCloud Home
Kada kliknem na link servisa
Tada bi naziv stranice trebao biti “spriteCloud - Services”

1 scenarij (1 prošao)
koraka (3 prošla)
0m1.261s

7.1.2. Složeniji scenariji

Zamislite da je tijek rada vašeg softvera takav da bi, ovisno o nekim kriterijima, vaši testovi trebali slijediti drugačiji put kroz softver. U Cucumberu svaki put mora biti zaseban testni scenarij jer grananje na drugi put Cucumber ne podržava.

Ovaj jednostavan tijek rada može se primijeniti na primjeru kada se uzima u obzir odakle se klijent povezuje. U primjeru navedenom u nastavku zaprimljena je tehnička specifikacija za opciju plaćanja koja je navedena za EU tržište i glavna valuta u specifikaciji bit će euro. Sljedeći zahtjev koji se može dobiti je da će aplikacija biti predstavljena i na drugim tržištima koja nemaju plaćanje eurom kao glavnom valutom. U ovom slučaju kada bi se aplikacija pripremala za npr. Ujedinjeno kraljevstvo, plaćanje bi se temeljilo na IP adresi korisnika aplikacije. Ako uzmemo u obzir takav slučaj, mogli bismo napraviti scenarij poput ovog:

```
Feature: Checkout
 Scenario Outline: International checkout
 Given I am on the checkout page
 When I enter my email address
 And I enter my password
 And I select
 And I provide
 Then I expect the payment details to be accepted
 And I expect to see an order confirmation page

 Examples:
```

```
| payment method | payment details |
| credit card | my CC number |
| paypal | my paypal email |
```

Ovaj je primjer posebice koristan ako se izrazi kao nacrt scenarija jer daje naznaku kako se jednostavne grane mogu uhvatiti u Cucumberu/Gherkinu, koristeći nacrte scenarija i tablice primjera.

Gore navedeni primjer također je prilično umjetan jer proces plaćanja kreditnom karticom obično zahtijeva da navedete ne samo broj kreditne kartice, već i ime vlasnika kartice i kontrolni kod. U nekim slučajevima može uključivati i dodatni korak provjere nakon toga.

Za drugi scenarij, možemo ponovno upotrijebiti dio prvog scenarija odozgo. Za početak, početne linije bit će identične.

```
Given I am on the checkout page
When I enter my email address
And I enter my password
```

Kraj bi trebao biti potpuno isti:

```
Then I expect the payment details to be accepted
And I expect to see an order confirmation page
```

Naš problem s grananjem može se vidjeti iz druge perspektive, ako pretpostavimo da se nešto može ponovno koristiti u svim scenarijima i kako su oni drugačiji. Ako se promatra u takvom svjetlu, problem nije toliko u suočavanju s granama, već u tome kako najbolje ponovno upotrijebiti postojeće korake. Prva stvar koja bi se trebala učiniti je usvojiti

najbolju praksu ***push how down***[8], što se odnosi na manje detaljne korake, a detalje ostaviti definicijama koraka.

Imajući to na umu, početni blok može se prepisati ovako:

```
Given I provide my credentials on the checkout page
Then I expect the checkout process to complete
```

Takav kompleksniji scenarij bi izgledao ovako:

```
Given /^I provide my credentials on the checkout page$/
 steps %Q{
 Given I am on the checkout page
 When I enter my email address
 And I enter my password
 }
end
```

7.1.3. Odabir govornog jezika za pisanje scenarija – hrvatski vs. engleski

Svoje testne slučajeve možete napisati na materinjem jeziku u obliku datoteka značajki u Cucumberu. Gherkin koristi skup posebnih ključnih riječi kako bi dao strukturu i značenje izvršnim specifikacijama. Svaka ključna riječ prevedena je na mnoge jezike. Potrebno je da svaka linija koja se koristi u Cucumberu počinje sintaksom Gherkina.

Najvažnije ključne riječi su:

- given
- when

[8] *Push how down* u kontekstu testiranja softvera predstavlja proces simuliranja opterećenja ili velikog broja zahtjeva kako bi se testirala stabilnost, performanse i ograničenja sustava.

- then
- and
- but

Postoji još i nekoliko sekundarnih poput:

- | - podatkovne tablice
- @ - oznake
- # - komentari

Na hrvatskom jeziku gornji primjer bi izgledao ovako:

```
Feature: Navigiranje po stranici
Given da sam na spriteCloud Home
When kliknem na poveznicu Usluge
Then naslov stranice trebao bi biti "spriteCloud - usluge"
```

#language: oznaka u zaglavlju u prvom retku značajke datoteke govori Cucumberu koji govorni jezik treba koristiti: na primjer # language: fr za francuski. Ako izostavite ovo zaglavlje, Cucumber će prema zadanim postavkama biti engleski (en).

Neke implementacije Cucumbera omogućuju postavljanje zadanog jezika u konfiguraciji, tako da nije potrebno postavljati zaglavlje # jezika u svaku datoteku.

7.2. Automatizacija testiranja *web*-sučelja sustavima za testiranje Cucumber i Selenium

Automatizirano testiranje *web*-sučelja uključuje korištenje softverskih alata za automatizaciju procesa testiranja *web*-aplikacija. **Cucumber** i **Selenium** popularni su alati koji se koriste u tu svrhu. **Cucumber** je alat primjeren za razvoj vođen ponašanjem (BDD) koji omogućuje stvaranje automatiziranih testova u formatu prirodnog jezika. Pruža

okvir za pisanje testova na poslovno čitljivom jeziku, olakšavajući netehničkim dionicima da razumiju testove.

Selenium je softver otvorenog koda koji se može potpuno besplatno koristiti i služi za automatizaciju *web*-aplikacija. Može se koristiti za automatizaciju raznih vrsta testova, uključujući funkcionalno testiranje i testiranje performansi. Može komunicirati s *web*-stranicom kao što bi to učinio korisnik - klikanjem gumba, ispunjavanjem obrazaca i slično.

Zajedno, Cucumber i Selenium mogu se koristiti za automatizaciju testiranja *web*-sučelja, pri čemu Cucumber pruža testne scenarije temeljene na prirodnom jeziku, a Selenium osigurava automatizaciju funkcionalnosti za interakciju s *web*-sučeljem i pokretanje testova. To može unaprijediti učinkovitost i točnost procesa testiranja i smanjiti vrijeme i trud potrebno za ručno testiranje.

7.2.1. Obrazac *Page Object*

Page Object Model, također poznat kao POM, obrazac je dizajna u Seleniumu koji stvara repozitorij objekata za pohranu svih *web*-elemenata. Koristan je za smanjenje problema dupliciranja koda i poboljšava održavanje testnih slučajeva.

U obrascu *Page Object* svaka *web*-stranica aplikacije smatra se **datotekom klase**[9]. Svaka datoteka klase sadržavat će samo odgovarajuće elemente *web*-stranice. Koristeći te elemente, testeri mogu izvoditi operacije na *web*-stranici koja se testira.

[9]Datoteka klase (engl. *Class file*) je binarna datoteka koja sadrži prevedeni kod jedne Java klase. Kada se napiše program u programskom jeziku Java, on se prevodi u bajtkod koji je zapisan u datoteci klase. Datoteka klase se može pokrenuti na bilo kojoj platformi koja podržava virtualnu mašinu Java (JVM), bez obzira na to na kojoj platformi je datoteka klase prevedena.
Svaka Java klasa se prevodi u jednu datoteku klase s istim nazivom kao i ime klase, ali sa dodatkom .class ekstenzije. Na primjer, ako imamo klasu nazvanu "MojProgram", njezina datoteka klase će se zvati "MojProgram.class". Datoteka klase se obično stvara prevođenjem Java izvornog koda.

Obrazac Page Object pomaže pri jednostavnom održavanju: koristan je kada postoji promjena u elementu korisničkog sučelja ili postoji promjena u radnji. Primjerice, ako je padajući izbornik promijenjen u radio gumb, u tom slučaju Page Object pomaže identificirati stranicu ili zaslon koji treba izmijeniti. Budući da će svaki zaslon imati različite Javascript datoteke[10], ova identifikacija je neophodna kako bi se izvršile potrebne promjene u pravim datotekama. **To čini testne slučajeve jednostavnima za održavanje i smanjuje pogreške** te također pomaže kod ponovnog korištenja koda. Korištenjem Page Object obrazaca, možete koristiti testni kod za jedan zaslon i ponovno ga koristiti u drugom testnom slučaju. Nema potrebe za prepisivanjem koda, čime se štedi vrijeme.

Obrazac Page Object ima prednost čitljivosti i pouzdanosti skripti. Kada svi zasloni imaju neovisne Java datoteke, lako se mogu identificirati radnje koje će se izvesti na određenom zaslonu kretanjem kroz Java datoteku. Ako se mora napraviti promjena u određenom odjeljku koda, to se može učinkovito učiniti bez utjecaja na druge datoteke.

7.2.2. Osnovne Selenium metode

Selenium predstavlja skup alata različitih mogućnosti za automatizirano testiranje *web*-aplikacije. Zahvaljujući raznovrsnosti alata, od kojih svaki ima drugačiji pristup testiranju i rješavanju specifičnih problema automatizacije, Selenium podržava testiranje *web*-aplikacije u gotovo svim dostupnim pretraživačima. Testne skripte mogu biti pisane u različitim programskim jezicima kao što su C#, Java, Ruby, Python i Perl te pokretani na Windows, Macintosh ili Linux operacijskim sustavima. Komponente koje Selenium uključuje su:

- Selenium *IDE*
- Selenium *WebDriver*
- Selenium *Grid.*

[10] JavaScript datoteke su tekstualne datoteke koje sadrže JavaScript kod. JavaScript je popularan skriptni jezik koji se koristi za razvoj dinamičkih *web*-stranica i aplikacija. JavaScript datoteke su obično pohranjene sa .js ekstenzijom.

Svaka testna skripta u Selenium *WebDriveru* i Selenium *IDE-u* počinje od stranice koja se testira. Osnovna metoda za prelazak na željenu stranicu koja se koristi u Selenium *WebDriveru*, je:

driver.get("http://www.google.com");

Osim ***get()*** metode, za prelazak na stranicu se koristi i ***navigate().to()*** metoda koja kao argument također prihvaća URL željene stranice. *Get()* i *navigate().to()* metode se primjenjuju na potpuno isti način, razlika je samo u zapisu. *Navigate()* metoda javlja se u još dva oblika koja omogućavaju kretanje unaprijed i unazad kroz povijest pretraživanja:

navigate().forward() navigate().back()

Metoda ***switchTo()*** omogućava prelazak s trenutne stranice na otvoreni okvir ili novi otvoreni prozor na trenutnoj stranici, kao i vraćanje s novootvorenog prozora na trenutnu stranicu.

driver.switchTo().window(<naziv prozora>);
driver.switchTo().frame(<naziv okvira>);

Jedna od novosti Selenium *WebDrivera* u odnosu na prethodne verzije Seleniuma je podrška „skočnih" prozora (engl. *Pop-up window*). Uz pomoć ***switchTo().alert()*** metode, može se pristupiti tekstu ispisanom na prozoru:

Alert alert = driver.switchTo().alert();

Ova naredba otvara skočni prozor s porukom kojim se sada može upravljati – da se prihvati ili odbije upozorenje i da se čita ili mijenja tekst poruke.

7.2.3. Vrste selektora

Selenium *WebDriver* koristi različite metode za lociranje elemenata na *web*-stranici. Međutim, prije pisanja samog koda, potrebno je pažljivo analizirati samu stranicu i elemente kako bi se razumjela njihova struktura u aplikaciji, na primjer, koji su atributi definirani za određene elemente, kakvi su JavaScript i Ajax[11] pozivi napravljeni iz aplikacije i slično.

Pretraživač prikazuje elemente u aplikaciji, skrivajući HTML kôd za krajnje korisnike. Zato se koriste različiti alati koji olakšavaju proučavanje koda u pozadini. Neki od njih su:

- **Firebug** za Firefox
- **Google Developer Tools** za Chrome
- **Web Inspector** za Safari.

To su takozvani *Browser Developer* alati koji su ugrađeni u preglednike i pružaju informacije na osnovu kojih se izvodi lociranje elemenata, kao i informacije o njihovim atributima, JavaScript pozivima, stilovima atributa, izvorima stranica i slično.

HTML pozadina *web*-stranice otvara se tipkom F12 na tipkovnici ili, ako se izabere neki element na *web*-stranici, klikom desnim klikom miša te se u prikazanoj listi odabere ***Inspect Element***.

Za lociranje elemenata u Selenium *WebDriveru* koriste se dvije metode koje su osigurale *WebDriver* i *WebElement* klase. *WebDriver* klasa predstavlja sučelje za upravljanje preglednikom. Pomoću ove klase možemo upravljati preglednikom, kao što je otvaranje stranice, klikanje na elemente, ispunjavanje formi, izvođenje JavaScript koda i sl. Za lociranje elemenara koriste se ***findElement()*** i ***findElements()*** metode. Metoda *findElement()* vraća prvi element *web*-stranice (objekt klase *WebElement*) koji zadovoljava kriterij na osnovu kojeg se izvodi pretraga. Ako ne uspije pronaći element

[11] AJAX (Asynchronous JavaScript and XML) je skup tehnologija koje se koriste za dinamičko učitavanje sadržaja *web*-stranice bez potrebe za ponovnim učitavanjem cijele stranice

koji odgovara kriteriju pretrage, metoda *findElement()* baca iznimku ***NoSuchElementFindException***. Metoda *findElements()* vraća listu *web*-elemenata koji zadovoljavaju kriterij lociranja elementa. Ako ne postoje takvi elementi, vraća praznu listu. Ova metoda je jako korisna kada se radi s grupom sličnih elemenata. Na primjer, možemo dobiti sve linkove prikazane na stranici, sve redove iz tablice ili sve vrijednosti iz padajuće liste. Selenium *WebDriver* osigurava *By* klasu, čiji se objekt koristi kao argument *„find*" metode, kako bi podržao različite načine lociranja elemenata na *web*-stranici. To su:

- **Lociranje elemenata putem ID-a** – najjednostavniji, najbrži i najefikasniji način za lociranje elemenata na *web*-stranici. U usporedbi s tekstom, skriptama koje koriste, ID-jevi su najmanje skloni promjenama u aplikaciji, zato je ovaj način lociranja elemenata i najzastupljeniji. Da bi lociranje elemenata ovom metodom bilo uspješno, potrebno je da različiti elementi imaju različite ID-eve, to jest da su ID-evi jedinstveno određeni. Poziv metode *findElement()*, u slučaju pretrage po ID-u, izgleda:

WebElement element = driver.findElement(By.id(<ID elementa>))

- **Lociranje elemenata putem imena atributa** – ime atributa je najčešće korišteno za lociranje elemenata kao što su tekst polja ili radio gumba. Kao i kod ID-a, mala je vjerojatnost promjene imena atributa. Ime atributa ne mora da biti jedinstveno te, u slučaju da postoji više elemenata s istim imenom, metoda vraća prvi element na strani s definiranom vrijednosti, što možda i nije element koji se zahtijeva. To može izazvati prekid testiranja.

WebElement element = driver.findElement(By.name(<vrijednost imena atributa>))

- **Lociranje elemenata putem imena klase** – Uloga atributa klase u HTML dokumentima je da omogući primjenu *Cascading Style Sheets*[12] na odgovarajući element, ali se također se može koristiti i za identificiranje elemenata.

WebElement element = driver.findElement(By.className(<vrijednost class atributa))

- **Lociranje elemenata putem imena etikete** – By klasa Selenium *WebDrivera* ima metodu *tagName()* za lociranje elemenata na osnovu njihove HTML etikete. Ova metoda je veoma korisna kada je potrebno locirati na primjer, sve etikete u nekoj tablici. Sljedeći segment koda prikazuje na koji način se mogu prebrojati redovi u tablici:

WebElement tabela = driver.findElement(By.id("Tabela")); List redovi = table.findElements(By.tagName("tr")); assertEquals(10, redovi.size());

- **Lociranje elemenata putem CSS selektora** – CSS selektor je predložak i dio CSS pravila koji se poklapa sa skupom elemenata u HTML dokumentu. CSS selektori raspolažu velikim brojem metoda, pravila i predložaka za lociranje elemenata na *web*-stranici koje i Selenium *WebDriver* koristi. Neki od njih su lociranje putem apsolutne putanje, lociranje putem relativne putanje, putem ID selektora, imena atributa i tako dalje. Lociranje elemenata putem CSS selektora brže je i pouzdanije u odnosu na lociranje putem *XPath* lokatora.

- **Lociranje elemenata putem XPath-a** – *XPath* predstavlja „drvoliku" reprezentaciju XML[13] dokumenta i omogućava navigaciju kroz drvo izborom čvorova po različitim kriterijima. *Selenium WebDriver* podržava *XPath* za lociranje

[12] CSS (Cascading Style Sheets) je stilski jezik koji se koristi za definiranje izgleda i dizajna *web*-stranica. CSS se koristi za definiranje različitih stilova i svojstava HTML elemenata, kao što su font, boja, pozadina, veličina i raspored elemenata na stranici.

[13] XML (*eng.eXtensible Markup Language*) je označivački jezik koji se koristi za strukturiranje podataka u obliku teksta. XML se koristi za označavanje elemenata i atributa unutar dokumenata koji sadrže podatke. Dokumenti koji se temelje na XML-u su lako čitljivi i razumljivi ljudima i strojevima.

elemenata korištenjem *XPath* izraza ili upita. Međutim, ova metoda za lociranje elemenata jako je spora i zbog toga nije baš pogodna za primjenu. Važna razlika između ove i metode lociranja elemenata putem CSS selektora je što se, korištenjem *XPatha*, mogu locirati elementi penjući se i spuštajući u hijerarhiji elemenata, dok CSS selektor dopušta samo spuštanje kroz hijerarhiju. To znači da *XPath* metodom možemo locirati „roditeljski" element polazeći od „dijete" elementa. *XPath*, kao i CSS, također raspolaže velikim brojem načina za lociranje elemenata putem apsolutne putanje, relativne putanje, ID selektora, atributa i drugih.

7.2.4. Povezivanje Cucumber koda i *feature* datoteka

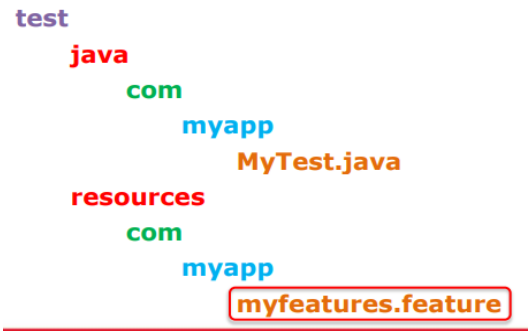
Da bi se Cucumber mogao pokrenuti, mora postojati aktivacijska klasa unutar testnog direktorija. Klasa ne treba sadržavati anotacije za pokretanje Cucumbera. *Ghekrin* kod se piše unutar *feature fileova*. ***Feature fileovi*** obični tekstualni su dokumenti s proširenim *featureom*.

myfeature.feature

Feature:

My first cucumber feature

Feature datoteke nalaze se u *resource* direktoriju u okviru hijerarhije poddirektorija, koja odgovara hijerarhiji poddirektorija testne klase. Vizualno to izgleda ovako:



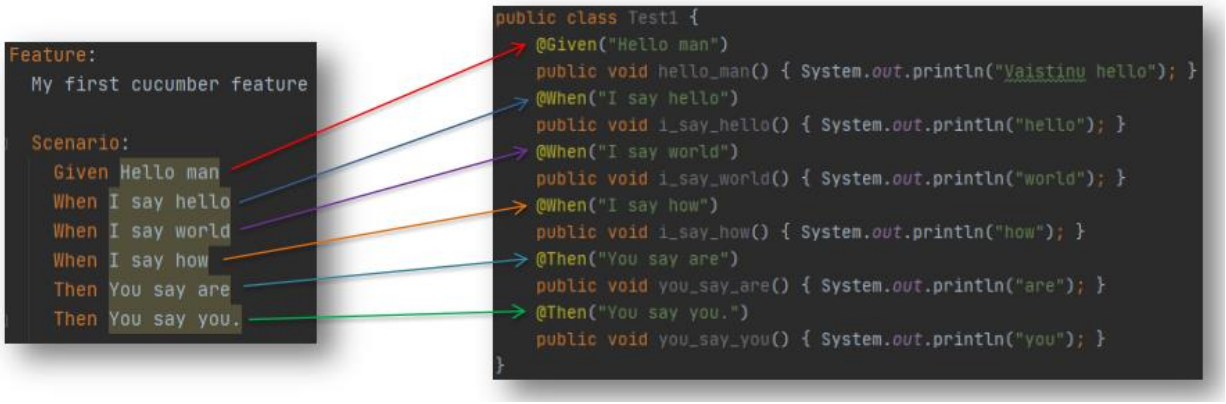
Slika 7.1. Vizualni prikaz kako se datoteke nalaze u feature datoteci

Kod pisanja scenarija trebamo se pridržavati nekih smjernica:

- *feature* sadrži jedan ili više scenarija
- svaki scenarij se sastoji od koraka
- jedan korak može biti jedna od Gherkin akcija *(given, when, then, and* ili *but)*
- akcije se izvršavaju serijski, jedna za drugom (prvo *given*, pa *when*, pa *then*…)
- svaki korak mora biti ispraćen odgovarajućom metodom u tekst klasi[14]
- metoda mora imati anotaciju koja odgovora definiranom koraku.

Slika 7.2. Prikaz koda u Javi s Cucumberom

Jedan scenarij najčešće sadrži više koraka, a svaki korak mora biti ispraćen odgovarajućom anotacijom koja ne smije biti duplicirana.



Slika 7.3. Prikaz kako scenarij pisan u Cucumberu izgleda u kodu pisanim u Javi

[14]U programiranju, tekst klasa predstavlja osnovnu klasu ili tip podataka za rad s tekstualnim podacima u programskom jeziku. Tekst klasa se koristi za pohranu i manipulaciju nizova znakova ili teksta u programu.

Kod provjere točnosti (engl. *Assertions*) Cucumber nema mehanizme za provjeru točnosti, već se u tu svrhu koriste druge biblioteke *(JUnit, TestNG…*).

```
Scenario:
  Given 2 plus 3 je 5
```

```
@Given ("{int} plus {int} je {int}}")
public void test.zbrajanje (int a, int b, int c){
    Assert.asserEquals(expected a+b,c);
        }
```

Slika 7.4. Prikaz provjere točnosti pisanim u Javi

Kod kolekcije parametara (engl. *Examples*) za svaki scenarij, moguće je proslijediti listu različitih parametara testa (tada scenarij mora biti ***Scenario Outline***[15]). U tom se slučaju metode testa ponavljaju za svaku proslijeđenu grupu parametara.

```
Scenario:
  Given prvi broj <a>
  Given drugi broj <b>
  When zbrajam
  Then potrebno je dobiti <c>

  Examples:
  | a | b | c |
  | 2 | 3 | 5 |
  | 4 | 3 | 7 |
  | 2 | 2 | 4 |
  | 1 | 5 | 6 |
```

```
public class Test 1 {
    int a;
    int b;
    int c;
    @Given("prvi broj {int}")
    public void prvi broj (int int1) {
        a = int1;
    }
    @Given("drugi broj {int}")
    public void drugi broj (int int1) {
        b = int1;
    }
    @When("zbrajam")
    public void zbrajam() {
        c = a + b
    }
    @Then("trebam dobiti {int}")
    public void trebam dobiti(int int1){
        Assert.assertEquals(c, int1);
        }
}
```

Slika 7.5. Slika prikaza metode koja se ponavlja za svaku proslijeđenu grupu parametra

[15] *Scenario Outline* je konstrukcija koja se koristi u okviru Gherkin sintakse za pisanje specifikacija ili testova koji koriste podatke iz tablice. *Scenario Outline* se koristi kada se isti scenarij ponavlja više puta s različitim ulaznim podacima.

