Testiranje web aplikacija

Kukuljan, Nina

Undergraduate thesis / Završni rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka / Sveučilište u Rijeci**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:195:326338

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-14**

Repository / Repozitorij:

Repository of the University of Rijeka, Faculty of Informatics and Digital Technologies - INFORI Repository

Sveučilište u Rijeci – Fakultet informatike i digitalnih tehnologija

Preddiplomski jednopredmetni studij informatike

Nina Kukuljan

Testiranje web aplikacija

Završni rad

Mentor: Doc. Dr. sc. Lucia Načinović Prskalo

Vanjski mentor: Andrej Arbanas, Juice d.o.o.

Rijeka, 25.6.2022.

ODJEL ZA INFORMATIKU
SVEUČILIŠTE U RIJECI

Rijeka, 20.5.2022.

Zadatak za završni rad

Pristupnik: Nina Kukuljan

Naziv završnog rada: Testiranje web aplikacija

Naziv završnog rada na eng. jeziku: Web application testing

Sadržaj zadatka: Glavni cilj ovog završnog rada je dati pregled razvoja i opisati vrste standarda za testiranje web aplikacija te na primjeru web aplikacije za finaliziranje testiranja web aplikacija pokazati osnovne funkcionalnosti i slijed postupaka od početne do završne faze testiranja.

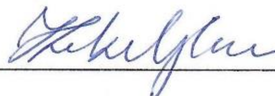
Mentor

Doc. dr. sc. Lucia Načinović Prskalo

Voditelj za završne radove

Doc. dr. sc. Miran Pobar

Zadatak preuzet: 20.5.2022.



(potpis pristupnika)

Sadržaj

1. Sažetak ... 4
2. Uvod ... 5
3. Testiranje kao metoda ... 7
4. Tipovi i tehnike testiranja ... 10
   4.1. Testiranje funkcionalnosti ... 10
   4.2. Testiranje sučelja ... 13
   4.3. Testiranje kompatibilnosti ... 15
   4.4. Testiranje performansi ... 15
   4.5. Testiranje baze podataka ... 17
   4.6. Testiranje sigurnosti ... 18
   4.7. Masovno testiranje ... 19
5. Izrada aplikacije za osiguravanje kvalitete ... 20
   5.1. Korišteni alati ... 21
   5.2. Načina rada Quality Assurance modula ... 23
   5.3. Pojašnjenje koda *Quality Assurance* modula ... 26
6. Zaključak ... 36
7. Popis Slika ... 37
8. Literatura ... 38

1. Sažetak

U ovom završnom radu osvrnut ću se na sve faze razvoja testiranja web aplikacija, za što se koristi, kako olakšava te koje mjesto zauzima u razvoju web aplikacija. Također će biti riječi i o tome kako je došlo do same potrebe za testiranjem web aplikacija te zašto nam je ono važno kao i opširan opis nekih od najkorištenijih tehnika kojim se testiranje provodi. Naposljetku, kroz primjer izrađene web aplikacije za ocjenjivanje kvalitete proći ću kroz postupke njene izrade i položaj u životnom vijeku drugih web aplikacija čije testiranje olakšava.

Ključne riječi: Testiranje, greške, metode, web aplikacija, funkcionalnosti, ocjenjivanje kvalitete