7.3. Automatiziranje testiranja *web*-servisa i *REST API-ja*

Representational State Transfer ili skraćeno REST je koncept koji je zasnovan na *web*-standardima i HTTP protokolu. REST je prvi put definirao 2000. godine Roy Fielding. Centralni dio REST arhitekture zauzima resurs kojem se pristupa preko zajedničkog sučelja uz korištenje HTTP metoda. Stoga se u ovakvoj arhitekturi ne koristi XML, a postiže se identična platformska i jezična nezavisnost i otpornost na *firewall*. REST omogućava resursima da imaju različitu reprezentaciju, kao što je, naprimjer tekst, XML, JSON i tako dalje

7.3.1. REST API kao interna struktura softvera i testiranje bijele kutije

REST API je stil softverske arhitekture koji definira skup pravila i ograničenja za stvaranje *web*-usluga. REST API-ji pružaju standardni i dosljedni način za pristup podacima i funkcionalnosti u *web*-aplikaciji.

U slučaju REST API-ja, testiranje bijele kutije uključuje testiranje implementacije API-ja kako bi se osiguralo da se pridržava REST arhitektonskih ograničenja i ispravno obrađuje sve zahtjeve i odgovore.

7.3.2. HTTP protokol

Često se *web*-serveri nazivaju i aplikacijski poslužitelji, zato što su u stanju pokrenuti određene aplikacije prilikom obrade klijentskog zahtjeva. Onog trenutka kada određena strana upotrijebi neku od aplikacija (funkcionalnosti) na serveru, ona zapravo postaje *web*-aplikacija. Programski kod (program) koji se izvršava na poslužitelju prilikom aktivacije klijentskog zahtjeva naziva se **poslužiteljski kod**.

Postoji više načina izvršavanja serverskog koda, odnosno, više mogućnosti njegove implementacije. Kada *web*-server primi klijentski zahtjev, čita njegove karakteristike (sadržane u zaglavlju zahtjeva) i na osnovu njih izvršava određenu akciju. Ako zahtjev ne sadrži zahtjev za aktivacijom *web*-aplikacije, već samo potražuje neki dokument sa servera, server će pronaći dokument u sustavu datoteka, a zatim ga proslijediti klijentu kroz odgovarajući odgovor. Ovakav scenarij podrazumijeva preuzimanje statičkih sadržaja s *web*-servera (HTML dokumenata, slika i sl.).

Svaka *web*-aplikacija ima dvije strane svog izvršavanja. Jedna strana je serverska, a druga klijentska. Klijentska strana je krajnja točka izvršavanja *web*-aplikacije i ona, obično sadrži logiku koja može poslati zahtjev serveru, kao i preuzeti te adekvatno pročitati odgovor. *Web*-pretraživač (Firefox, Chrome, Edge, Opera, Safari...) najčešći je oblik

klijentske *web*-aplikacije. Korisnik *web*-aplikacije ne mora biti obavezno *web*-pretraživač, već može biti bilo koja aplikacija, sve dok u svom izvršavanju podrazumijeva i komunikaciju s *web*-serverom.

HTTP je *request-response* protokol, što znači da njegov način funkcioniranja počiva na međusobnoj razmjeni zahtjeva i odgovora između klijenta i *web*-servera. Kada klijent zatraži neki resurs, taj zahtjev obično sadrži identifikaciju resursa koji je zahtjevan, i to u formi *Uniform Resource Locatora* (URL).

HTTP komunikacija između klijenta i servera započinje HTTP zahtjevom klijenta. Zahtjev sadrži u sebi i HTTP metodu, odnosno obrazac koji *web*-server treba ispoštovati prilikom obrade zahtjeva i rukovanja resursima. Metodu bi se najbolje moglo da opisati kao najobičniju naredbu. Ako bismo npr. upisali u preglednik "*www.mysite.com/index.html"*, možemo biti sigurni da će negdje u našem zahtjevu postojati linija:

GET /index.html HTTP/1.1

Kao i linija:
host: www.mysite.com

Pri čemu je:

- *GET naziv* metode
- */index.html* adresa koja se zahtijeva
- *HTTP/1.1* verzija protokola
- *host: www.mysite.com* naziv *top-level* domene za tu *web*-stranicu.

Na osnovu ovih informacija *web*-server formira odgovor i prosljeđuje ga klijentu. Na zahtjev formatiran na spomenuti način server formira odgovor i prosljeđuje ga nazad pošiljatelju, a server u svakom trenutku zna IP adresu i *port*, na koje se od njega očekuje da prosljedi odgovor. Odgovor servera sadrži status **200** ako je sve u redu te mnoge

druge brojeve grešaka ako nešto nije kako treba. Brojevi s kojima se najčešće susreće su **404** (tražena strana ne postoji) i **500** (greška na *web*-serveru). Zapravo, statusi se po brojevima dijele na određene grupacije: 1xx za informacije, 2xx sve vrste uspješnih zahtjeva, 3xx preusmjeravanja, 4xx klijentske greške, 5xx serverske greške.

Primjer HTTP zahtjeva:

- korisnik unosi adresu:

http://www.mysite.com/index.html

- *User Agent* otvara *Socket* preko porta 80 i šalje zahtjev preko njega:
    - GET /path/file.html HTTP/1.0
      host: www.mojastranica.com
- server, nakon što prihvati zahtjev, na isti port, preko istog *Socketa*, šalje sljedeći odgovor:

```
HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 500
<html>
 <body>
some server response...
 </body>
</html>
```

Opis HTTP zahtjeva sadrži tri informacije bitne serveru:

- HTTP metoda
- putanju do traženog dokumenta
- verziju HTTP protokola

```
GET /mypage HTTP/1.1
```

Metoda je GET, traženi dokument je */mypage* a verzija protokola 1.1. To za server jednostavno znači: dostaviti datoteku *mypage.*

HTTP podrazumijeva slanje pitanja serveru i dobivanje odgovora od servera. Ova pitanja i odgovori imaju istu strukturu i zovu se HTTP poruke. Jedna HTTP poruka sastoji se od redova teksta koji se dijele u dvije grupacije: zaglavlje i tijelo.

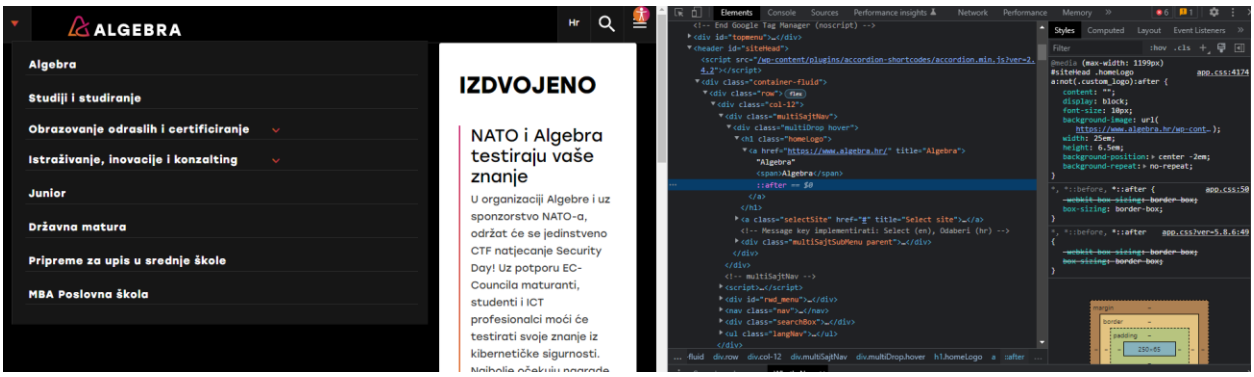
- **zaglavlje** je prvi, obavezni dio poruke
- **tijelo** sadrži ključne podatke zahtjeva, dok zaglavlja sadrže dodatne, manje važne informacije.

Provjera sadržaja putem HTTP protokola, najčešće počinje alatom ***cURL:***

```
# curl http://localhost/mypage.php
Go to <a href='testlogin.php'>Login</a>
```

Slika 7.6. cUrl putanja provjere HTTP protoka

Inspekcija HTML koda može se obaviti pomoću developerske konzole preglednika (imaju je svi suvremeniji preglednici), a najčešće se koristi *Chrome Developer tools*.

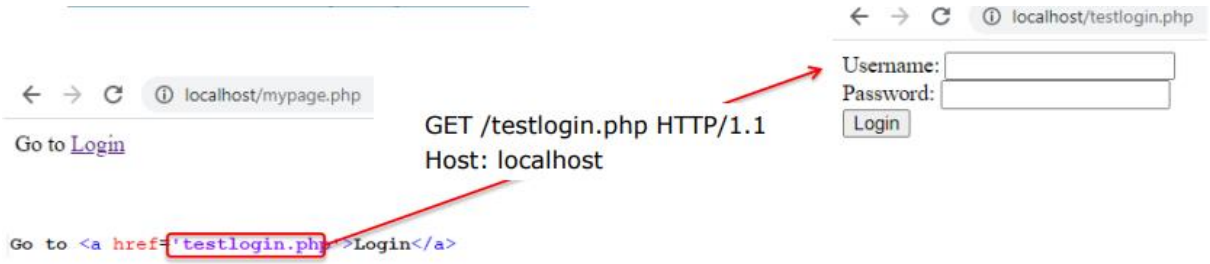


Slika 7.7. Inspekcija HTML koda pomoću Developer toola u Chrome pretraživaču

Na slici je prikazano kako smo pomoću *Developer toola* u pregledniku Chrome pregledali stranice Visokog učilišta Algebra gdje nam je prikazan HTML kod stranice, a služi nam

da vidimo elemente pisanog koda, što nam pomaže u testiranju određenih dijelova stranice.

Osnovu interakcije s korisnikom u HTML-u čine poveznice (engl. *Hyperlinks* ili *Anchor tags*). Klikom na poveznicu aktivira se otvaranje povezanog dokumenta u pregledniku.



Slika 7.8. prikaz provjere poveznica

Klik na poveznicu u HTML-u generira **GET** zahtjev.

GET zahtjev je jedna od osnovnih metoda HTTP protokola koja se koristi za dobivanje (uzimanje) resursa s *web*-poslužitelja. Za razliku od drugih HTTP metoda (kao što su POST, PUT ili DELETE), GET zahtjev nema tijelo, što znači da ne prenosi podatke klijenta na poslužitelj. Zbog toga što GET zahtjev nema tijelo i ne prenosi podatke, on se lako može izvršiti na više načina. Na primjer, GET zahtjev može se izvršiti putem URL-a u *web*-pregledniku, u skriptama JavaScripta ili drugim programskim jezicima koji imaju mogućnost slanja HTTP zahtjeva.

```
# curl http://localhost/mypage.php
Go to <a href='testlogin.php'>Login</a>
```

```
import requests

res = requests.get("http://localhost/mypage.php")
print("Status:",res.status_code)
for h,v in res.headers.items():
    print(h,":",v)
print("Page content:")
print(res.text)
```

Slika 7.9. Generiranje GET zahtjeva

Druga vrsta interakcije s korisnikom su forme, a ciljana akcija je definirana u okviru ***action atributa***. *Atribut method* određuje tip HTTP zahtjeva. Svi elementi s atributom engl. *Name*, bit će pridruženi u vidu HTTP parametara.

Slika 7.10. Generiranje POST zahtjeva

Forme generiraju POST zahtjeve i oni se nešto teže kreiraju jer sadrže tijelo i zahtijevaju specifična zaglavlja.

```
# curl -X POST -d "username=dovla&password=123" http://localhost/login.php
Hello dovla
```

```
res = requests.post("http://localhost/login.php",
    data="username=dovla&password=123",
    headers={"Content-Type":"application/x-www-form-urlencoded"}
)
print("Status:",res.status_code)
for h,v in res.headers.items():
    print(h,":",v)
print("Page content:")
print(res.text)
```

Slika 7.11. Generiranje POST zahtjeva preko cUrl putanje

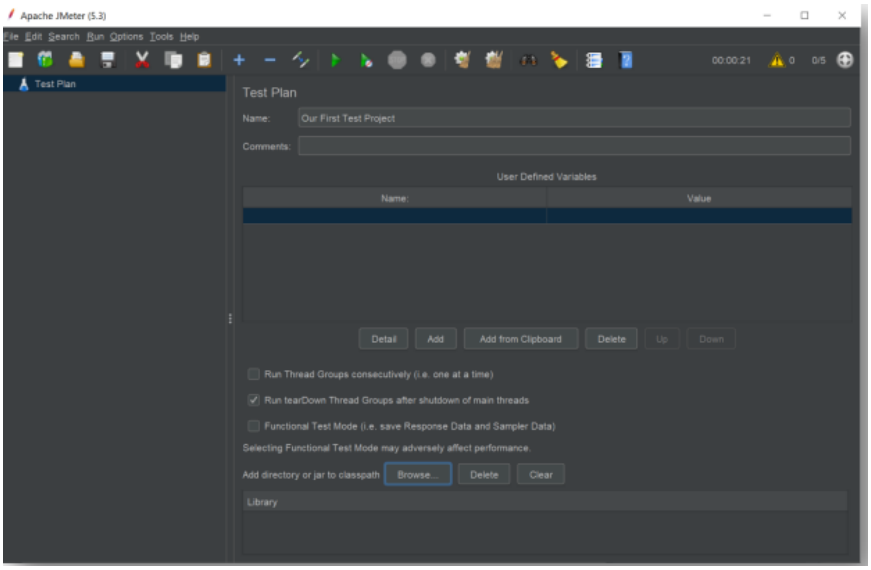
7.3.3. Postavke alata Apache JMeter

Apache JMeter je alat za automatsko testiranje opterećenosti i performanse. JMeter omogućava testiranje na različitim protokolima, a izvršava se na platformi Java. Može se koristiti u konzolnom ili grafičkom režimu.

Neke glavne postavke kod kreiranja plana testiranja su:

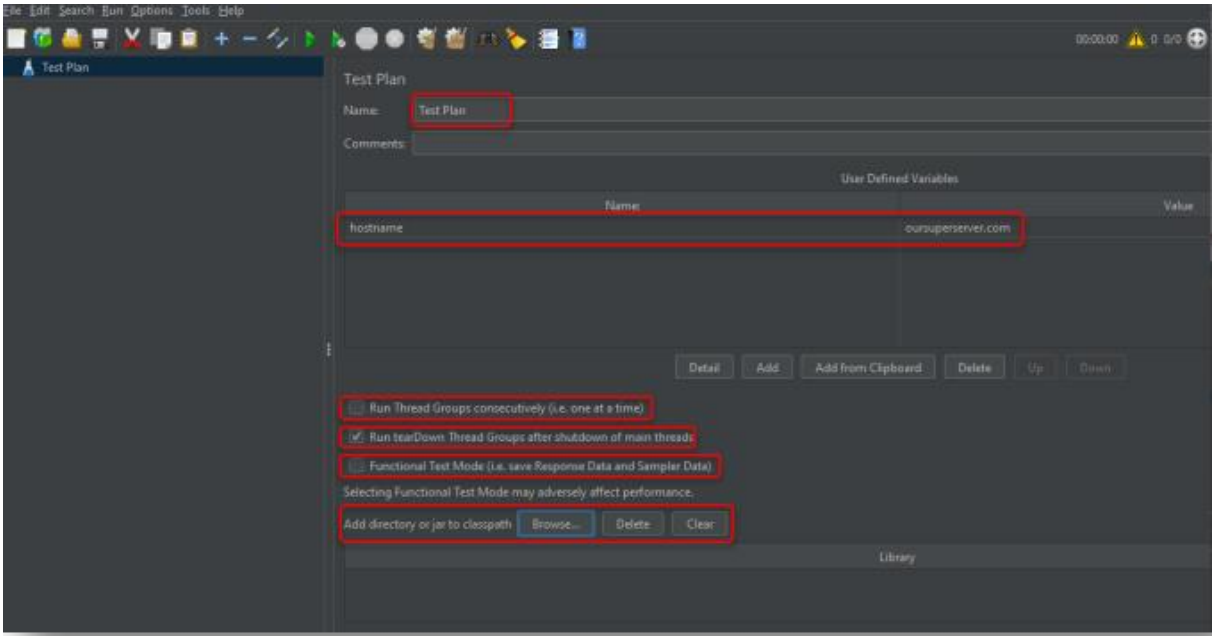
1. TEST PLAN

Da bi test JMeter mogao biti pokrenut, on mora postojati u sustavu datoteka u formi plana za testiranje. Testni plan (engl. *Test plan*) je JMeter projekt koji je osnovica plana za testiranje, a planovi se čuvaju u obliku datoteka s jmx ekstenzijom.



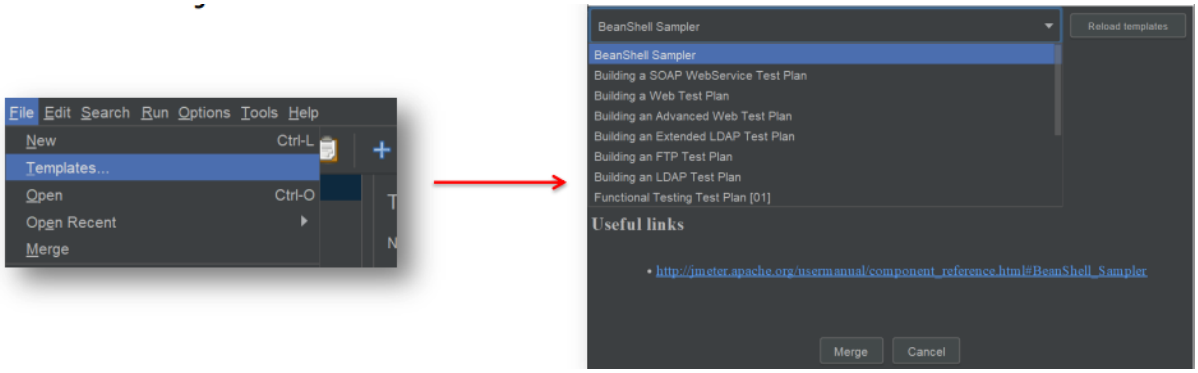
Slika.7.12. Jmeter i kreiranje test plana

Da bi se uspješno pokrenuo i izvršio test, potrebno je podesiti plan testiranja - navesti ime plana testiranja u za to namijenjeno polje, definiranje varijable i kako će se niti pokrenuti te izvršiti.



Slika 7.13. Podešavanja testnog plana u JMeter alata

U olakšavanju kreiranja testnog plana JMeter sadrži i nekoliko primjeraka testnih planova. Primjerci testnih planova pokrivaju sve standardne slučajeve korištenja.

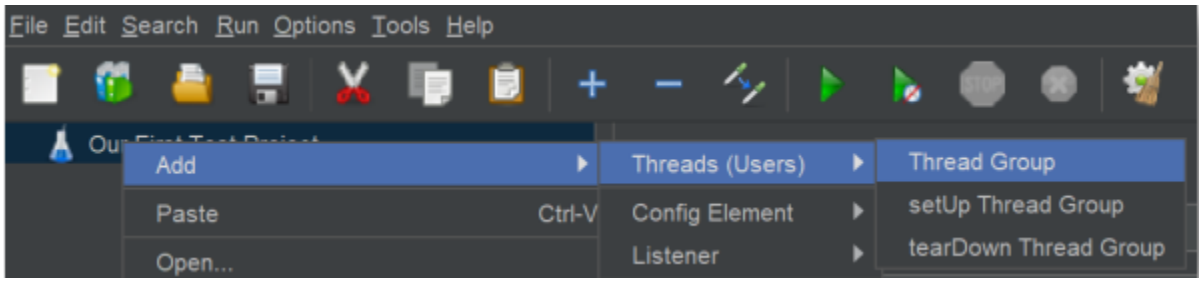


Slika.7.14. Kreiranje test plana uz pomoć predloška.

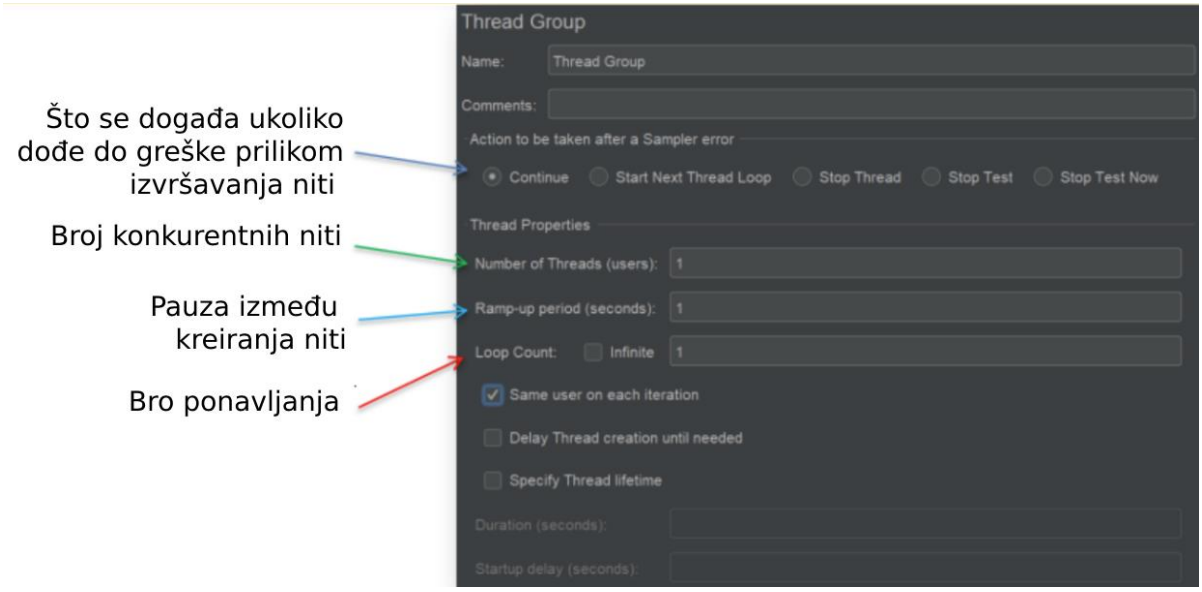
2. *THREAD* GRUPA (GRUPA NITI)

Thread grupa logička je cjelina pod kojom se izvršava test. Unutar same grupe nalaze se podešavanja o broju konkurentnih korisnika, periodu uhodavanja testa,

broju iteracija testa i slično. Osim glavne *thread* grupe, postoje i dvije dodatne koje se koriste za postavku i raspuštanje testnog okruženja.