2. Uvod

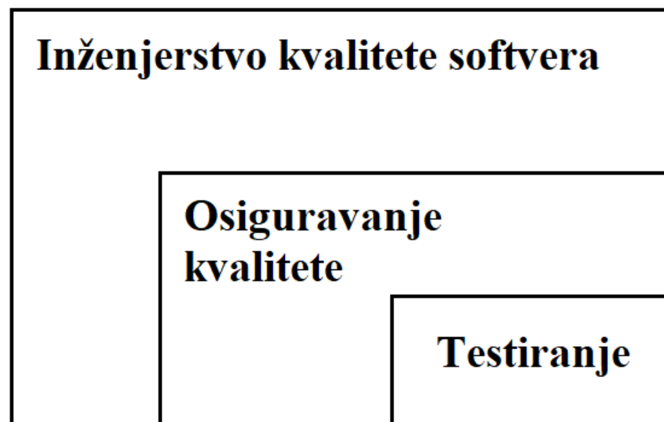
Prije razvoja softvera i raznih drugih informacijskih tehnologija kvaliteta se asocirala isključivo s fizičkim objektima poput auta, televizija, radija itd. U tom kontekstu se osiguravanje kvalitete (*eng. Quality assurance; QA*) većinom povezivalo s procesom izrade proizvoda, odnosno provjera odgovara li svojim specifikacijama. U to vrijeme česta mjera kvalitete je bila prijava defekata od strane korisnika, pošto su se same specifikacije većinom dobivale uz proizvod najčešće u obliku priručnika za upotrebu. Te prijave s nekom određenom veličinom uzorka mogle su se kasnije koristiti kao mjera kvalitete. (Tian 2005)

S razvojem industrije i dolaskom softvera kao proizvoda na tržište, pogled na kvalitetu i njeno mjerenje se promijenio. Fokus više nije bio na problemima s proizvodom već problemima koje korisnici imaju s proizvodom van njegovih predodređenih specifikacija. Iskustvo korisnika preuzima veću važnost od praćenja specifikacija. Ukratko, korisnici trebaju potvrdu da su njihova očekivanja vezana za kvalitetu softvera zadovoljena.

Proizvodnja web aplikacija gotovo uvijek prati ustaljeni slijed akcija. Od samih početaka koje uključuje planiranje i dizajniranje pa preko same izrade i programiranja aplikacije, neizbježni dio životnog vijeka web aplikacije je i osiguravanje njegove kvalitete. Od svakodnevnih potrošača do velikih kompanija, danas se gotovo svi na računala oslanjaju više nego ikada. Samim time posljedice nedostataka, grešaka i potencijalnih problema unutar bilo kakvog računalnog sustava postaju sve ozbiljnije.

Raznim *Quality Assurance* (QA) aktivnostima pokušavaju se eliminirati neki od tih problema ili barem smanjiti njihovo pojavljivanje. Metode koje takve aktivnosti uključuju su primjerice inspekcija, formalna verifikacija, prevencija defekata i tolerancija grešaka. Inspekcija na primjer provjerava kod i druge artefakte te tako pronalazi i eliminira probleme direktno, bez da se uopće dolazi do izvršavanja.
Tolerancija grešaka pak sprječava globalne sistemske propuste čak i ako na nekim lokalnim dijelovima postoje problemi.

Testiranje kao jedna od tih metoda pomoću raznih tehnika osigurava kvalitetu aplikacije tako što testira njene funkcionalnosti, specifikacije i planirane ishode dogovorene pri njenom početnom planiranju i zahtjevima klijenta. (Tian 2005)



Slika 1 Hijerarhija testiranja, osiguravanja kvalitete i inženjerstva kvalitete softvera

Niti jedna web aplikacija nije imuna na greške, '*bugove'* i mnoge druge probleme. Čak i najsavršenije programiranje i dalje može imati neplanirane ishode te bez testiranja krajnji proizvod nikada neće biti na nivou potpuno funkcionalne i kompletne aplikacije. Štoviše, na današnjem tržištu gdje su web aplikacije ništa više no proizvodi, pravilan pristup testiranju može izbjeći mnogobrojne troškove. Što se više približavamo kompletiranoj aplikaciji, to je testiranje i popravak nađenih problema skuplji i teži za izvedbu.

3. Testiranje kao metoda

Testiranje se od njegovih skromnih početaka kao dio procesa otklanjanja pogrešaka *(eng. Debugging)* u 60-im godinama prošlog stoljeća postupno poboljšavalo i unaprjeđivalo. (Webomates 2022) No paralelno s razvojem tehnika testiranja, tehnologije i web aplikacije za koje se ono koristi nisu nimalo zaostajale, dapače njihovo unapređivanje i razvoj možda je i brži.