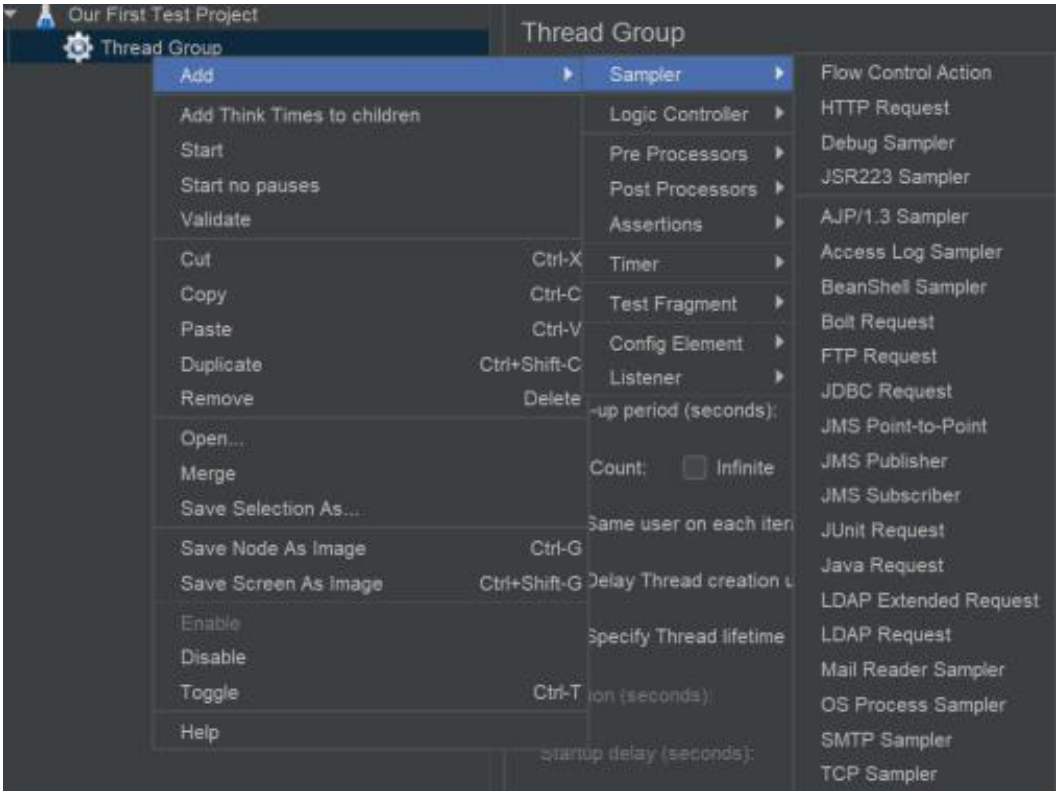
Slika 7.15. Dodavanje *Thread* grupe



Slika 7.16. Objašnjenje Thread grupa u samoj postavci

3. SAMPLERS (Uzorci)

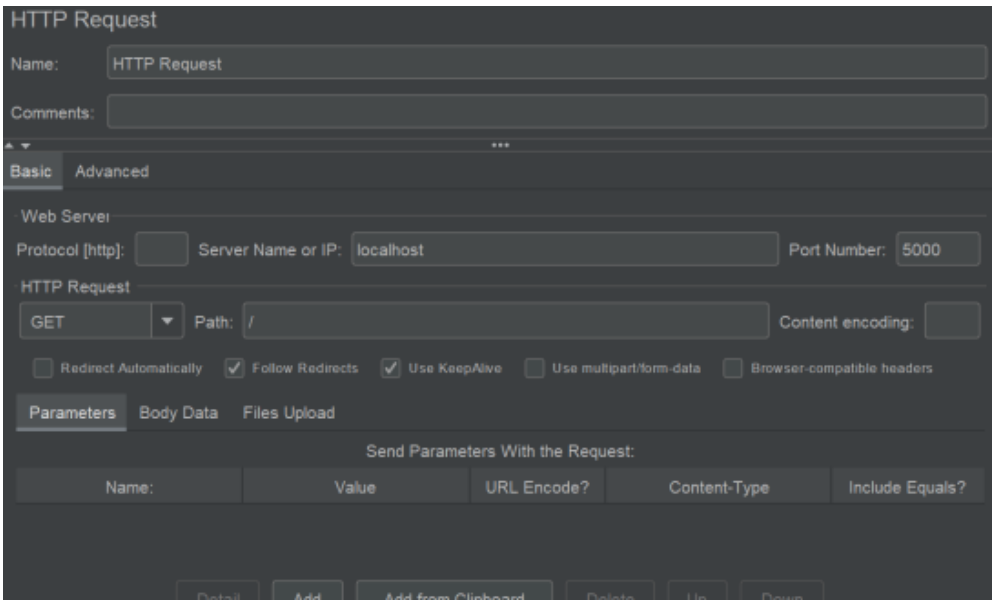
Samplers su različite konkretne operacije koje JMeter izvršava u okviru jedne grupne niti. *Samplers* su najvećim dijelom zahtjevi ili neke druge vrste mrežnih operacija. Najčešće korišteni *sampler* je HTTP *Request* koji ćemo detaljnije objasniti u nastavku priručnika.



Slika 7.17. Sampler

4. HTTP ZAHTJEVI

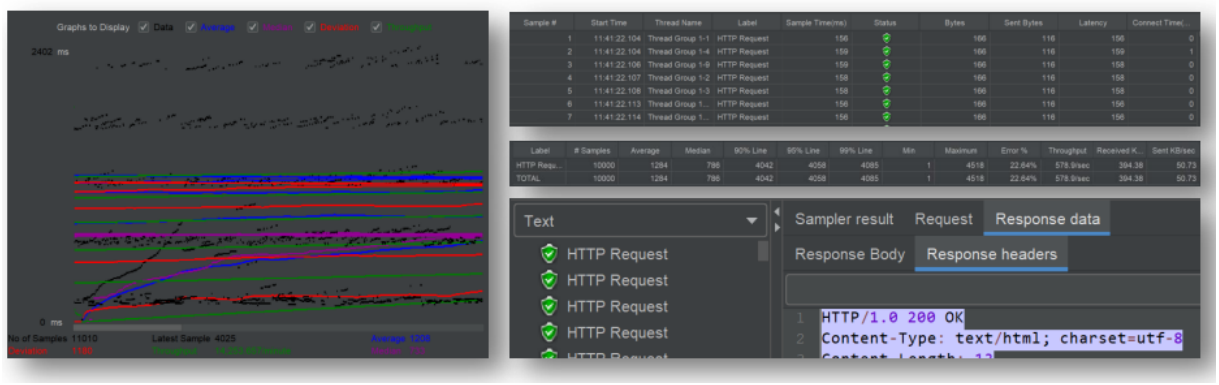
HTTP *Request* predstavlja jedan HTTP zahtjev.



Slika 7.18. Slika HTTP zahtjeva

5. SLUŠAČI

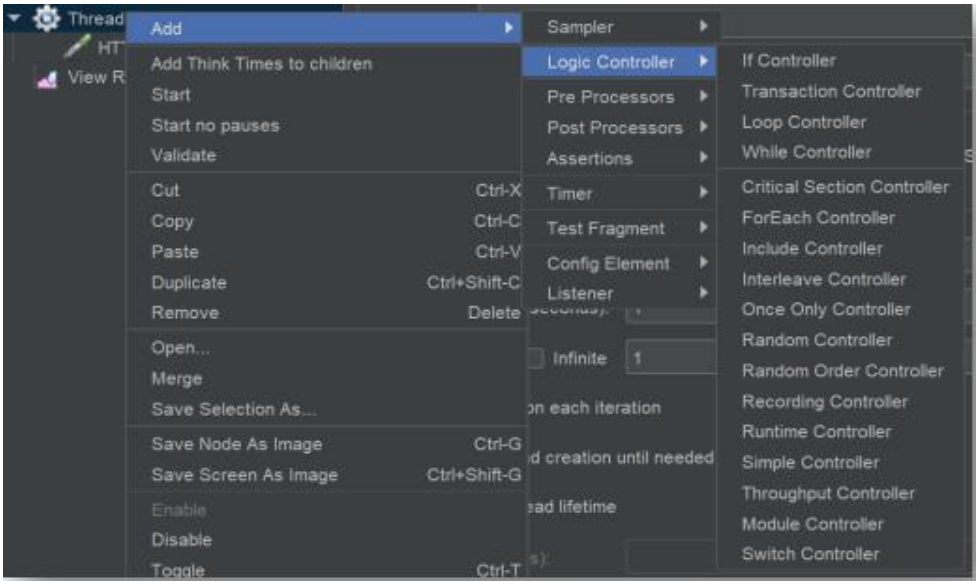
Pomoću slušača (engl. *Listeners*) materijaliziraju se rezultati izvršavanja *samplera* u okviru grupa niti. Slušači mogu vizualizirati rezultate, transformirati ih ili proslijediti drugim sustavima (sustavu datoteka, bazi podataka i slično).



Slika.7.19. Prikaz primjera slušača

6. LOGIČKI KONTROLERI

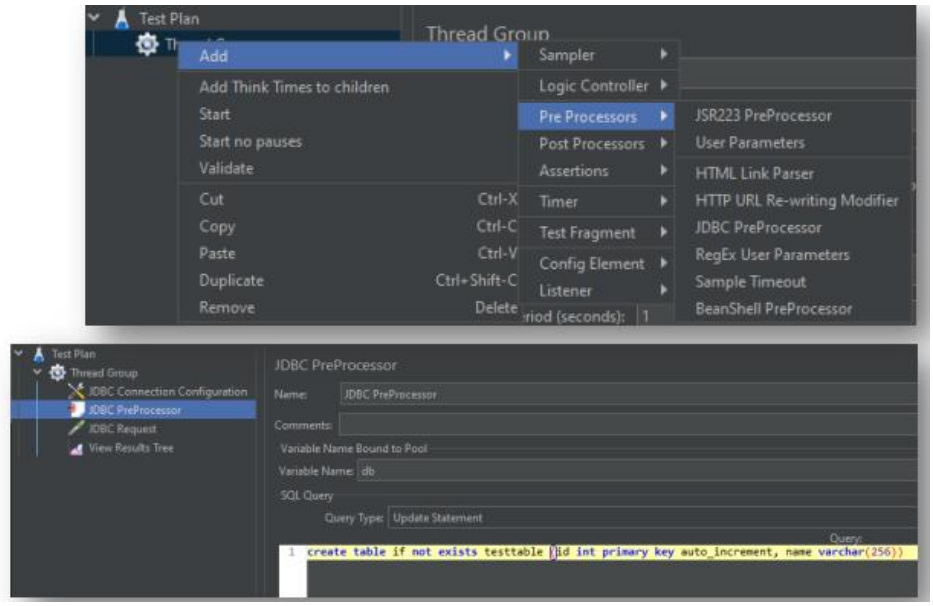
Određuju kriterije po kojima će se *sampleri* izvršavati. Operacije kontrolera slične su operacijama za kontrolu toka u programiranju.



Slika 7.20. Logički kontroleri i njegove opcije kod dodavanja u testni plan

7. PRETPROCESIRANJE

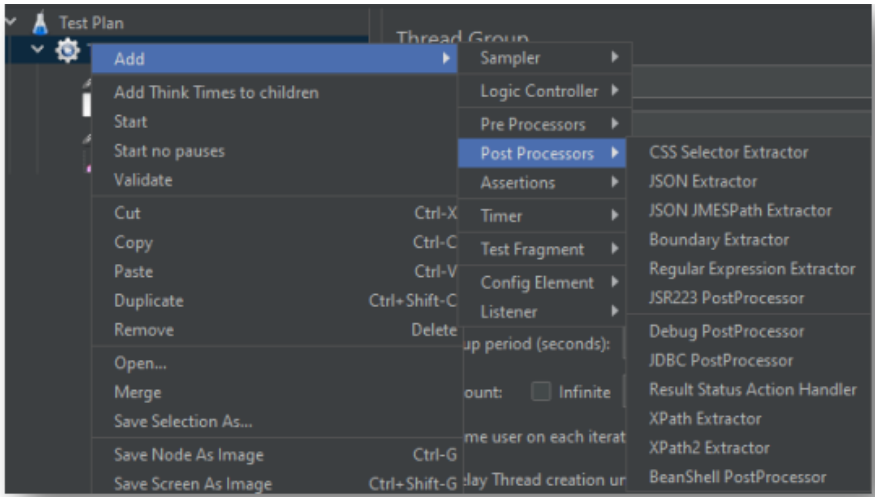
U fazi pretprocesiranja može se pripremiti okruženje za izvršavanje određenog *samplera*. Npr podaci se mogu unijeti u bazu, korisnik se može ulogirati i slično.



Slika 7.21. Faza pretprocesiranja

8. POSTPROCESIRANJE

Nakon izvršenog zahtjeva dobiju se sirovi, nesređeni podaci. Takvi se podaci pripremaju za daljnju upotrebu u fazi postprocesiranja.



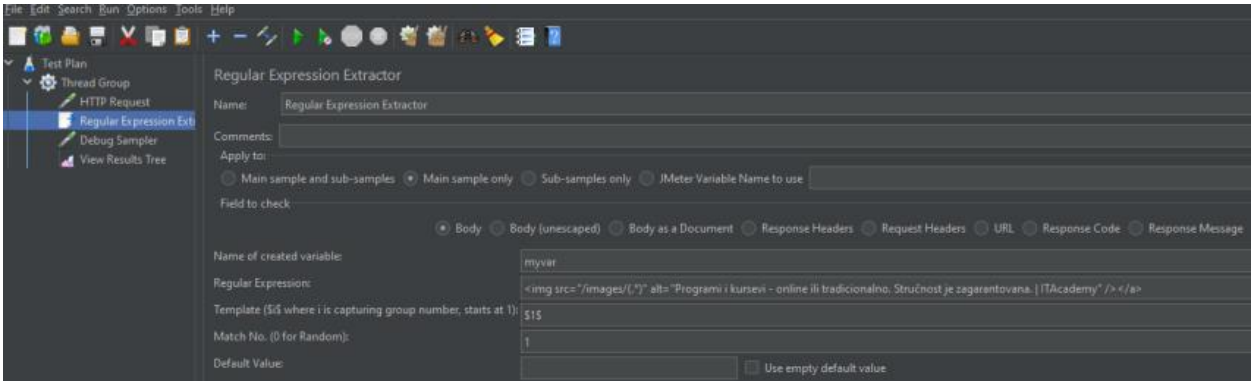
Slika 7.22. Faza postprocesiranja i njezine opcije

9. IZVLAČENJE PODATAKA ZA REZULTAT

U fazi postprocesiranja najčešće se izoliraju dijelovi rezultata koji se pretvaraju u varijable kako bi kasnije bile upotrebljive za daljnje zahtjeve ili provjeru validacije.

10. IZVLAČENJE PODATAKA REGULARNIM IZRAZOM

U JMeteru regularni izrazi koriste se za izvlačenje podataka iz odgovora koje vraća server. Mogu se koristiti za ekstrakciju vrijednosti iz odgovora koji se koriste za daljnje testiranje, kao što su identifikatori sesije, vrijednosti za zahtjev, itd. Dobivene vrijednosti mogu se koristiti u daljnjim zahtjevima putem varijabli koje su definirane u *Reference Name*[16]*.* To omogućava da automatizaciju procesa i testiranje *web*-aplikacije na način koji odgovara realnom korisničkom iskustvu.



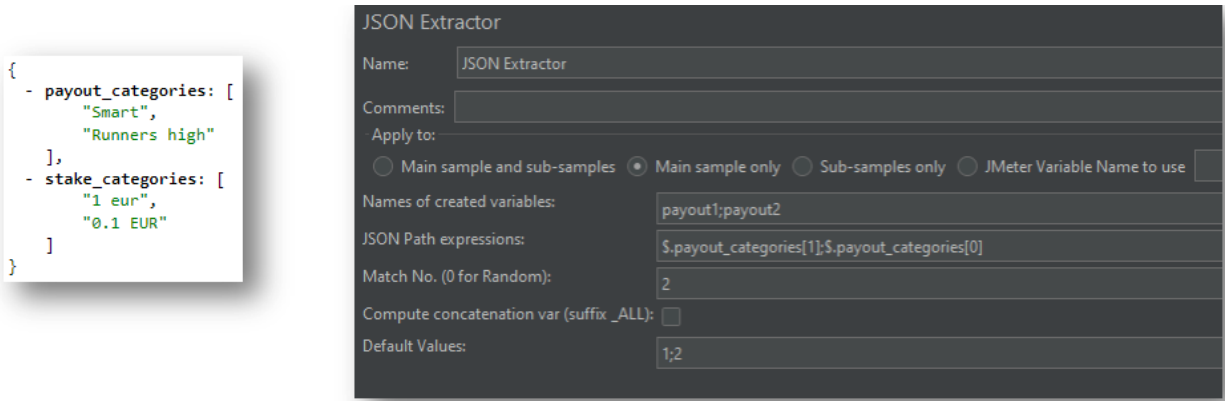
Slika 7.23. Prikaz izvlačenja podataka regularnim izrazom

11. IZVLAČENJE PODATAKA PUTEM JSON-a

U JMeteru, podaci se često izvlače iz odgovora u JSON formatu. JSON (JavaScript Object Notation) je popularan format za prenošenje podataka između klijenta i servera.

[16] Reference name u JMeteru predstavlja varijablu koja se koristi za pohranjivanje vrijednosti iz prethodnog koraka u testnom planu, kako bi se ta vrijednost mogla koristiti u kasnijim koracima testiranja.

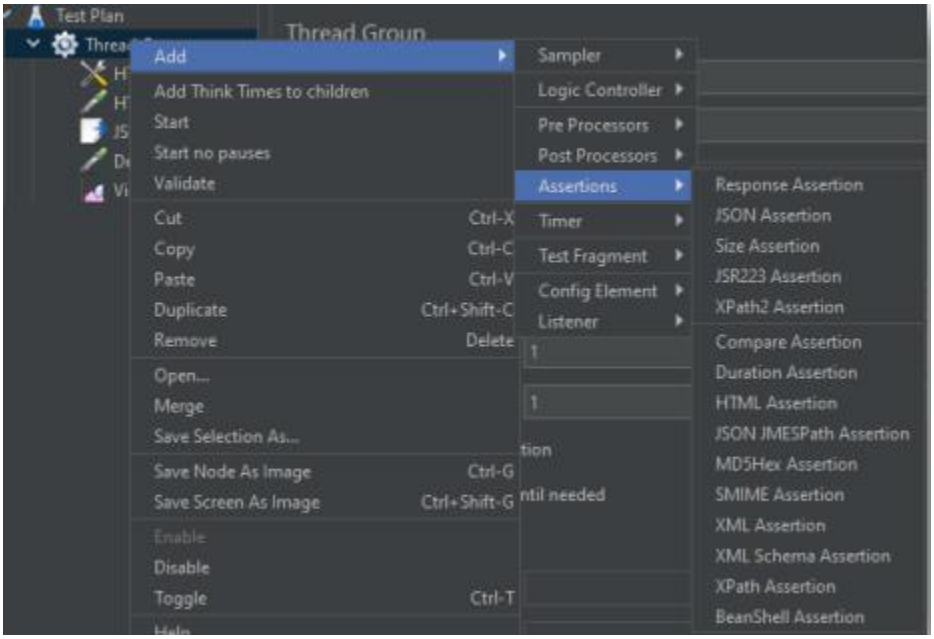
Ekstraktne vrijednosti mogu se koristiti u daljnjim zahtjevima putem varijabli koje su definirane u *Reference Name*. To omogućava automatizaciju procesa i testiranje *web*-aplikacije na način koji odgovara realnom korisničkom iskustvu.



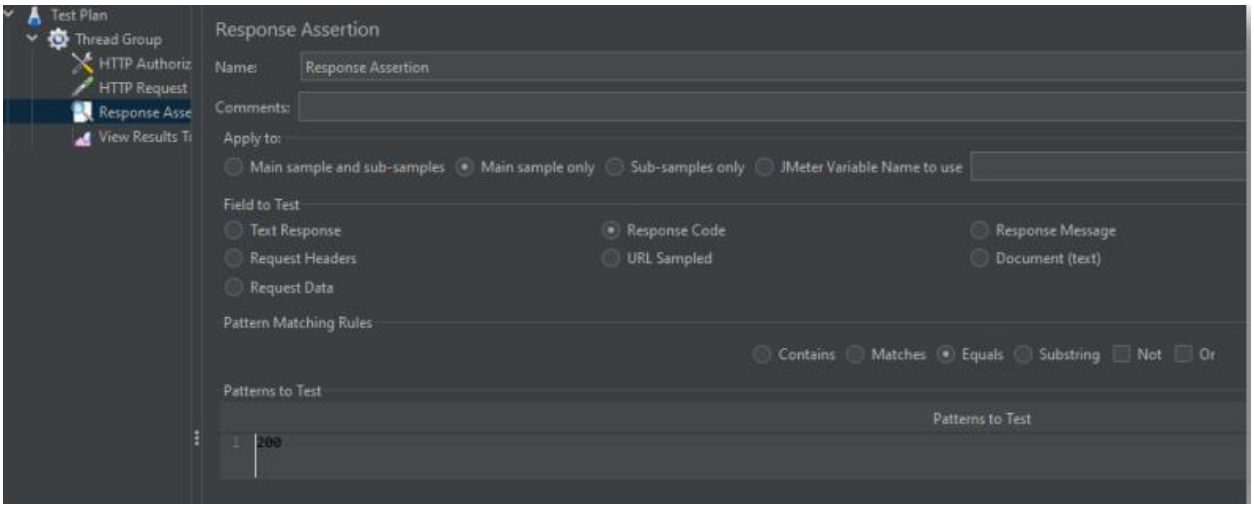
Slika 7.24. Primjer podataka izvlačenjem putem JSON-a

12. TVRDNJE

Osim provjere uspješnosti zahtjeva, također je moguće definirati tvrdnje kojima će biti izvršena korisnički definirana validacija sadržaja.