S povećanjem kompleksnosti softvera i njegova razvoja, zahtjevi koje ono stvara za ljude uključene u proces testiranja se također povećavaju i poprimaju sve veći značaj. Testeri moraju pratiti ako aplikacija pravilno ispunjava zadane funkcije te otkriti probleme koji možda nisu bili predviđeni u originalnom dizajnu. Da bi se svi ti zahtjevi uspješno i što kvalitetnije odradili testeri moraju razumjeti proizvod i tehnologije korištene u njegovu razvoju. Od njih se očekuje da imaju široki spektar znanja pošto je testiranje najefikasnije upravo kada imamo unaprijed razvijene ideje što sve može poći po krivu i znanje kako takve greške prepoznati. To znanje se prikuplja iz mnogo različitih područja, kako samog poznavanja aplikacije i njene arhitekture, njenih traženih funkcionalnosti i načina korištenja pa sve do same integracije i razvoja na razini koda i programiranja. Uza sve te zahtjeve i potrebne kompetencije testera, mnoge kompanije dodatno otežavaju stvari s uvjerenjem da jedini način kako konkurirati na današnjem tržištu je proizvesti softver što je brže moguće. Iako nekad istinito, s druge strane postaje sve jasnije da iako je važnost brze isporuke velika, tolerancija tržišta za '*buggy*' aplikacije drastično i naglo opada. Osim što se web aplikacija danas mora izraditi što brže, bitno je povećati ulaganja i alokaciju resursa ka testiranju.

Različite ciljne skupine za koje se web aplikacija proizvodi, zahtijevaju i različite načine testiranja. Generalno, većina tipova testiranja može se podijeliti na dva glavna pristupa: takozvani *black-box* i *white-box* testiranje (Hung Q. Nguyen 2003). Kod *black-box* testiranja fokus je na očekivanom ishodu i/ili ponašanju aplikacije bez ikakvog znanja o internim funkcijama ili radu softvera od čega i potječe naziv – testeri ne vide 'unutar crne kutije' već samo mogu vidjeti njenu vanjštinu. S druge strane, *white-box* testiranje također poznato pod nazivom *glass-box* u kojem opet ime objašnjava pristup – testeri vide 'unutrašnjosti kutije' tj. znaju internu strukturu aplikacije kao i strukturu podataka te razumiju logički slijed koda. U većini testnih slučajeva koristiti samo jedan od ova dva ekstrema nije previše povoljno.

S tim na umu u priču dolazi *gray-box* testiranje čije ime također slikovito objašnjava o čemu se radi. Naime *gray-box* testiranje kombinira aspekte *white* i *black-box* testiranja u jedan kohezivan i visoko funkcionalan pristup. Bavi se ishodima s korisničke strane no također koristi i specifična tehnička i sistemska znanja.

Gray-box pristup (Hung Q. Nguyen 2003) savršeno paše testiranju web aplikacija upravo jer ima mogućnost otkrivanja problema koje white ili *black-box* metode previde ili nemaju mogućnosti pronaći. Greške specifično vezane za kontekst u kojem se događaju ili problemi vezani za opći tijek informacija od 'početka do kraja' se često uspješno pronalaze upravo pomoću ove tehnike. No biti ekspert i poznavati svaki nivo aplikacije nije najjednostavnije niti nužno potrebno te testni timovi koji u sebi imaju *white-box* i *black-box* članove mogu imati veću pokrivenost na raznim područjima specijalnosti te jednako dobro, ako ne i bolje, određivati kvalitetu i testirati web aplikaciju.

Black-box	**Gray-box**	***White-box***
Nije potrebno ekstenzivno znanje o softveru niti njegovu radu.	Potrebno je znanje softvera, no samo ono koje je relevantno.	Potrebno je detaljno razumijevanje koda i baze podataka.
Korisno za testiranje funkcionalnosti, dizajna sučelja…	Korisno za dubinsko testiranje funkcionalnosti.	Pogodno za jedinično testiranje *(eng. Unit testing)*.
Prilično jednostavno za izvedbu; osim testera izvode ga i korisnici.	Nešto teže za izvedbu; osim korisnika izvode ga i programeri i testeri.	Najteže za izvedbu; izvode ga isključivo programeri i ponekad testeri.

Tablica 1 Usporedba različitih pristupa testiranju (Hung Q. Nguyen 2003)

Pošto niti jedan tester nije isključivo white, black ili gray box tester, kombinacije ove tri metode kroz godine razvile su se u brojne podtipove i načine testiranja. Od manualnog testiranja koje koristi detaljne testne slučajeve, preko automatiziranih testova koji danas osim pozadinskih (*eng. Backend)* funkcionalnosti, povezanih s bazom podataka, mogu testirati i pristupne (*eng. Frontend)* dijelove aplikacije brže i ekonomičnije nego ikad. Još jedna bitna pojava u načinima testiranja je i kontinuirana integracija i implementacija koja još dodatno olakšava i skraćuje isporuku web aplikacija (*eng. CI/CD*, *Continuous integration/Continuous deployment*). Danas postaje sve popularnije uključivati i koristiti umjetnu inteligenciju za testiranje koja CI/CD dovodi do nove razine efikasnosti.

4. Tipovi i tehnike testiranja

Kako različiti ciljevi zahtijevaju različite pristupe testiranju, tako su se za pojedine dijelove web aplikacije razvili tipovi testiranja koji im najviše odgovaraju. Neki od tih tipova padaju gotovo u potpunosti pod *black-box* metodu kao na primjer testiranje funkcionalnosti (*eng. Functionality testing*) i regresijsko testiranje (*eng. Regression testing)* dok ispitivanje stanja (*eng. Condition testing*) spadaju u podskup *white-box* metode testiranja. U ovom poglavlju detaljnije ću objasniti neke od najkorištenijih tehnika odnosno onih koje se po današnjim standardima mogu smatrati dobrom i poželjnom praksom.

4.1. Testiranje funkcionalnosti

Testiranje funkcionalnosti je jedna od najširih kategorija testiranja. Glavni fokus je upravo testiranje osnovnih funkcionalnosti aplikacije. Jesu li te funkcionalnosti korisne klijentu? Rade li na predviđen način? Ta i mnoga druga pitanja vezana uz samu srž korištenja aplikacije testiraju se uz pomoć raznih metoda. Neke od njih uključuju (Hung Q. Nguyen 2003):

- FAST - *(eng. Functional acceptance simple tests)* provjerava osnovne funkcionalnosti svake naredbe programa; vrlo plitka pokrivenost pošto se ne testiraju kombinacije tih funkcija
- TOFT - *(eng. Task-oriented functional tests)* provjerava izvršavaju li se određeni zadaci unutar razumnih očekivanja korisnika
- Testovi prisilne pogreške – *(eng. Forced error tests)* namjerno izazivanje grešaka kako bi se pronašle loše odrađene ili neotkrivene greške
- Testiranje graničnih uvjeta – *(eng. Boundary condition tests)* ekstenzija TOFT-a; bavi se ispravnim radom uslijed postignutih krajnjih vrijednosti specificiranog korištenja
- Eksplorativno testiranje – *(eng. Exploratory testing)* uključivanje iskustva u testiranju problematičnih dijelova aplikacije

4.1.1. Jednostavni testovi funkcionalne prihvatljivosti

Jednostavni testovi funkcionalne prihvatljivosti ili *eng. FAST* (Hung Q. Nguyen 2003), pokriva veliku širinu no vrlo plitku korisnost aplikacije. Testiraju se najjednostavnije funkcionalnosti aplikacije, no ne i njihove kombinacije (bez obzira koliko čvrsto integrirane). Jedan od glavnih ciljeva *FAST*-a je testiranje kontrola korisničkog sučelja. Bilo da se radi o polju za upis, radio gumbu, padajućem meniju, klizaču… Provjerava se prikazuju li se te komponente, prikazuju li se ispravno, na pravom mjestu, postoji li originalna vrijednost i pokazuje li se ispravno. Osim toga ova metoda daje uvid u sam način implementacije logike i kreiranja tih funkcionalnosti koje se zahtijevaju. Time ona nudi važne informacije potrebne za kreiranje daljnjih testnih slučajeva koji će se baviti upravo preispitivanjem tih logika.