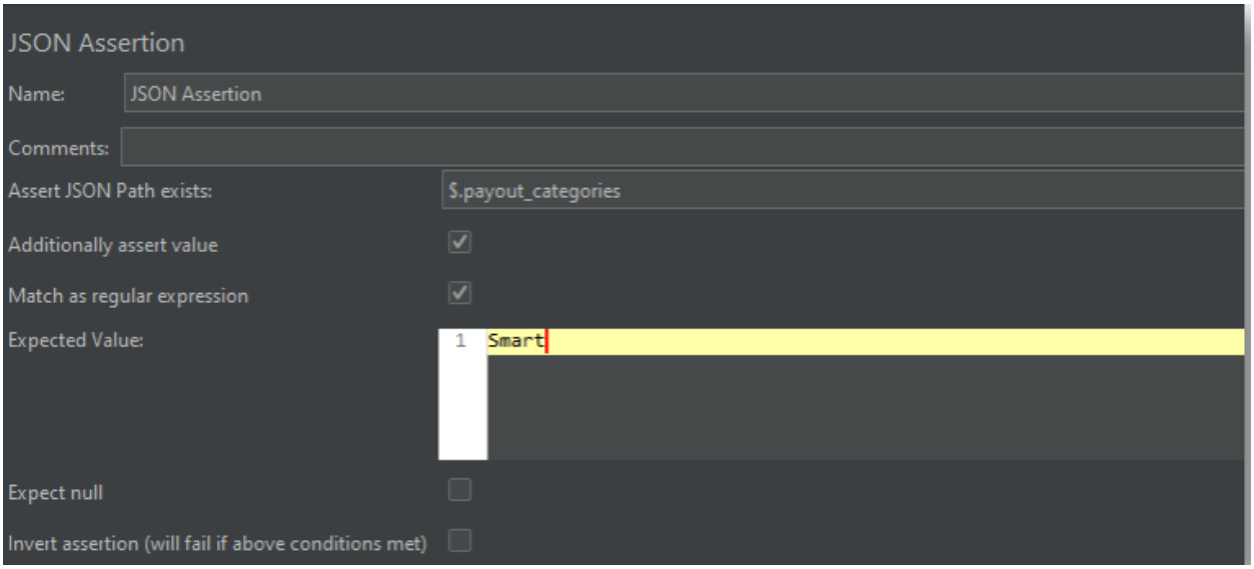


Slika 7.25. Prikaz opcija tvrdnji kojima definiramo validaciju sadržaja



Slika 7.26. Prikaz odgovora zahtjeva koji smo postavili u planu testiranja

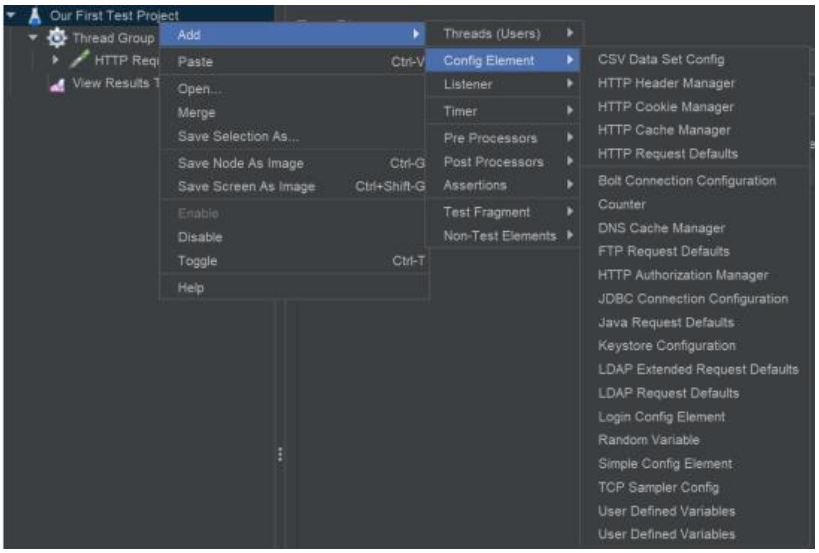
Tvrdnje JSON sadržaja koji provjerava sadržaj po JSON primjerku:



Slika 7.27. JSON sadržaj koji će biti prikazan u odgovoru

13. KONFIGURACIJE

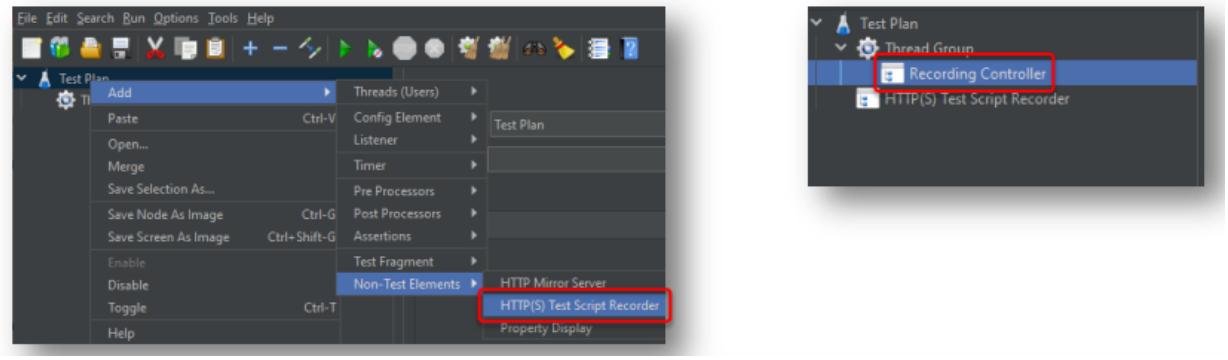
Na nivou plana testiranja, moguće je konfigurirati različite parametre koji će kasnije biti korišteni tijekom testiranja. Među podacima koji se konfiguriraju najčešće se nalaze inicijalni podaci, HTTP zaglavlja.



Slika 7.28. Opcije konfiguracije elemenata

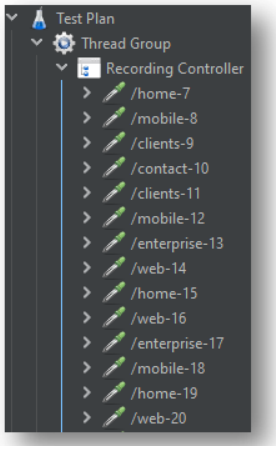
14. POSTAVLJANJE *PROXY* SERVERA

JMeter ima mogućnost aktivacije *proxy* servera[17], prilikom čega može pratiti i bilježiti promet između preglednika i servera. Na osnovu HTTP paleta, automatski se prave *request sampleri*. Da bi snimanje moglo biti obavljeno, u planu treba postojati *Recording Controller.*



Slika 7.29. Recording Controller opcija mora postojati u postavljanju proxy servera

[17] Proxy server je posrednik između klijenta (npr. *web*-preglednika) i poslužitelja (npr. *web*-servera) u mreži. Kada se klijent povezuje s poslužiteljem, umjesto da se izravno povezuje, on se povezuje s proxy serverom koji zatim prosljeđuje zahtjev poslužitelju.



Nakon ispravne postavke *proxy* servera, *Recording Controller* će se prilikom navigacije u pregledniku automatski popuniti generiranim HTTP *Request Samplerima*.

7.3.4. JSON (engl. *JavaScript Object Notation*) - format za razmjenu podataka

JSON je format za tekstualnu razmjenu podataka koji se često koristi u *web*-aplikacijama. JSON je dizajniran da bude jednostavan za čitanje i pisanje, kao i za parsiranje i generiranje podataka. Kako bi komunikacija između servera i klijenta bila uspješna, potreban je format koji će predstavljati podatke i biti lak za raščlanjivanje podataka. Potpuno je neovisan o izboru programskog jezika.

Ovaj tekstualni format idealan je za razmjenu podataka jer koristi konvencije iz programskih jezika. JSON format će podatke koje prikazuje poslagati hijerarhijski po objektima i objektnim varijablama.

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
```

```
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  }
}
```

Osnovne karakteristike JSON formata:

- minimalna veličina poruke
- podržava objekte, polja i varijable po konvencijama svih programskih jezika
- razumljiv osobama koje nemaju tehničko predznanje
- brz za raščlanjivanje zbog strukture poruke.

7.3.5. REST API – programsko sučelje mrežne aplikacije

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 s klijentskog uređaja. Neka od glavnih pravila kojih se pridržava REST koncept su:

- predmemorija servera ne koristi se za čuvanje stanja podataka
- uniformno sučelje
- odnos klijent-server u komunikaciji

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:

- *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. Pri odgovoru servera (engl*. Response*) REST API vraća poruku uz standardni HTTP kod statusa. Načinom REST komunikacije koriste se mnogi serveri.

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 ne može ispuniti validan zahtjev

7.3.6. Funkcionalno testiranje REST API-ja

Funkcionalno API testiranje važno je kako bi se osiguralo da API i aplikacije funkcioniraju prema očekivanom. Sučelje aplikacijskog programa (engl*. API - Application Program Interface*), sustav je komunikacijskih metoda koji programerima omogućava pristup programima, procedurama, funkcijama i uslugama. Najčešći protokol koji se koristi u API-ju je HTTP, s REST arhitekturom.

Primjer *web*-servisa koji ćemo prikazati funkcionalnim testiranjem bit će platforma na koju se korisnici mogu registrirati.

RUTA ZA REGISTRACIJU

- URL: https://imgybackend.herokuapp.com/auth/sign-up (upisuje se u preglednik)
- METODA: POST
- ZAGLAVLJA – *Authorization-Static* – statični token (*„imgybackend“*)

- TIJELO ZAHTJEVA (Format FORM-DATA)


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


Ako je zahtjev ispravan, uspješan odgovor servera sa statusnim kodom 200 u tijelu odgovora servera imat će pristupni token, ID korisnika i poruku o uspješnoj registraciji


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


7.4. Automatiziranje testova performansi

Automatizirano testiranje performansi proces je korištenja specijaliziranih softverskih alata za testiranje i procjenu performansi softverske aplikacije ili sustava. Ova vrsta testiranja uključuje automatiziranje izvođenja različitih testova performansi, kao što su testiranje opterećenja, testiranje preopterećenosti i testiranje izdržljivosti, kako bi se procijenilo ponašanje aplikacije u različitim uvjetima. **Cilj automatiziranog testiranja performansi je identificirati i riješiti uska grla performansi, poboljšati skalabilnost aplikacije i osigurati da može podnijeti očekivano radno opterećenje.** Automatizirano testiranje performansi omogućuje brzo testiranje performansi aplikacija, smanjujući vrijeme i napore potrebne za ručno testiranje i poboljšavajući ukupnu kvalitetu te pouzdanost softvera.

7.4.1. Testiranje performansi

REST je obrazac dizajna softvera koji se obično koristi za *web*-aplikacije. O REST-u možemo razmišljati kao o stilu softverske arhitekture. Osnovna ideja REST-a je tretiranje objekata na strani poslužitelja (npr. redaka u tablici baze podataka) kao resursa koji se mogu stvoriti ili uništiti. Kada kažemo RESTful API, mislimo na implementaciju *web*-usluga REST arhitekture.

RESTful API definira skup funkcija koje programeri mogu koristiti za izvođenje zahtjeva i primanje odgovora putem HTTP protokola, kao što su *GET, POST, PUT* i *DELETE.*

Slanje **GET** zahtjeva na /*users* dohvatilo bi korisnike s razine baze podataka.
Slanje **POST** zahtjeva na /*users* stvorilo bi novog korisnika na razini baze podataka.
Slanje **PUT** zahtjeva na /*users*/:*id* ažuriralo bi atribute određenog korisnika, ponovno identificiranog jedinstvenim ID-om.
Slanje **DELETE** zahtjeva na /*users*/:*id* izbrisalo bi danog korisnika, ponovno identificiranog jedinstvenim ID-om.

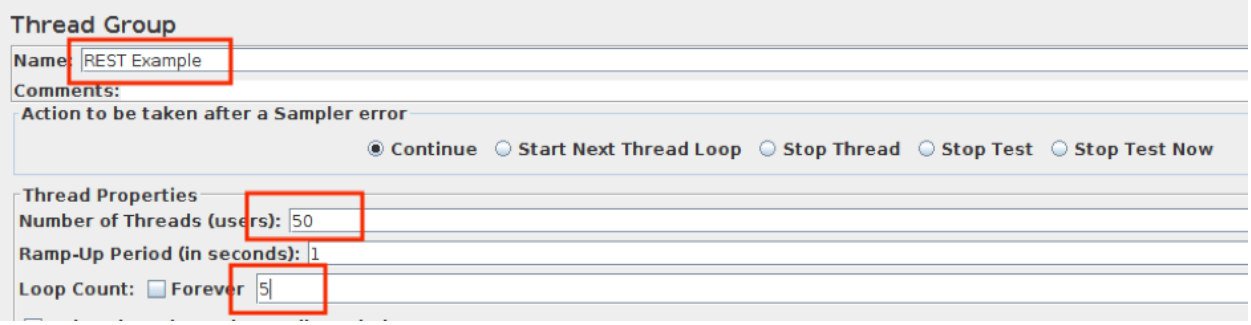
Testiranje učitavanja RESTful API-ja može se jednostavno obaviti u JMeteru pomoću HTTP *Request Samplera*, a korake koje moramo pratiti su opisani u nastavku.

Naše radno okruženje:
Ubuntu - zadnja verzija
JMeter - zadnja verzija
RESTful API dostupan na ***jsonplaceholder.typicode.com***. Vrlo je jednostavan, ali dovoljan za naš rad i za vježbanje. Ako trebate učitati test, morate dodati naziv vlastitog poslužitelja.

1. Dodajte grupu niti

Desni klik -> Dodaj-> Grupa niti

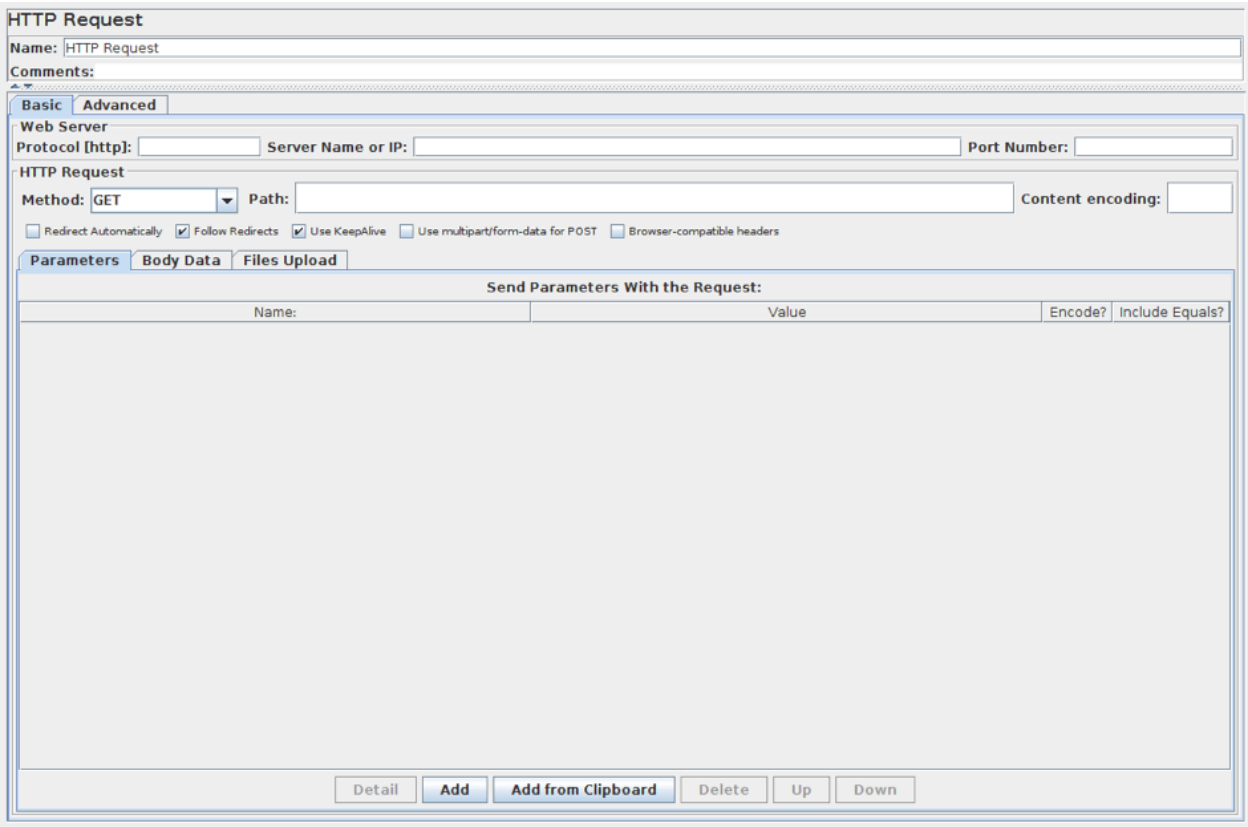
Ovdje smo definirali "Broj niti=50" i "Broj petlji=5". To nam omogućuje simulaciju 50 različitih zahtjeva 5 puta. Grupu niti možemo nazvati "REST Primjer".



Slika 7.30. Prikaz dodavanja grupa niti

2. Dodajte HTTP zahtjev

Desni klik na REST Primjer -> Dodaj -> Uzorkivač -> HTTP zahtjev.



Slika 7.31. Dodavanje HTTP zahtjeva

3. Ispunite potrebne vrijednosti:

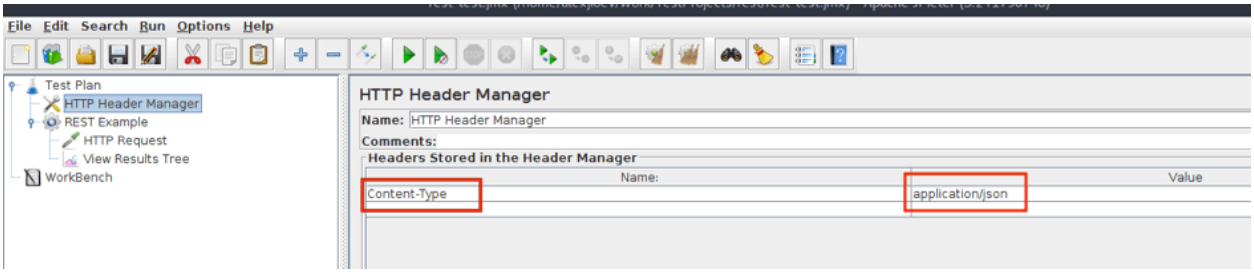
Name - naziv trenutnog uzorkivača
Protocol - prema zadanim postavkama ovo je HTTP, ali može biti i HTTPS ili FILE
Server name* ili *IP - IP adresa ili naziv domene poslužitelja. Ovo polje je obavezno
Port Number - *web*-poslužitelj koji port sluša. Zadano: 80
Method - GET, POST, PUT, DELETE, itd. Ovo polje je obavezno
Path – put do resursa. Ako ovo zahtijeva niz parametara u upitu, dodajte ih u nastavku u odjeljku "Pošalji parametre sa zahtjevom"
Content encoding - ako koristite POST, PUT, PATCH ili FILE, potrebno je koristiti kodiranje znakova.

POST kodiranje sadržaja - objavit ćemo podatke kao JSON objekt. JSON ili JavaScript Object Notation je način za pohranjivanje informacija na organiziran način.

4. Postavite zaglavlje Content-Type.

Desni klik na Test Plan -> Add -> Config element -> HTTP Header Manager

5. Dodajte polje "Content-Type" jednako "application/json", kao što je prikazano na slici ispod:



Slika 7.32. Dodavanje polja

5. Dodajte HTTP Request Sampler u grupu niti REST Primjer.

Desni klik na REST Primjer -> Add -> Rest Example -> HTTP zahtjev

6. Ispunite odgovarajuća polja:

Name - POST metoda

Server name ili IP - jsonplaceholder.typicode.com

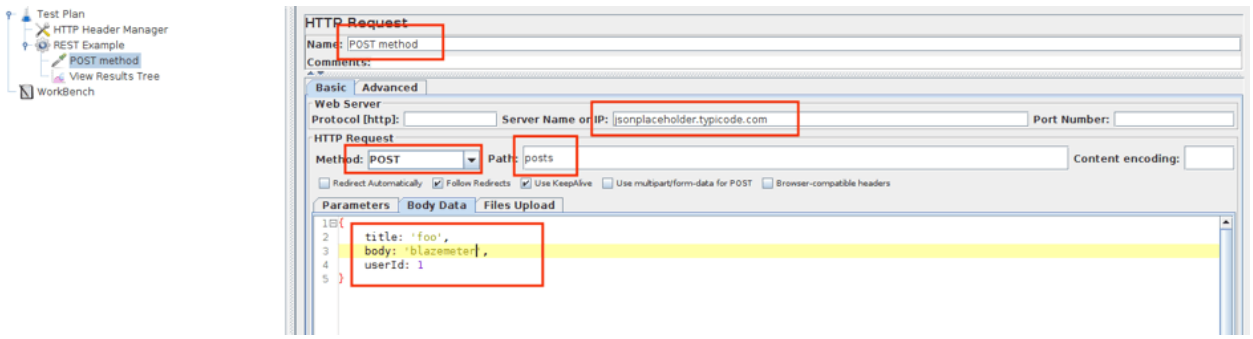
Method - POST

Path - "/articles"

Parameters-Post Body - dodajte tijelo zahtjeva

{title:'foo',body:'blazemeter',userId:1}

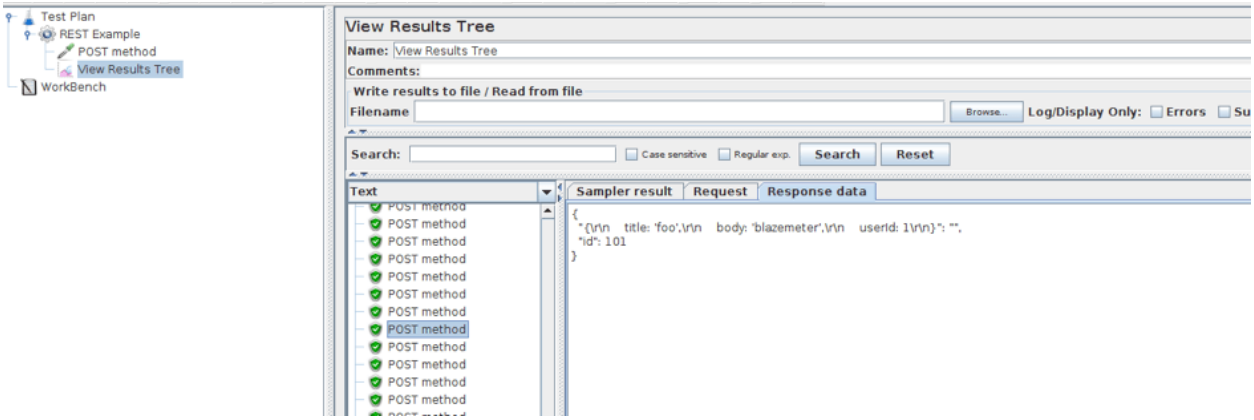
Rezultat bi trebao izgledati kako je prikazano na primjeru ispod:



Slika 7.33. Prikaz rezultata

7. Dodajte Slušatelja da vidite odgovor Zahtjeva.

Desni klik na REST Primjer -> Dodaj -> Slušatelj -> Pregled stabla rezultat. Izgledat će ovako:

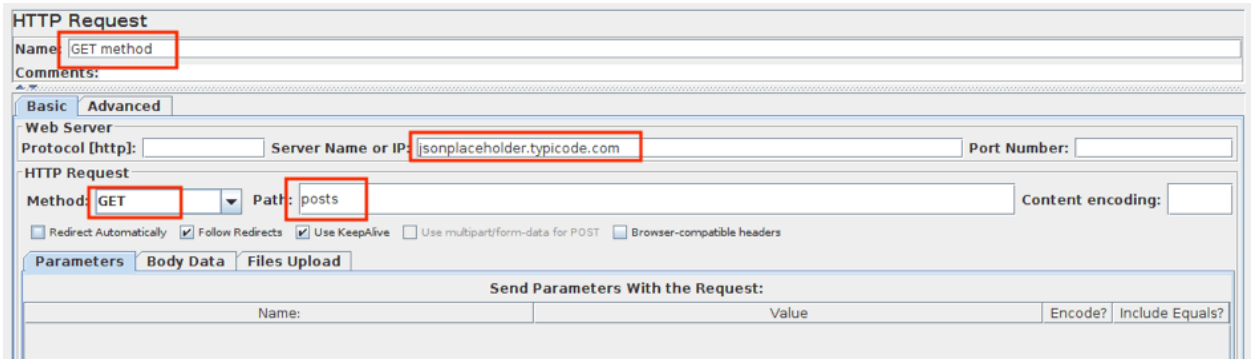


Slika 7.34. Pregled stabla rezultata

8. GET kodiranje sadržaja

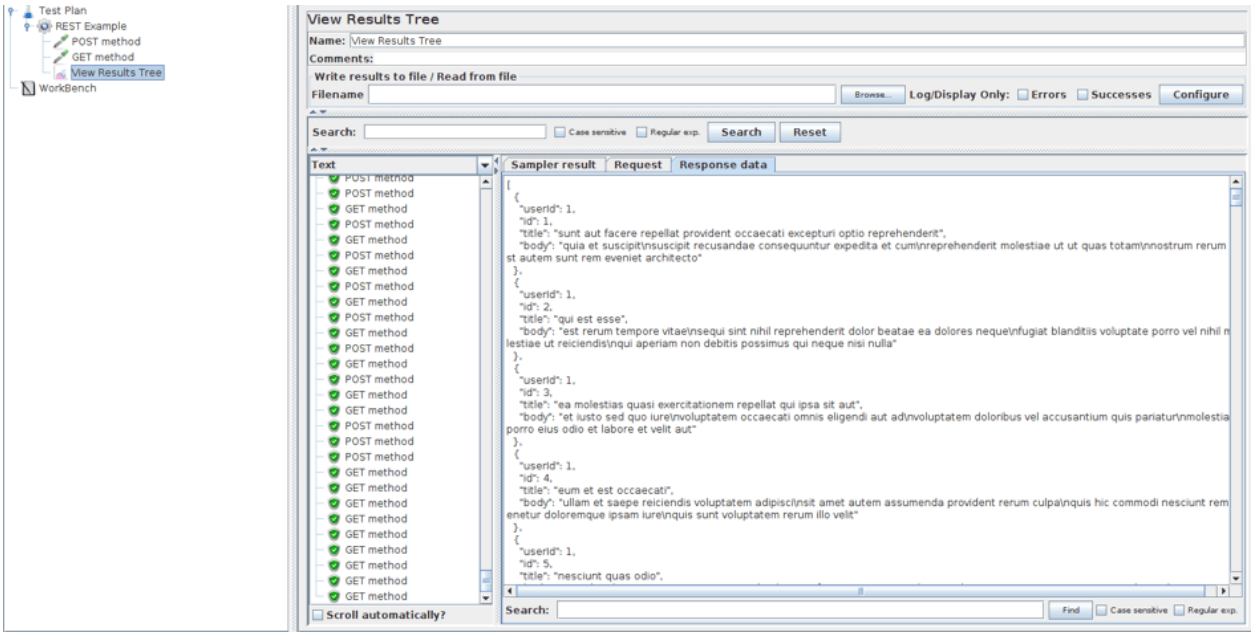
Kako bismo provjerili metodu GET, kreirajmo novi sampler HTTP zahtjeva i ispunimo sljedeće podatke:

Name - GET metoda
Server Name ili *IP* - jsonplaceholder.typicode.com
Method - DOBITI
Path - "/articles"



Slika 7.35. Kreiranje novog samplera HTTP zahtjeva

U *View Results Tree* možete vidjeti sve objave primljene metodom GET:

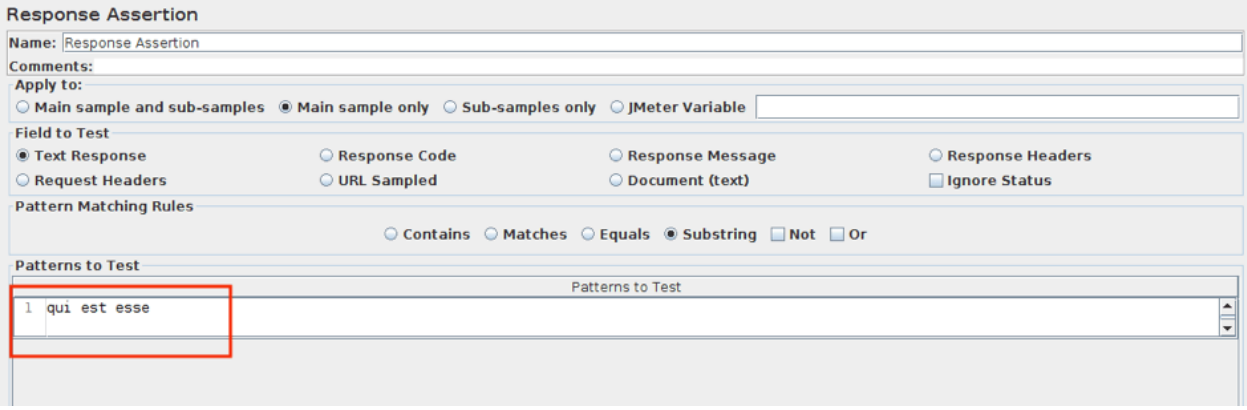


Slika 7.36. Prikaz svih primljenih metoda

10. U sklopu testiranja REST API-ja, za potvrdu da je objava uspješno predana, koristite tvrdnje odgovora (engl. *Response assertion*). Pomoću njih je moguće provjeriti je očekivani sadržaj vraćen u primljenom odgovoru na temelju definiranih kriterija.

Desni klik na metodu GET -> Add -> Assertions -> Response Assertion

11. Ostavite postavke ovog prozora kao zadane. Trebate samo konfigurirati polje "Uzorci za testiranje", kako je prikazano ispod, u skladu s odgovorom koji želite testirati.



Slika 7.38. Postavljanje uzorka za testiranje

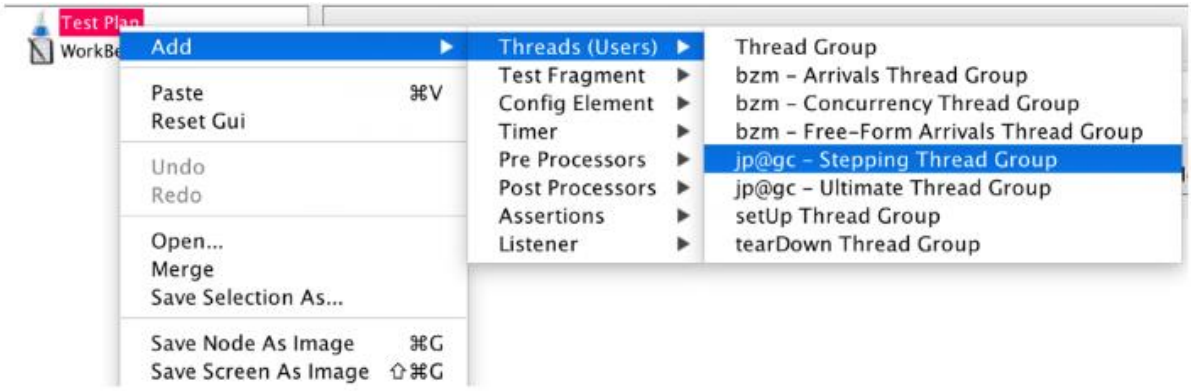
Kako funkcionira tvrdnja odgovora? U odgovorima, različiti postovi imaju različite ID-ove, naslove i tijela. Tvrdnja odgovora pregledava cijeli odgovor i traži podudaranja između obrazaca koje ste zapisali i onoga što odgovor sadrži. Ako postoji podudaranje, test će proći. Ako ne, test neće proći.

7.4.2. Testiranje opterećenosti (engl. *Load test*)

JMeter testiranje opterećenja je proces testiranja koji utvrđuje mogu li *web*-aplikacije koje se testiraju zadovoljiti zahtjeve visokog opterećenja. Tehnike koje se koriste unutar JMetera za utvrđivanje problema u sustavu uključuju grupe niti za korake i paralelnost.

1. **Grupa konačnih niti** (engl. *Stepping Thread groups*)

Najprije instalirajte *Customer Thread Groups* iz JMeter *Plugins Managera*, a nakon toga dodajte grupu koračnih niti iz plana testiranja.



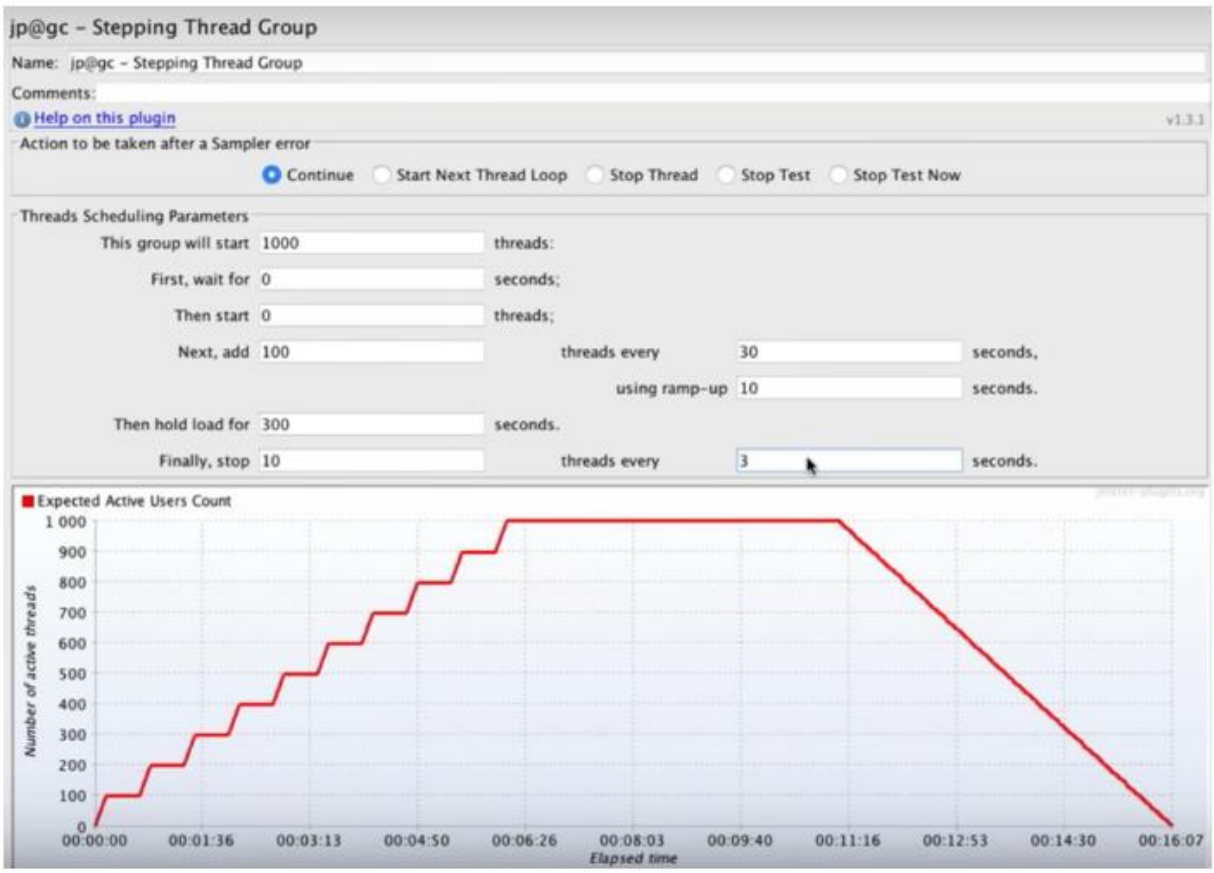
Slika 7.39. Kreiranje grupe konačnih niti

Ako pogledamo ovaj scenarij:

- 1000 niti kao ciljno opterećenje
- 0 sekundi čekanja nakon početka testa
- 0 niti se pokreće na neposrednom početku testa
- 100 niti se dodaje svakih 30 sekundi s povećanjem (ili vremenom prijelaza koraka) od 10 sekundi
- ciljno opterećenje se održava 300 sekundi (5 minuta)
- konačno, 10 niti se zaustavlja svake 3 sekunde.

Ovo bi značilo da test počinje odmah kada se JMeter pokrene, budući da se čeka 0 sekundi. Svakih 30 sekundi dodavat će se 100 korisnika, dok ne dosegnemo 1000 korisnika. Prvi korak je 1-100, drugi 101-200, itd., jer smo definirali 0 niti za pokretanje na početku. Ako smo definirali 50 niti za izvođenje, prvi korak bi bio 51 - 150, drugi 151 - 250, itd. i test bi se završio brže. Za dovršetak svakog od koraka (s po 100 korisnika) trebat će 10 sekundi. Nakon toga JMeter čeka 30 sekundi prije nego što započne sljedeći korak.