4.1.2. Funkcionalni testovi usmjereni na zadaće

Funkcionalni testovi usmjereni na zadaće, ili skraćeno *TOFT*, tipovi su testova koji uzimaju u obzir ono što *FAST* nije uspio. Kombinacije funkcija te uz njih i mogućnost aplikacije da izvede zadatke koji su se od nje tražili ne samo na očekivan već i koristan način. Rezultati izvršenih zadataka uspoređuju se sa zadanim specifikacijama i zahtjevima od strane korisnika. Jasno je da se ovakvi testovi fokusiraju i vode ka listama značajki i samim time vrlo je važno da se od korisnika jasno znaju specifikacije projekta i njihova razumna očekivanja. Ovakvim pristupom često se otkrivaju i loše definirani zahtjevi ili zahtjevi bez dobrih specifikacija.

4.1.3. Testovi prisilne pogreške

Takozvani *FETs* (Hung Q. Nguyen 2003) se fokusiraju na neminovne neuspjehe i greške aplikacije. Svaka aplikacija ima situacije gdje ‘pukne’, no takve situacije su gotovo neizbježne i trebaju se riješiti na što elegantniji način. Primjeri ‘gracioznih’ rješavanja grešaka bili bi da se aplikacija, kao i cijeli sustav, oporavi uspješno, da se obustava rada odradi bez gubitaka i/ili korupcije podataka, ili čak nešto jednostavno poput notifikacije da je predviđena tablica prazna umjesto totalne zbrke i crvene poruke u konzoli. Još jednostavniji primjer forsiranja pogrešaka bilo bi testiranje polja u obrascu. Većinu vremena određena polja zahtijevaju određene tipove ili oblike podataka. Namjerno upisivanje krivih podataka i provjera da se ispravna poruka o grešci prikazuje dio je ove prakse.

4.1.4. Testiranje graničnih uvjeta

Ova vrsta testiranja (Hung Q. Nguyen 2003) je zapravo ekstenzija ranije spomenutih metoda TOFT i FET testiranja s nekim preklapanjima ali i nekim dodatnim detaljima koje one ne pokrivaju. Sličnosti se najjasnije vide u tome što se kao i FET, ovo testiranje bavi pravilnim radom funkcionalnosti kod nekih graničnih varijabli. Primjer s obrascem najbolje ilustrira tu sličnost. Osim što se provjerava pojava pogreške kod određenih polja, bitna značajka može biti i ograničenje znakova što nužno ne mora pokazati ili izazvati poruku o grešci, no svakako se mora na neki način ograničiti.

4.1.5. Eksplorativno testiranje

Eksplorativno testiranje (Hung Q. Nguyen 2003) širi definiciju testiranja i u priču uvodi iskustvo, znanje i volju za učenjem. Umjesto da se unaprijed određuju testni slučajevi, eksplorativno testiranje proučava softver; simultano se uči i testira. Samim time potrebna je veća kompetencija testera no ovim pristupom često se pronalaze defekti koje nije lako otkriti većinom drugih metoda. Iako se količina dokumentacije i pripreme smanjuje, ovakav pristup koji je usko povezan za intuiciju testera ne može se automatizirati.

4.2. Testiranje sučelja

Uz testiranje funkcionalnosti web aplikacije često se primijete i dovode u pitanje problemi korisničkog sučelja *(eng. User interface; UI)*. Samim time odmah je jasno kako se ova vrsta testiranja rijetko izvodi samostalno te zapravo najbolje komplimentira testiranje funkcionalnosti, eksplorativno i TOFT testiranje. Korisničko sučelje i njegov dizajn može se preispitati sa stajališta dizajnera i stajališta developera. Jedino na takav način možemo efektivno testirati dizajn i njegovu implementaciju u isto vrijeme.