Nakon što dosegnete 1000 niti, sve će one nastaviti raditi i zajedno pogađati poslužitelj 5 minuta. Na kraju će se 10 niti zaustaviti svake 3 sekunde. Grupa konačnih niti prikazana nam je u JMeter test opterećenju u grafu pregleda u stvarnom vremenu.



Slika 7.40. Prikaz aktivnih korisnika

Promjenom vrijednosti možemo povećati opterećenje za praćenje metrike poslužitelja. Sada je sve spremno za pokretanje testa. Kako bismo vidjeli radi li sve prema očekivanjima, dodavanjem grafikona aktivnih niti možemo pratiti tijek izvedbe testa u stvarnom vremenu.

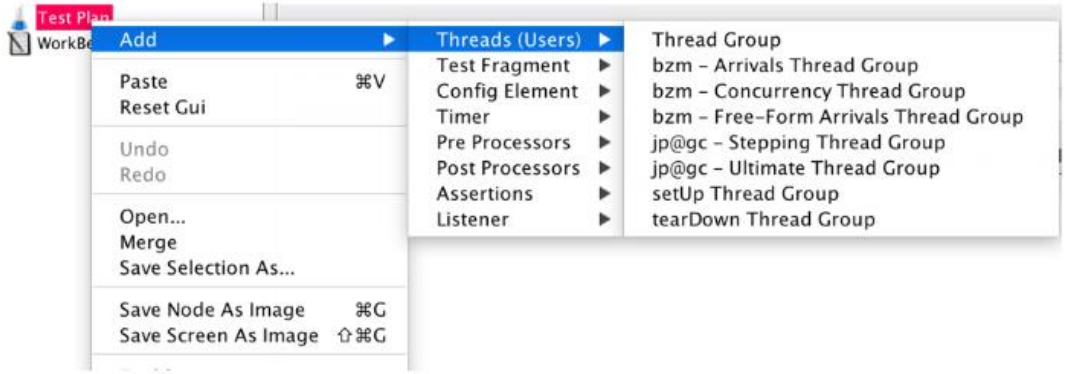
Skupina istodobnih niti pruža bolju simulaciju ponašanja korisnika jer omogućuje da lakše kontroliramo duljinu testa i stvara zamjenske niti u slučaju da nit završi usred procesa. Osim toga, grupa istodobnih niti ne stvara sve niti unaprijed čime se štedi na memoriji.

Parametri koje možemo definirati su:

- ciljana konkurentnost (broj niti)
- vrijeme pojačanja: za cijeli test

- broj koraka pri rampi
- zadrži ciljno vrijeme
- jedinica vremena: minute ili sekunde
- ograničenje ponavljanja niti (broj petlji)
- bilježenje statusa niti u datoteku: spremanje početka i zaustavljanja niti
- grupa istodobnih niti ne pruža mogućnost definiranja početnog kašnjenja, postupnog prijelaza i smanjivanja - što čini grupu konačnih niti.

U ovom koraku dodajemo grupu niti istovremenosti iz plana testiranja.



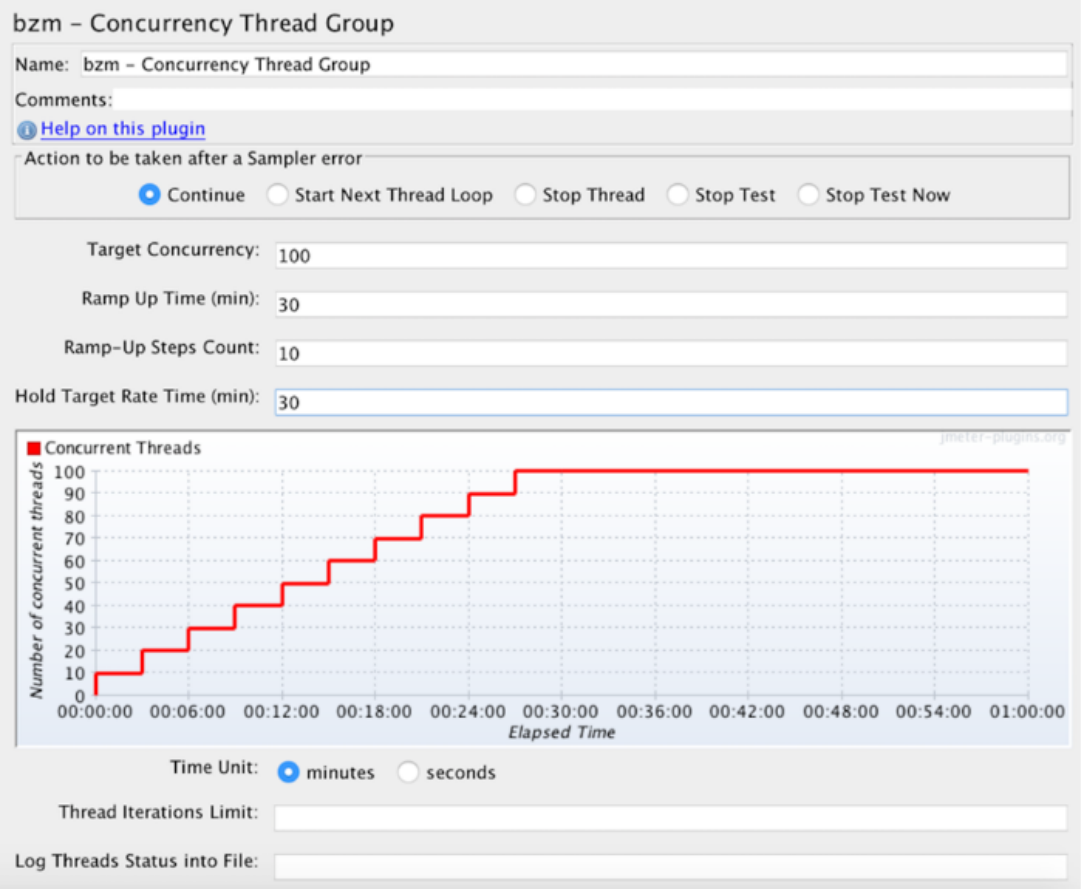
Slika 7.41. Dodavanje grupa niti istovremenosti iz plana testiranja

U ovom scenariju smo pokrili:

- 100 niti
- 30 minuta vrijeme pojačanja
- 10 koraka za povećanje
- 30 minuta držeći ciljnu brzinu

Ovo znači da će se svake 3 minute dodavati 10 korisnika dok ne dosegnemo 100 korisnika. (30 minuta podijeljeno s 10 koraka jednako je 3 minute po koraku. 100 korisnika podijeljeno s 10 koraka jednako je 10 korisnika po koraku. Ukupno: 10 korisnika svake 3 minute). Nakon što se dosegne 100 niti, sve će one nastaviti raditi i pogađati poslužitelja zajedno 30 minuta.

Grupa istodobnih niti također nam pokazuje koji je test prikazan u grafu, a pregled je prikazan u stvarnom vremenu.



Slika 7.42. Prikaz istodobnih niti u stvarnom vremenu

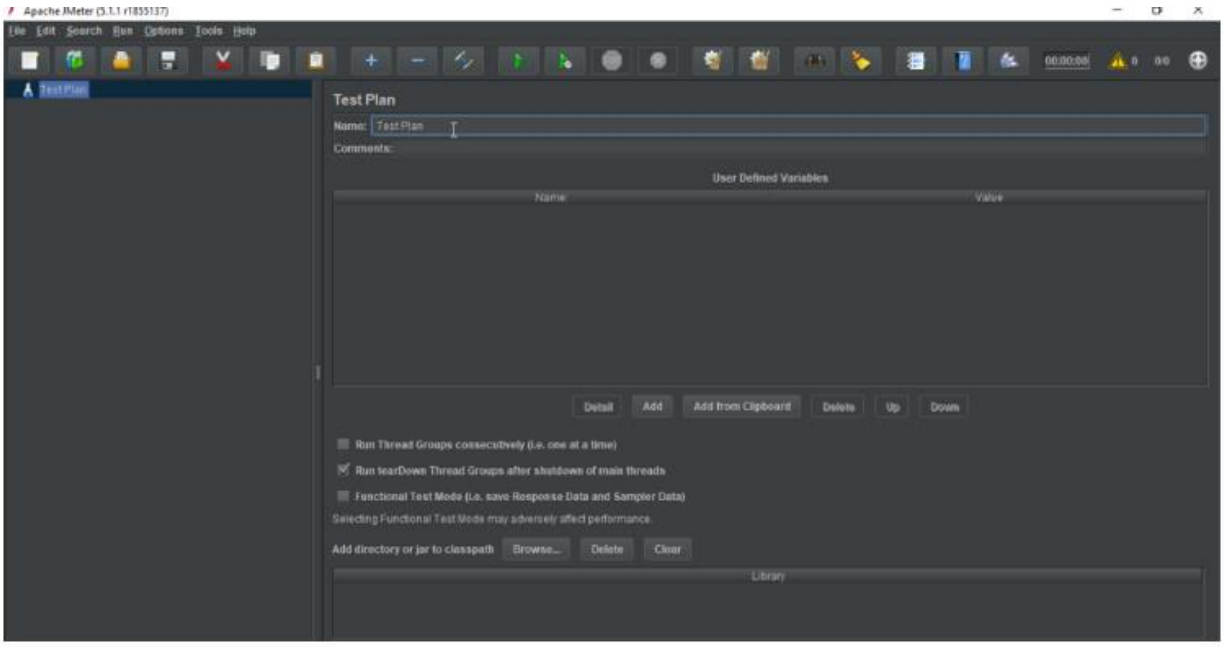
7.4.3. Testiranje preopterećenosti (engl. *Stress test*)

Važno je provesti testiranje preopterećenosti na *web*-stranici kad god dođe do skoka ili iznenadnog porasta prometa. Testiranje preopterećenosti potrebno je iz sljedećih razloga:

- za provjeru radi li sustav u nenormalnim uvjetima
- prikazuje se odgovarajuća poruka o pogrešci kada je sustav pod stresom
- kvar sustava u ekstremnim uvjetima može rezultirati ogromnim gubitkom prihoda

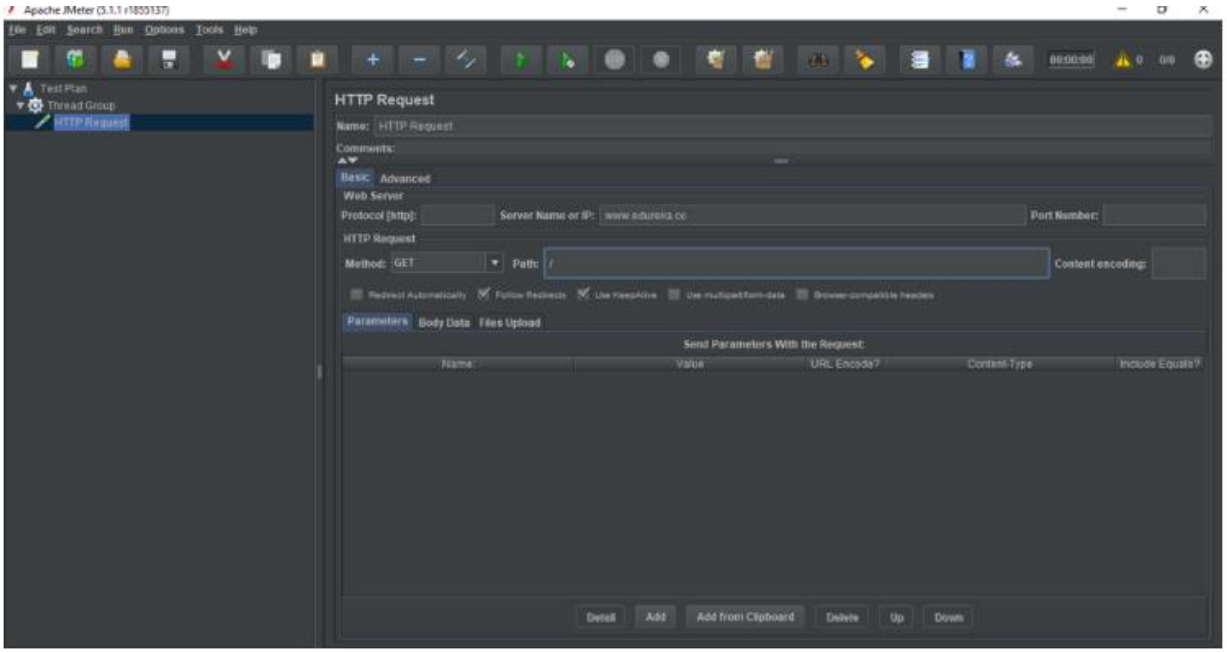
- za pripremu *web*-stranicu za ekstremne uvjete provođenjem testiranja otpornosti na stres.

1. Izrada vlastitog plana testiranja u JMeteru



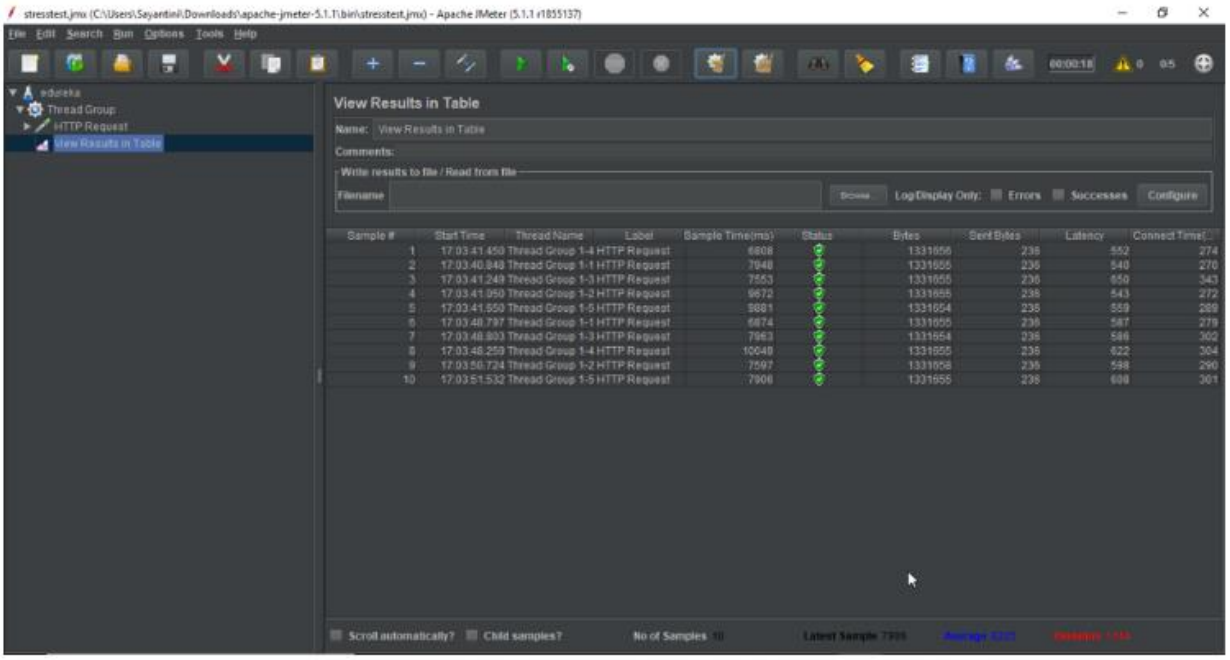
Slika 7.43. Kreiranje plana testiranja u JMeter

2. Umetnite grupu niti i dodajte HTTP zahtjev s nazivom poslužitelja *web*-stranice na kojoj želite izvršiti testiranje preopterećenosti



Slika 7.44. Dodavanje grupa niti i HTTP zahtjev

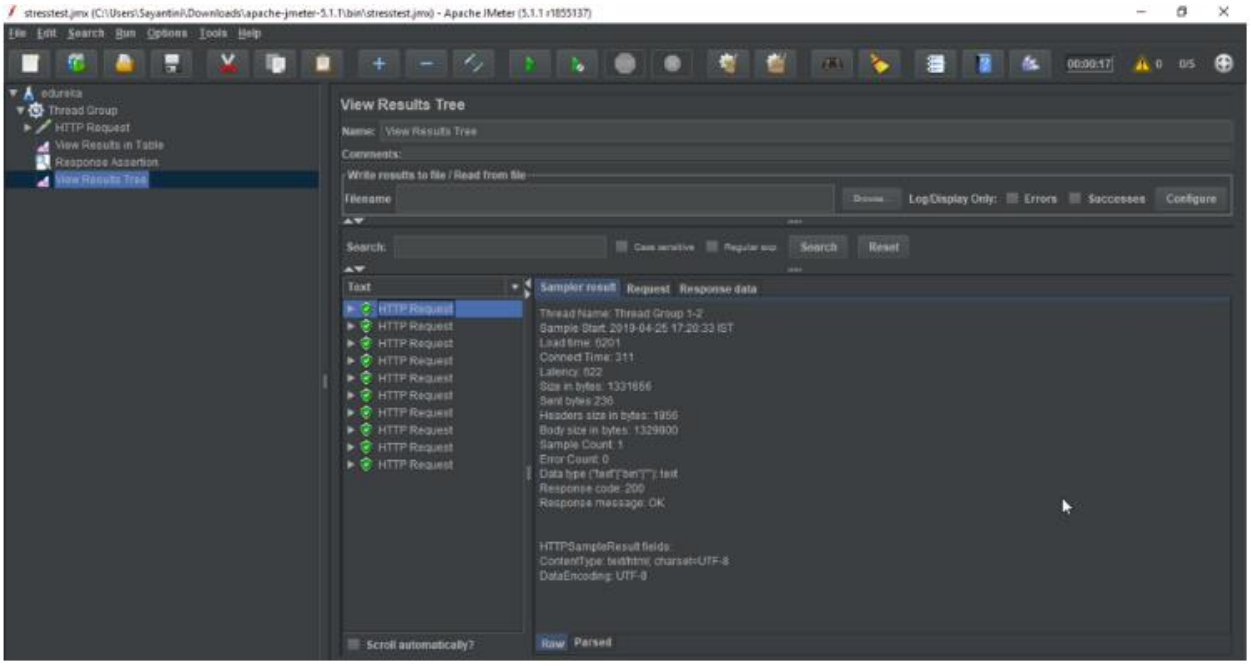
3. Zatim se moraju dodati slušatelji unutar grupe niti i pregledati rezultati testa. Prikazat će se status obavljenog testa.



Slika 7.45. Prikaz rezultata testa

4. Sada se dodaje tvrdnja odgovora (engl. *Assertions*) unutar grupe niti. To je jedan od najvažnijih elemenata jer pomaže potvrditi odgovor na zahtjev u planu testiranja preopterećenosti softvera.

U ovom planu testiranja utvrdit ćemo zahtjev odgovora 200 pomoću tvrdnje odgovora. Kōd odgovora 200 predstavlja uspjeh HTTP zahtjeva. Ako se promijeni vrijednost s 200 na bilo koji drugi broj, status testa će se promijeniti u neuspješan.



Slika 7.46. Slika odgovora kod HTTP zahtjeva

Kako biste povećali preopterećenje na poslužitelju, sada se može dodati određeni broj korisnika ili niti te povećati broj petlji.

7.4.4. Nefunkcionalni zahtjevi za performanse aplikacije

Nefunkcionalni zahtjevi su ciljevi testiranja stvoreni posebno za testiranje performansi, testiranje sigurnosti, testiranje upotrebljivosti itd. To je kombinirani zahtjev za sve vrste nefunkcionalnih testova.

Nefunkcionalni zahtjevi u testiranju performansi uključuju:

- **vrijeme odgovora**: vrijeme koje je potrebno sustavu da odgovori na zahtjeve korisnika
- **propusnost**: brzina kojom sustav može obraditi zahtjeve
- **iskorištenost resursa**: količina resursa sustava koje koristi sustav
- **skalabilnost**: sposobnost sustava da se poveća ili smanji kada je to potrebno
- **pouzdanost**: sposobnost sustava da dosljedno daje točne rezultate
- **sigurnost**: sposobnost sustava da ostane siguran od napada
- **dostupnost**: sposobnost sustava da ostane dostupan kada je to potrebno.

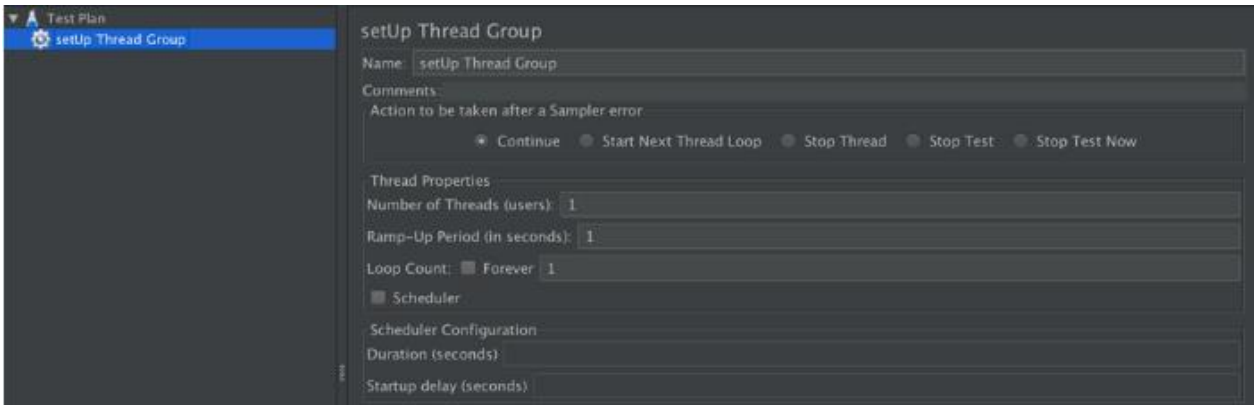
7.4.5. Definiranje testne grupe (engl. *Threads*) za simuliranje korisnika u alatu JMeter

JMeter je alat za testiranje performansi koji omogućava testerima da simuliraju veliki broj korisnika koji pristupaju aplikaciji istovremeno. Ova funkcionalnost se ostvaruje korištenjem testnih grupa ili *threadova*. Testna grupa u JMeteru predstavlja jednog virtualnog korisnika koji pristupa aplikaciji i izvodi određene radnje, poput preuzimanja stranice ili slanja zahtjeva. Broj testnih grupa koje se koriste u JMeteru određuje broj istovremenih korisnika koji simuliraju pristup aplikaciji.

Podešavanje testne grupe u JMeteru omogućava kontrolu broja korisnika koji simuliraju pristup aplikaciji, što daje precizniji uvid u performanse aplikacije pod opterećenjem. To je važno za procjenu performansi aplikacije i identifikaciju mogućih problema koji bi se mogli javiti kod velikog broja stvarnih korisnika. Prateći korake koje ćemo navesti u nastavku, definirat ćemo testne grupe za simuliranje korisnika.

1. **Priprema grupe niti**

Grupe niti (*threads*) jedna su od ključnih funkcionalnosti u JMeteru koje se koriste za simuliranje više korisnika koji istovremeno pristupaju aplikaciji.



Slika 7.47. JMeter će automatski pokrenuti određeni broj niti i simulirati korisnike koji pristupaju aplikaciji istovremeno (u ovom slučaju jedan korisnik)

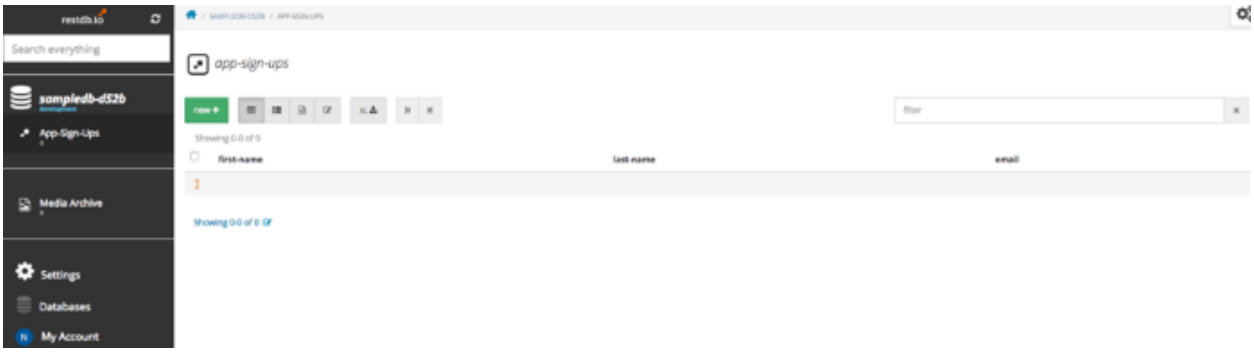
2. Kreiranje skripte

Definirajmo sljedeći scenarij: Želimo testirati 10 korisnika koji će se prijaviti na *web*-stranicu i kupiti odjeću. Prije izvođenja testa potrebno je kreirati korisnike. To se može učiniti pomoću ***SetUp Thread Group***. *SetUp Thread Group* je jedna od vrsta *thread* grupa (skupina niti) koje se koriste u JMeteru. Ova se *thread* grupa koristi za konfiguraciju određenih parametara prije nego se izvrši glavni dio testa i obično se koristi za postavljanje podataka ili izvođenje drugih operacija koje su potrebne za pripremu testa. Na primjer, može se koristiti za stvaranje korisničkih računa, započinjanje sesija ili postavljanje početnih vrijednosti varijabli. *SetUp Thread Group* se izvršava samo jednom prije početka glavne *thread* grupe koja izvodi stvarni test. To osigurava da su svi potrebni parametri postavljeni prije nego se započne glavni test i pomaže u osiguravanju pouzdanosti i točnosti rezultata testa. *SetUp Thread Group* se konfigurira korištenjem istih parametara kao i druge vrste *thread* grupa, kao što su broj niti (*threads*), broj ponavljanja (*loop count*), kašnjenje (*ramp-up*), trajanje testa (*duration*), itd.

Ova *thread* grupa dodaje se u testni plan kao poseban element, a parametri se konfiguriraju putem korisničkog sučelja ili datoteke konfiguracije.

3. Konfiguriranje baze podataka

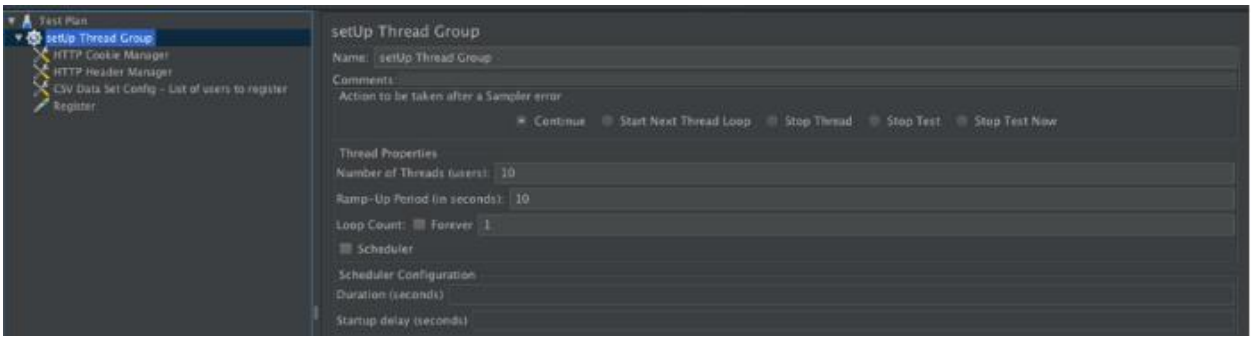
Stvorili smo zbirku baze podataka pod nazivom "prijave za aplikacije" (koristit će besplatnu probnu verziju ***restdb.io***). Konfigurirana su 3 polja: **“ime**”, **prezime”** i **“e-mail”**. Trenutno zbirka nema unosa.



Slika 7.48. Konfiguriranje baze podataka

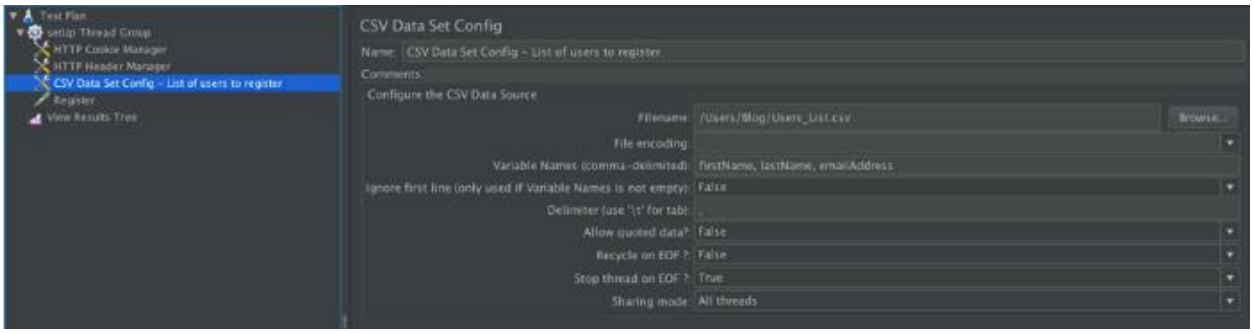
4. Konfiguriranje naše skripte

Najprije ćemo trebati izraditi testni plan koji sadrži *SetUp Thread Group.*



Slika 7.49. Konfiguriranje skripte koja sadrži SetUp Thread Group

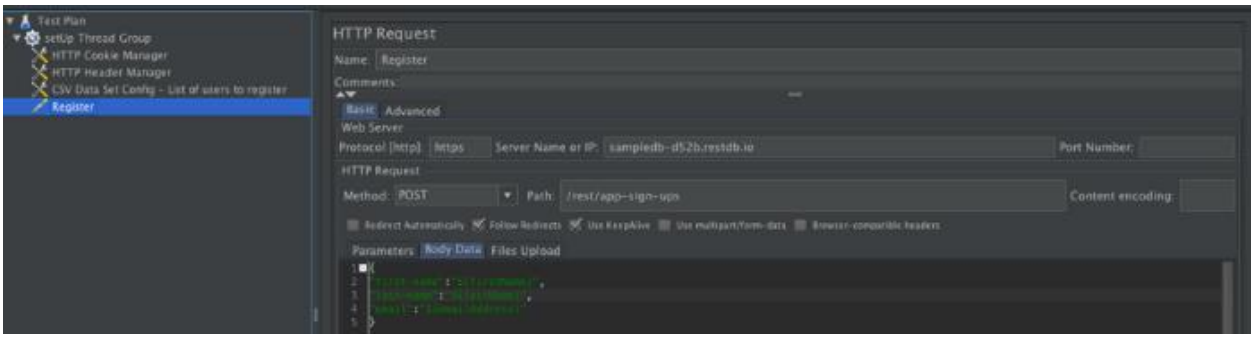
SetUp Thread Group sastoji se od *CSV Data Set Config*[18] s referencom na CSV datoteku[19] koja sadrži popis korisnika za registraciju (ime, prezime i adresa e-pošte).



Slika 7.50. Konfiguriranje opcije CSV Data Set Config koja će imati CSV datoteku

U ovom primjeru šaljemo zahtjev za registraciju s 3 varijable: **imeKorisnika**, **prezimeKorisnika** i **e-mailKorisnika**. JMeter će analizirati CSV datoteku i popuniti te varijable s vrijednostima koje odgovaraju podacima iz CSV datoteke.

Snimak zaslona CSV datoteke koja sadrži podatke o 10 korisnika:



Slika 7.51. Snimak zaslona CSV datoteke koja sadrži podatke o korisnicima

Sada će se konfigurirati zahtjev za registraciju s POST metodom koja će pokrenuti API poziv za dodavanje novog zapisa u kolekciju (registraciju korisnika). Koristit će se sljedeći API poziv:

[18]CSV Data Set Config je element u JMeteru koji se koristi za čitanje podataka iz CSV datoteke i upotrebu tih podataka u testiranju performansi *web*-aplikacija. CSV Data Set Config omogućuje JMeteru da čita podatke iz vanjske datoteke i koristi ih za generiranje zahtjeva tijekom testiranja.

[19]CSV datoteka (*Comma Separated Values*) je tekstualna datoteka koja se koristi za pohranu tabličnih podataka u obliku vrijednosti razdvojenih zarezom. Svaka linija u CSV datoteci predstavlja jedan redak tablice, a vrijednosti su odvojene zarezom ili drugim znakom koji se koristi kao separator.

POST https://sampledb-d52b.restdb.io/rest/app-sign-ups

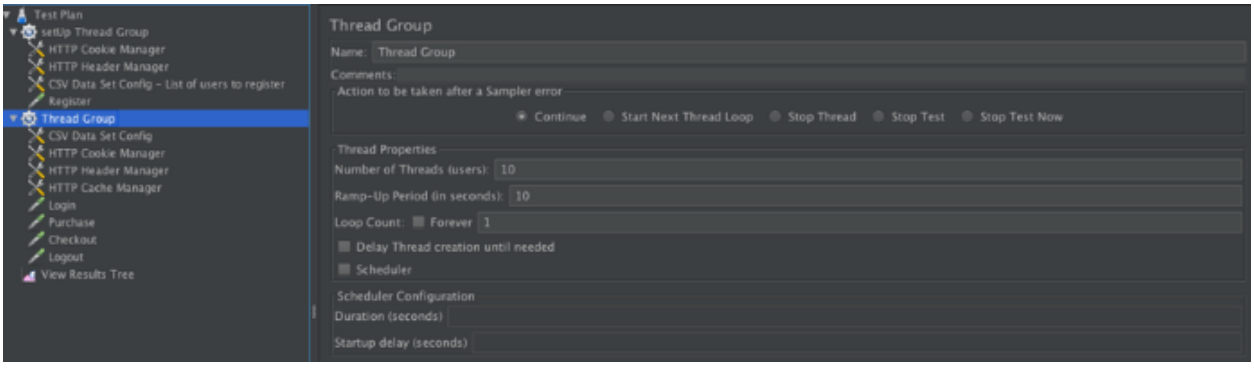
I podaci koji će se poslati u tijelu zahtjeva:

```
{
  "ime":"${imeKorisnika}",
  "prezime":"${prezimeKorisnika}",
  "e-mail":"${emailKorisnika}"
}
```

Moramo biti sigurni da koristimo iste nazive varijabli koje smo definirali u CSV-u i ista polja koja smo definirali u bazi podataka.

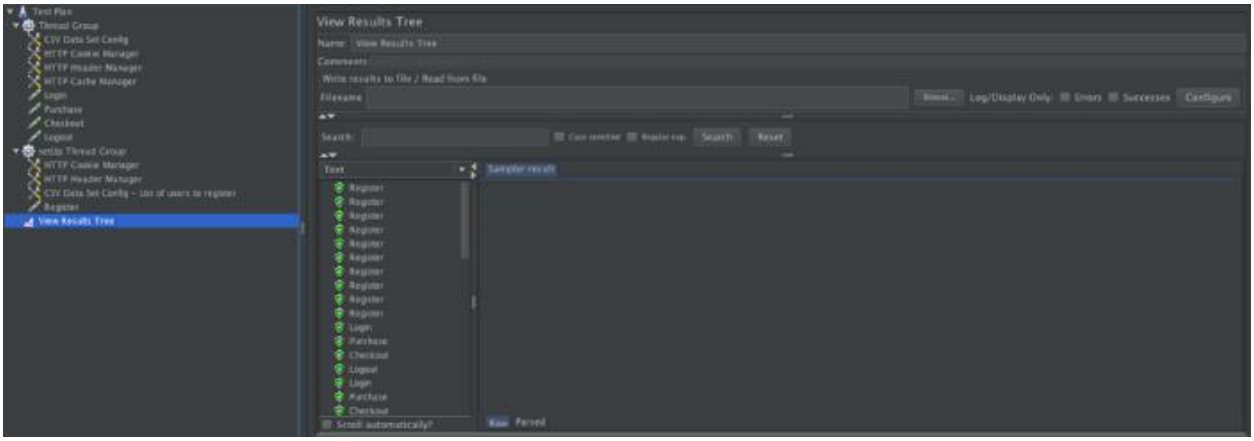
Dodali smo i *HTTP Header Manager* koji specificira vrstu sadržaja kao *application/json* i API ključ s vrijednošću API ključa baze podataka. Zatim dodajemo regularnu grupu niti koja sadrži stvarnu testnu logiku i uzorke:

- prijaviti se
- kupiti
- provjeri
- odjavite se



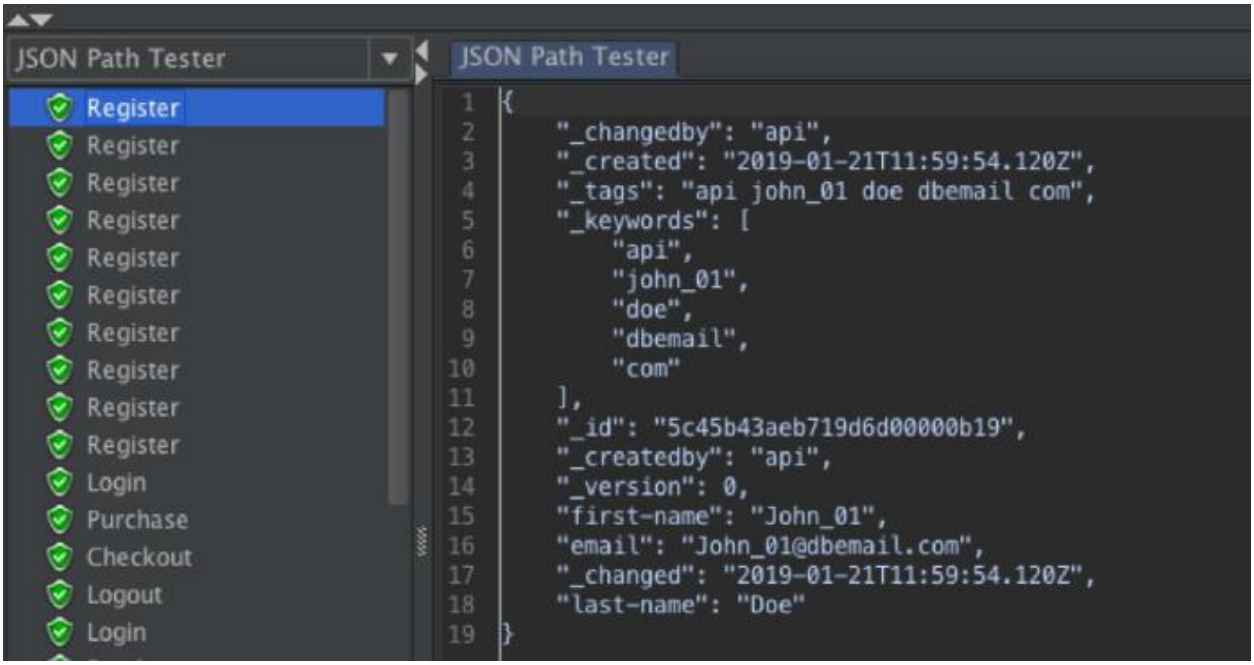
Slika 7.52. Dodavanje regularne grupe niti

Nakon što se pokrene skripta, prva grupa niti koja će se pokrenuti bit će *SetUp Thread Group.* Stvorit će 10 korisnika jer je konfiguriran s 10 niti s jednom iteracijom. Nakon što je grupa niti za postavljanje završena, druga grupa niti će se početi izvršavati, što će biti početak testa opterećenja. Zahtjevi poslani bazi podataka bili su uspješni i primljen je kod odgovora 200. Primijetit ćete da će se grupa niti za postavljanje prva izvršiti, bez obzira na redoslijed kojim se pojavljuje u GUI-ju.



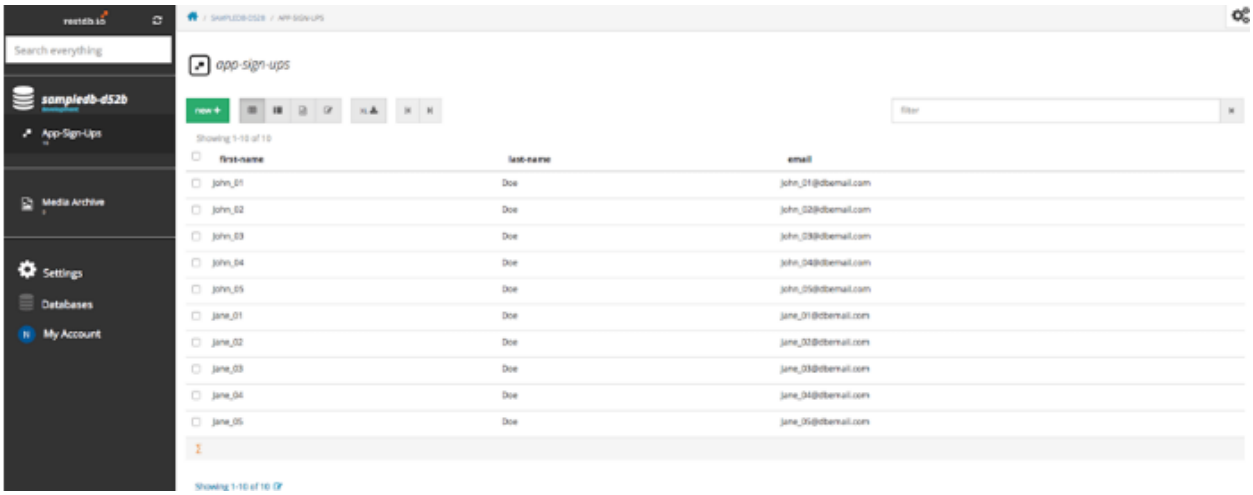
Slika 7.53. Prikaz rezultata

JSON odgovor pokazuje da je zahtjev bio uspješan:



Slika 7.54. Prikaz JSON odgovora

Ako se vratimo na bazu podataka možemo vidjeti da je kreirano 10 korisnika.



Slika 7.55. Prikaz kreiranih korisnika

7.4.6. Čekanje i mjerenje vremena u testiranju pod opterećenjem (engl. *Ramp-up* i *Timer*)

Ramp-up je količina vremena koja će biti potrebna JMeteru za dodavanje svih testnih korisnika (niti) u izvođenje testa. Drugim riječima, koliko će JMeteru trebati da pokrene izvršavanje svih niti. Ako stavimo, na primjer:

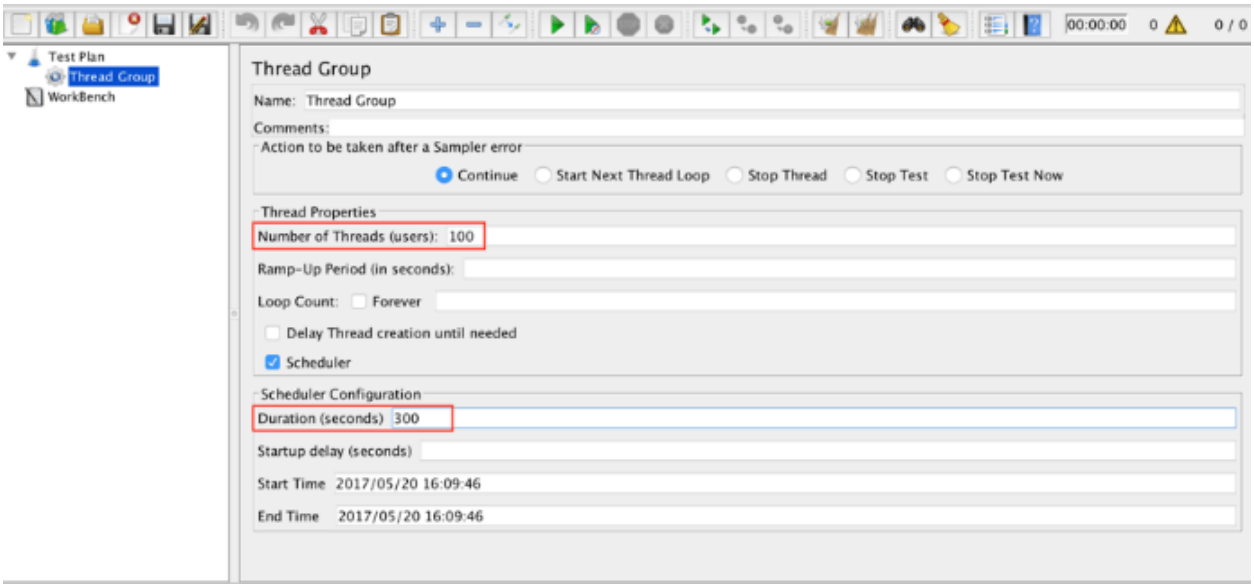
- 1000 ciljnih niti s povećanjem od 1000 sekundi, JMeter će dodati jednog korisnika svake sekunde
- 1000 ciljnih niti s povećanjem od 100 sekundi, JMeter će dodati 10 korisnika svake sekunde
- 1000 ciljanih niti s povećanjem od 50 sekundi, JMeter će dodati 20 korisnika svake sekunde

Ako želimo vidjeti na primjeru, možemo to učini testirajući neku *web*-stranicu.

1. Potrebno je kreirati grupu niti s ciljanim brojem niti (recimo 100).

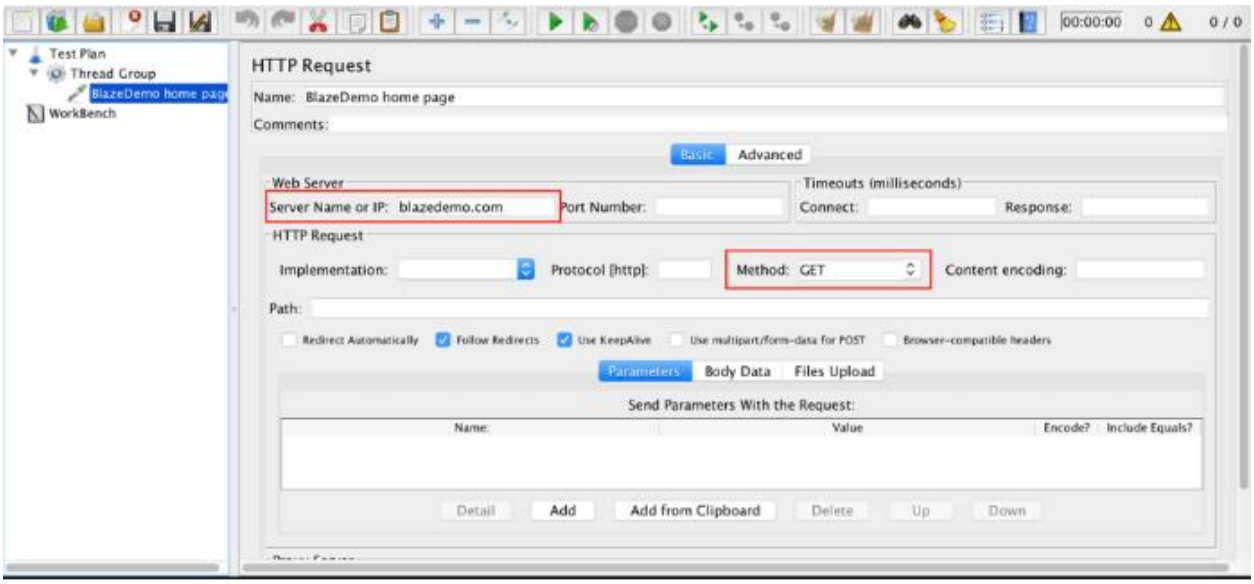
Desni klik -> Dodaj -> Niti -> Grupa niti

Promijenite vrijednost broja niti na 100. Pretpostavimo da želite pokrenuti test na 5 minuta. Morate navesti vrijednost 300 u polju "Trajanje". Za sada neka polje *Ramp-Up Period* u grupi niti bude prazno.



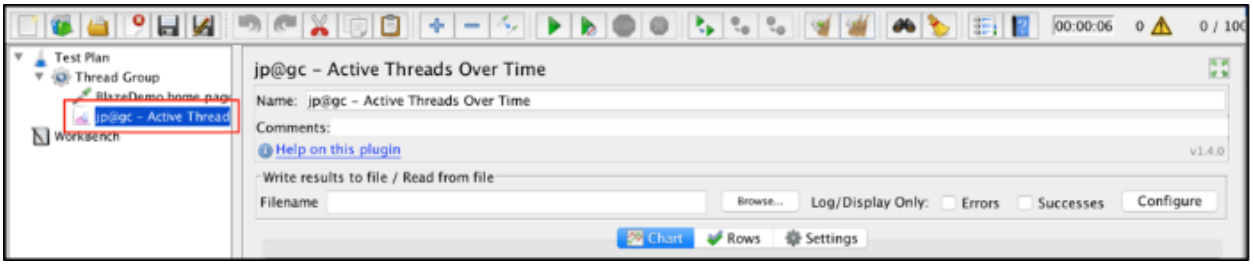
Slika 7.56. Dodavanje Thread Grupe s dodanom vrijednošću 100 za broj niti

2. Dodajte jedan uzorak koji dolazi na početnu stranicu testirane *web*-aplikacije. To će biti GET zahtjev na temelju HTTP protokola.



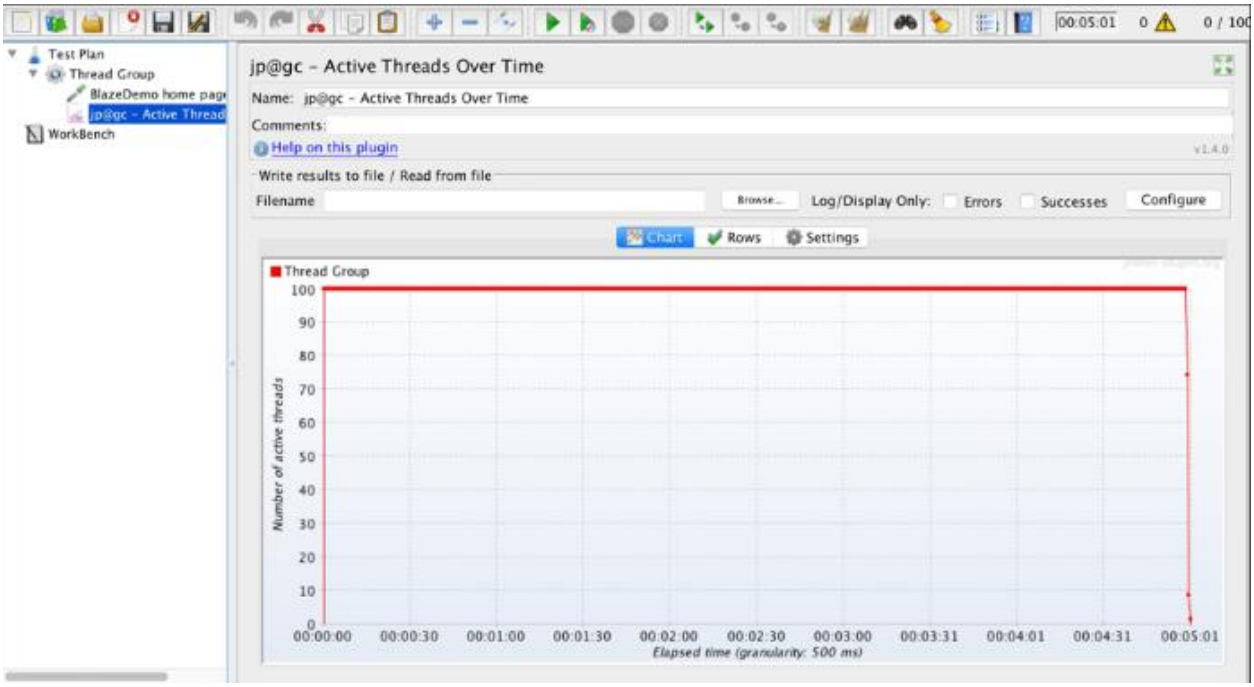
Slika 7.58. Dodavanje GET zahtjeva

3. Korisno je vidjeti grafikon s distribucijom korisnika. Takav grafikon pomaže vizualizirati uzorak opterećenja i provjeriti što će se testirati. JMeter ne nudi ove opcije odmah, ali mogu se jednostavno instalirati pomoću *Custom Plugins Package*, koji sadrži mnoge korisne slušače. Jedna od dodataka zove se “Aktivne niti tijekom vremena” i to je upravo ono što nam za potrebe ovog primjera treba.



Slika 7.59. Dodavanje dodatka “Aktivnih niti tijekom vremena”

Ne zaboravimo da razdoblje povećanja ne bi trebalo biti 0 ako imamo veliki broj ciljnih niti. *Ramp-up* 0 znači da će JMeter skripta za testiranje dodati sve niti odjednom na početku izvođenja testa i stoga će vrlo ozbiljno opteretiti aplikaciju odjednom. Naravno, ponekad ćemo možda htjeti simulirati takvo opterećenje, na primjer tijekom testova performansi skokova, ali ovo je zaseban slučaj koji ima zaseban pristup.



7.60. Prikaz rezultata u obliku grafikona

Ovaj scenarij ne izgleda realno i obično takvo širenje opterećenja nema smisla. Ako imate *web*-aplikaciju, korisnici će je obično više ili manje postupno posjećivati i vaši poslužitelji imaju dovoljno vremena da se prilagode i skaliraju na odgovarajući način.

Ramp-up distribucija trebala bi se temeljiti na vašim potrebama. Stoga, prva stvar koju treba učiniti je razumjeti vaše ciljeve testa. Najbolji način da se razumije cilj testiranja je proučiti trenutne obrasce opterećenja iz verzije koja je već puštena u produkciju i koja ima stvarne korisnike. U isto vrijeme, u mnogim slučajevima dovoljno je stvoriti linearno povećanje, koje pokazuje da korisnici postupno dolaze na testiranu *web*-lokaciju ili aplikaciju.

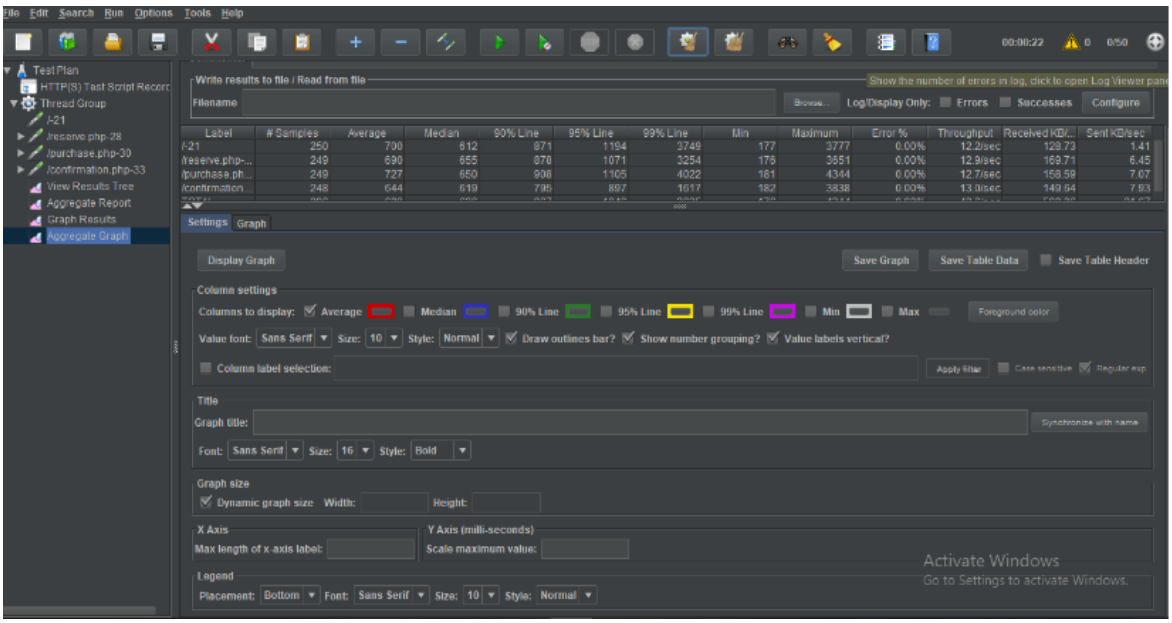
7.4.7. Slušatelji rezultata prilikom testiranja (engl. *Listener*)

Slušatelji pokazuju rezultate izvršenja testa, mogu prikazati rezultate u drugom formatu, kao što je stablo, tablica, grafikon ili datoteka dnevnika.

- Slušatelji rezultata grafikona (*engl. Graph result*) prikazuju vremena odgovora poslužitelja na grafikonu.
- Prikaz stabla rezultata (*engl. View result tree*) prikazuje rezultate korisničkog zahtjeva u osnovnom HTML formatu
- Tablica Rezultati (*engl. Table result*) prikazuje sažetak rezultata testa u obliku tablice.
- Dnevnik (*engl. Log summary*) prikazuje sažetak rezultata testa u tekstualnoj datoteci

Slušatelji rezultata grafikona

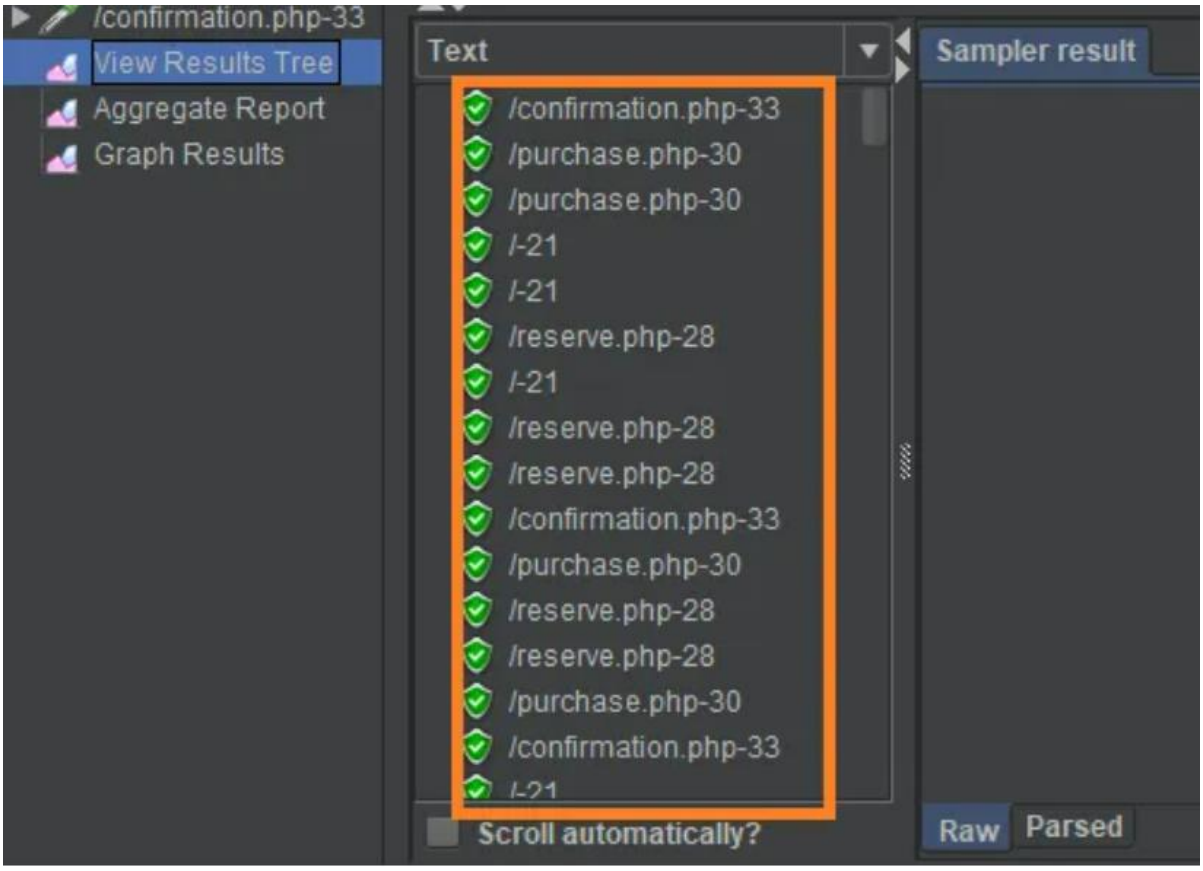
Skupni grafikoni omogućuju nam jednostavno generiranje stupčastih grafikona i odabir postavki prikaza grafikona. Generirani podaci grafikona i tablice mogu se eksplicitno spremiti u obliku PNG i CSV.



Slika 7.61. Dodavanja skupnih grafikona

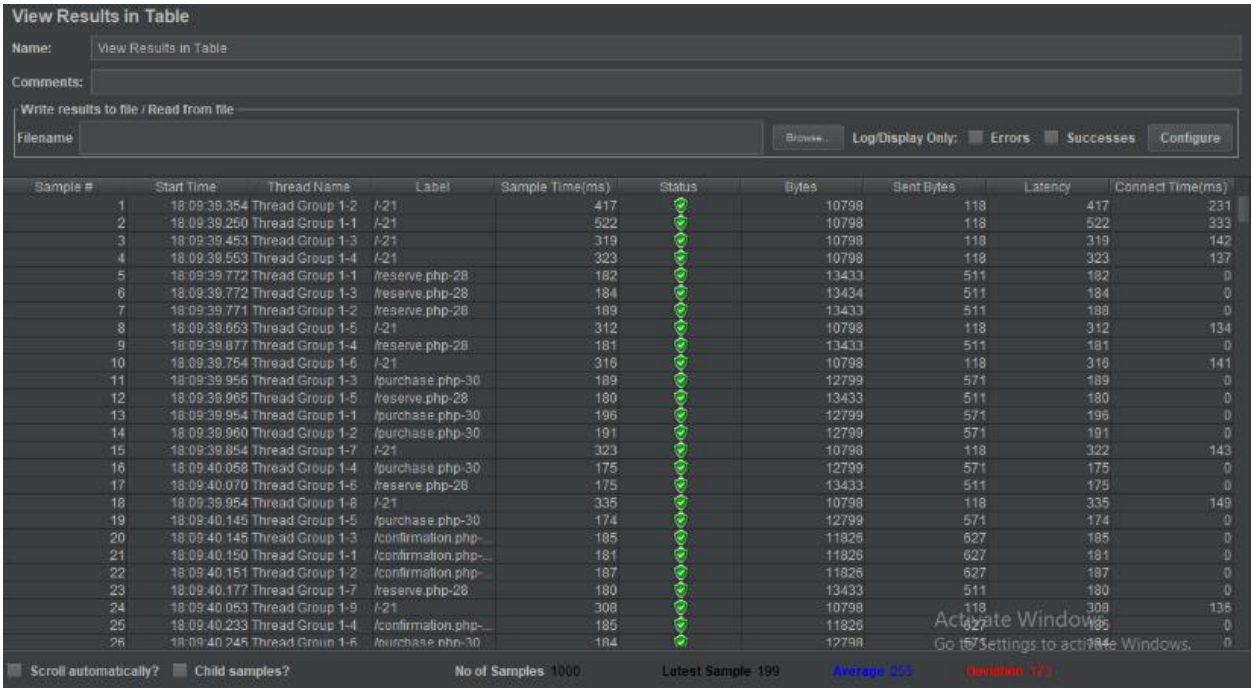
Prikaz stabla rezultata prikazuje rezultate koji se sastoje od svih odgovora *Samplera* s njihovim zahtjevima. Prikaz stabla rezultata ne koristi se tijekom testa opterećenja/stresa

jer troši puno resursa poput memorije i CPU-a. Preporučuje se koristiti ga za funkcionalno testiranje ili u svrhu otklanjanja pogrešaka.



Slika 7.62. Prikaz stabla rezultata

Tablica rezultata stvara i prikazuje red za svaki uzorak/rezultat zahtjeva zasebno. Tablica rezultata se isto tako ne smije koristiti tijekom testa opterećenja/stresa jer troši puno resursa poput memorije i CPU-a. Preporučuje se koristiti ga u svrhu funkcionalnog testiranja ili uklanjanja pogrešaka.



Slika 7.63. Tablica rezultata sa statusom svake Thread grupe

7.4.8. Pokretanje JMetera iz komandne linije i parametri

Apache JMeter je alat za testiranje otvorenog koda. JMeter funkcionira za analizu i mjerenje performansi *web*-aplikacija s nizom drugih usluga. GUI način rada JMetera savršen je za dodavanje i uređivanje novih konfiguracijskih elemenata, grupa niti i uzorkivača, a rezultat toga možete vidjeti kao niz različitih slušatelja koji pomažu u otklanjanju pogrešaka.

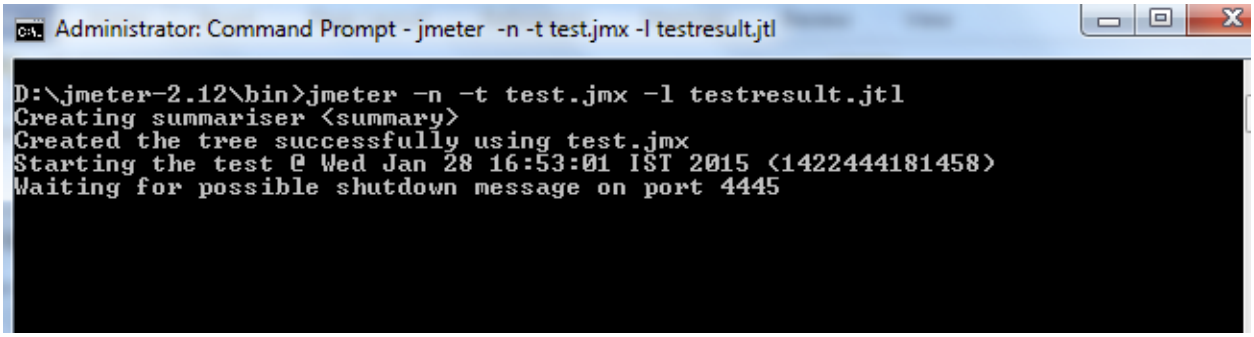
Međutim, GUI način rada ima ograničenje koje usporava korištenje procesora tijekom pokretanja snimljene skripte. Ako pokrećemo više slušača u skripti, to utječe na performanse JMetera. Da bi se prevladala takva situacija, skriptu treba pokrenuti u načinu rada koji nije GUI. Postoji prednost pokretanja više zahtjeva u sekundi iz JMetera u načinu rada bez GUI-a.

Zašto koristiti JMeter u načinu rada bez GUI/komandne linije:

- povećanje niti (nakon određenog ograničenja) zbog čega se JMeter ruši u GUI načinu
- za teške testne scenarije (npr. aplikacija za kupovinu: prijava-pregled proizvoda-dodaj u košaricu-pregled računa-ukloni proizvod-plaćanje računa) JMeter troši memoriju i CPU i može utjecati na rezultate testa
- poveća mogućnosti JMetera, tj. dobiva se više zahtjeva u sekundi.

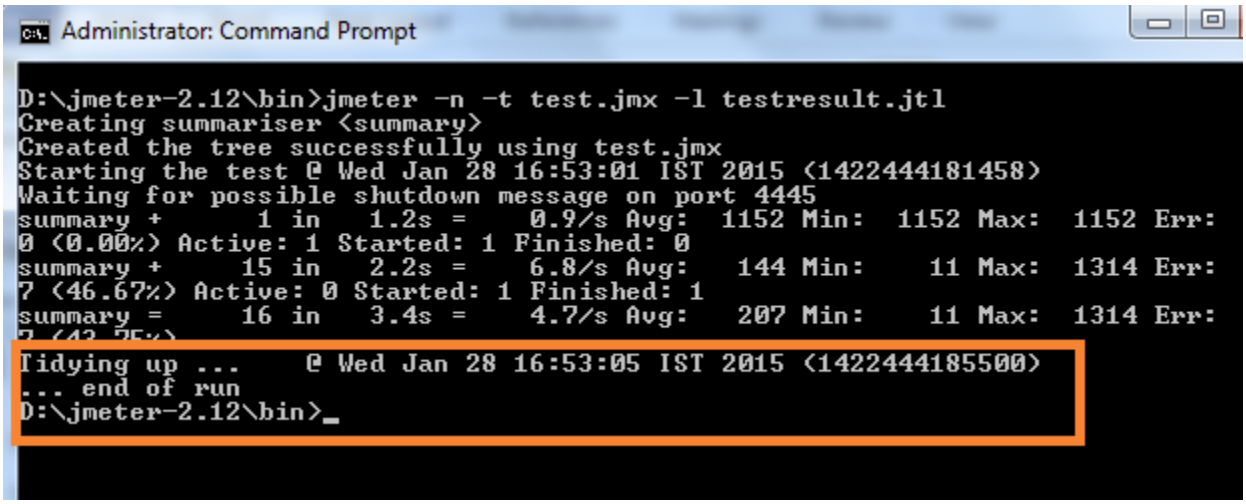
Pokretanje JMetera korištenjem naredbenog retka u načinu rada koji nije GUI vrlo je jednostavno:

- otvorite naredbeni redak i otvorite u JMeterovu mapu ***bin***
- unesite sljedeću naredbu, ***jmeter -n –t test.jmx -l testresults.jtl***.



Slika 7.64. Pokretanje JMetera korištenjem naredbenog retka

Kada je test dovršen, na zaslonu će se prikazati ovo:



Slika 7.65. Prikaz dovršenog testa

Nakon završetka testa, JTL datoteka generirat će se u mapi jmeter2.12/bin. Ovo su koraci koje trebamo poduzeti kako bismo vidjeli rezultate testa iz JTL datoteke:

- Otvorite JMeter u GUI modu.
- Dodajte bilo kojeg slušatelja, npr. Prikaz stabla rezultata.
- Kliknite gumb Pregledaj polja naziva datoteke u slušatelju.
- Otvorite datoteku testresult.jtl.

Pitanja za ponavljanje:

1. Što je to Cucumber?
2. Kako povezati Cucumber s *feature* datotekom?
3. Koje su metode koje pozivamo kod HTTP protokola?
4. Što znači kod 200?
5. Što je JMeter i za što se koristi?
6. Što su to slušači u Jmeteru i za što se koriste?
7. Što nam prikazuje View result tree?
8. Što je to JSON odgovor u JMeteru?

8. Zaštita na radu

U OVOM POGLAVLJU NAUČIT ĆETE:

- razlikovati mjere zaštite na radu
- razlikovati različite izvore opasnosti pri radu s računalom
- navesti mjere zaštite od električnog udara na radnom mjestu
- objasniti upotrebu računala na siguran način primjenom mjera zaštite od ozljeda.

8.1. Izvori opasnosti i rad u uredu

Postoji mnogo izvora opasnosti koje treba uzeti u obzir u uredskom radnom okruženju. Za zaštitu od izvora opasnosti, preporučljivo je implementirati sigurnosne procedure i osigurati da su svi uređaji i instalacije u dobrom stanju. Ako se pojave bilo kakvi sigurnosni problemi, potrebno ih je odmah prijaviti i riješiti u što kraćem roku.

8.1.1. Utjecaj obavljanja poslova na vid

Određene aktivnosti povezane s radom u uredu, kao što su dugotrajno sjedenje, rad na računalu i čitanje dokumenata, mogu imati negativan utjecaj na vid. Kako bi se smanjili ovakvi negativni učinci na vid, preporučljivo je osigurati da radno okruženje uključuje adekvatno osvjetljenje, korištenje anti-refleksne zaštite za ekran računala i redovito mijenjanje položaja tijekom rada. Osim toga, važno je redovito vježbati oči i odmarati ih, korištenjem naočala za rad na računalu ili pauze za čitanje dokumenata.

Prije svega, potrebno je provjeriti jesu li zaslon i zaslonsko okruženje prilagođeni za rad s obzirom na udaljenost, nagib i jasnoću. Također, potrebno je provjeriti je li razina osvjetljenja u prostoru u kojem se obavlja posao primjerena. Ako je osvjetljenje prejako, treba ga smanjiti. Pored toga, potrebno je osigurati da su performanse zaslona i osvjetljenja u skladu s preporukama proizvođača.

Također je korisno razmotriti potrebu za pojačanim zaštitnim naočalama od zaslona. Ove naočale mogu smanjiti utjecaj zaslona na vid i olakšati prilagodbu položaja zaslona. Također, važno je provjeriti postoji li potreba za podešavanjem postavki zaslona kako bi se postigla optimalna razina udobnosti.

Pored toga, potrebno je provjeriti imate li pravilno držanje tijekom obavljanja posla. Ako je držanje nepravilno, može uzrokovati neugodnosti i utjecati na kvalitetu obavljanja posla. Konačno, potrebno je provjeriti izvodi li se prilagođavanje položaja zaslona redovito. To će pomoći osigurati da je zaslon postavljen na optimalnu udaljenost i nagib te da je jasna i boja zaslona postavljena na najbolju razinu.

Kako biste osigurali da obavljanje posla ne utječe negativno na vid i prilagodbu položaja zaslona, potrebno je pridržavati se preporučenih mjera.

8.1.2. Posljedice dugotrajnog sjedenja i ergonomski faktori

Dugotrajno sjedenje može imati negativne posljedice za zdravlje, uključujući povećan rizik od dijabetesa, prekomjerne tjelesne težine, visok krvni tlak, visok kolesterol i bolesti srca. To može dovesti i do povećanog rizika od povreda leđa i vratnih mišića.

Ergonomski faktori koji mogu pomoći kod dugotrajnog sjedenja uključuju odabir kvalitetne i pravilno postavljene stolice. Kvalitetna stolica treba biti udobna, s potporom za leđa i potporom za ruke. Dijelovi stolca kao što su nasloni za ruke i potpora za noge trebaju biti podešeni tako da odgovaraju vašem tijelu. Stolica bi trebala biti postavljena na odgovarajuću visinu, tako da se koljena mogu ravnomjerno rastegnuti pod stolom. Rasterećenje je važno kako bi se smanjili mišićni i zglobni uvjeti. To uključuje pravovremeno pokretanje i kretanje tijekom dana, kao i odabir udobnijih poza.

Udobnost sjedenja se može postići promjenom položaja ili odabirom pravog naslona za leđa. Konačno, potrebno je osigurati udobno sjedenje tako da se leđa zadržava uspravno i da se stopala ravno postavljaju na pod.

8.1.3. Mehaničke opasnosti

Mehaničke opasnosti rada u uredu uključuju ozljede od pada i napore dizanja ili premještanja električnih uređaja. Kako bi se izbjegle ove opasnosti, treba se pridržavati sljedećih preporuka:

1. Redovito provjerite opremu kako biste se uvjerili da je bez oštećenja i da je ispravno postavljena.
2. Koristite opremu kako je namijenjeno, odnosno uvijek slijedite upute proizvođača
3. Redovito održavajte svoju opremu kako biste se uvjerili da radi ispravno.

8.1.4. Opasnosti od električnog udara

Opasnost od električnog udara je stvarna i stoga je važno pažljivo postupati s električnim uređajima. Izbjegavajte sve nepotrebne kontakte s električnim kablovima, utičnicama i drugim dijelovima električnih uređaja.

Kada rukujete električnim uređajima i napravama, obavezno koristite zaštitne rukavice i cipele. Nikada ne dodirujte električne dijelove s bilo kojom vlažnom stranom tijela, jer to može uzrokovati električni udar.

Ako često rukujete električnim uređajima i uređajima, obavezno ih pregledajte prije upotrebe. Provjerite postoje li na njima oštećenja koja mogu uzrokovati električni udar.

Kada rukujete električnim uređajima i uređajima, obavezno držite njihove dijelove od sebe. Čak i ako su uređaji i naprave ispravni, postoji mogućnost da se električno polje širi između dva dijela, što može uzrokovati električni udar.

Također, važno je pridržavati se uputa proizvođača u vezi s rukovanjem električnim uređajima. Ovi savjeti obično uključuju upozorenja o korištenju odgovarajuće zaštitne opreme, kao i upute za ispravno povezivanje električnih uređaja.

Ako se redovito pridržavate ovih mjera opreza, možete značajno smanjiti rizik od električnog udara. Ako se nađete u situaciji da je potrebna pomoć osobi koja je pod utjecajem strujnog udara važno je pridržavati se ovih koraka:

1. prekinite vezu između izvora struje i osobe
2. ako je osoba nastradala dok je bila u vodi, udaljite je od vode prije nego što pokušate prekinuti vezu s izvorom struje.
3. provjerite je li osoba živa. Ako nije, pokušajte izvršiti reanimaciju.
4. ako je osoba živa, postavite je tako da leži na ravnoj površini.
5. uklonite sve metalne predmete s osobe (na primjer nakit, satove, ogrlice i naušnice).

6. ako postoje otekline ili opekotine, ohladite ta mjesta hladnom vodom.
7. ako se pojave konvulzije, postavite osobu na bok kako biste spriječili da se uguši.
8. ako su prisutni povraćanje ili dijareja, postavite osobu u sjedeći položaj.
9. ako je osoba svjesna, pokušajte je umiriti i ohrabriti da se opusti.
10. pozovite hitnu pomoć.

8.1.5. Opasnost od požara

Opasnost od požara postoji u slučaju kada se uređaji pregrijavaju, što može dovesti do kratkog spoja, požara ili čak eksplozije. Kako biste spriječili takve situacije, potrebno je ispravno isključiti uređaje, provjeravati njihovu funkcionalnost, redovito čistiti ventilacijske otvore, a ako je potrebno i servisirati. Prije uključivanja uređaja, potrebno je provjeriti jesu li napojni kabel i utikač u dobrom stanju i je li napajanje preopterećeno. Također, uređaj treba staviti na mjesto gdje neće biti izložen toplini ili direktnom sunčevom svjetlu, kao i odvojiti je od ostalih uređaja. Ako uređaj ne radi ispravno, ili je previše vruć, potrebno je odmah ga isključiti iz napajanja.

Pitanja za ponavljanje:

1. Objasnite kako položaj monitora može utjecati na vid.
2. Koje su mehaničke opasnosti rada u uredu?
3. Koji su koraci potrebni kako bi se pomoglo osobi koja je pod utjecajem strujnog udara?
4. Kako spriječiti opasnost od kratkog spoja, požara ili eksplozije?
5. Nabrojite neke od ergonomskih faktora koji utječu kod obavljanja poslova koji se velikim dijelom obavljaju sjedeći.

LITERATURA:

- Hendrickson, E. (2013). Explore It! Reduce Risk and Increase Confidence with Exploratory Testing, Pragmatic Bookshelf
- Myers, G. J., Badgett, T. i Sandler C. (2012). The Art of Software Testing, Third Edition, Word Association.
- Smart, J. F. (2014). BDD in Action: Behavior-driven development for the whole software lifecycle, Manning
- Spillner, A., Linz, T., Shaefer, H. (2014). Software Testing Foundations, 4th Edition: A Study Guide for the Certified Tester Exam, Rocky Nook

MREŽNE STRANICE:

- Apache JMeter™, <https://jmeter.apache.org/usermanual/index.html>. Pristupljeno 28.12.2022.
- Gamba, S., Graham, D. (2018) A Journey Through Test Automation Patterns, <https://testautomationpatterns.org/wiki/index.php/References>. Pristupljeno 28.12.2022.
- Golovkova, A. (2021). REST API Testing - How to Do it Right. <https://www.blazemeter.com/blog/rest-api-testing#authoranastasiagolovkova >. Pristupljeno 28.12.2022.
- ISTQB®, Certified Tester Foundation Level Sylabus (CTFL). (2018), <https://www.istqb.org/certifications/certified-tester-foundation-level>. Pristupljeno 22.12.2022.
- Knight, A. (2017) Automation Panda < https://automationpanda.com/2017/02/03/bdd-101-automation/>. Pristupljeno 28.12.2022.
- Preventa, Zaštita na radu pri uredskim poslovima I radu s računalom, <https://preventa.hr/zastita-na-radu-upit/zastita-na-radu-pri-uredskim-poslovima-i-radu-s-racunalom>. Pristupljeno 28.12.2022.

- Zaštita na radu pri uredskim poslovima Centar za integralnu sigurnost. https://preventa.hr/zastita-na-radu-upit/zastita-na-radu-pri-uredskim-poslovima-i-radu-s-racunalom

Svi logotipi i nazivi prikazani u ovom priručniku su vlasništvo respektivnih kompanija.

“Sadržaj publikacije/emitiranog materijala isključiva je odgovornost RCK ELPROS”

Projekt je sufinancirala Europska unija iz Europskog socijalnog fonda.