Testiranje **dizajna** vodi brigu o tome da li dizajn pruža jasne smjernice u navigiranju, korisne povratne informacije i općenitu konzistenciju , bilo u jeziku, estetici ili pristupu. Rezultate ove metode testiranja teško je popisati ili rigidno izmjeriti pošto su impresije o dobrom dizajnu ili *'osjećaju'* aplikacije kao i njegovoj jednostavnosti korištenja vrlo subjektivne te vrlo male impresije pojedinaca na njih mogu imati velikih utjecaja. Kod testiranja dizajna traži se konzistentnost estetike, povratnih informacija i interaktivnosti bilo u jednostavnosti korištenja navigacije ili razmještaja određenih komponenti. Sve to se korisnicima najčešće predstavlja u obliku raznih povratnih znakova koje aplikacija pruža, ako su oni nejasni, automatski se narušava jednostavnost korištenja i sama interakcija između korisnika i aplikacije. Zbog toga na umu uvijek treba imati tko su ciljani korisnici. Ako se zna da većina korisnika koji će koristiti našu aplikaciju nema puno informatičkog iskustva, razdvajanje određenih procesa na više ekrana imati će više smisla, dok će za iskusnije i bolje informatički potkovane korisnike obrnuto biti korisnije pošto će htjeti što brže obaviti određenu funkciju, navigiranje kroz više ekrana za njih će biti samo nespretno oduzimanje vremena.

Testiranje **implementacije** (Hung Q. Nguyen 2003) UI-a bavi se funkcijom pojedinih UI-elemenata. Iako određeno svojstvo aplikacije radi kako je zamišljeno, bez funkcionalnog korisničkog sučelja pristupačnost tim svojstvima bit će ograničeno. Vrlo jednostavan primjer ovoga bio bi link koji je određene boje na pozadini čija se boja mijenja. Dizajn je možda konzistentan s estetikom aplikacije i sam link vodi na stranicu koju treba, no u određenim kombinacijama boja taj link je teško vidljiv i njegova pristupačnost je otežana.

4.2.1. Testiranje upotrebljivosti i pristupačnosti

Iako često usko vezano uz UI, testiranje upotrebljivosti i pristupačnosti je ipak odvojena vlastita metoda testiranja pošto se kod UI testiranja samo do određene mjere testira ako je aplikacija uistinu laka za korištenje i to za sve moguće korisnike. Proces testiranja upotrebljivosti, odnosno lakoće korištenja, sastoji se od puno dijelova. Od određivanja zadataka koje korisnici ili testna skupina moraju izvršiti, preko njihova izvršavanja sve do proučavanja i formaliziranja podataka koji se skupe. Ovdje se određuje i mjeri zadovoljstvo korisnika pri korištenju web aplikacije. Česta pitanja koja se u ovoj fazi postavljaju mogu biti je li netko tko prvi puta koristi aplikaciju uspješno izvršio neke jednostavne zadatke? Koliko je lako to izveo? Je li korisnik općenito brzo uspio naučiti koristiti aplikaciju itd.

Korištenje aplikacije mora biti ugodno svima pa se uz ovu metodu nadovezuje i testiranje pristupačnosti. Većinom se to odnosi na mogućnosti navigiranja kroz aplikaciju koristeći neko drugo sredstvo osim miša, na primjer korištenje samo tipkovnice ili bilo kakvog vanjskog specijaliziranog uređaja za unos podataka. Obje metode se bave sličnim ciljem a to je lakoća uporabe, no testiranjem pristupačnosti uspijeva se isto postići za širu publiku time što se uključuje ljude s posebnim potrebama/invaliditetom.

4.3. Testiranje kompatibilnosti

Nastavljajući tezu kako osim funkcionalnosti aplikacija mora biti sposobna za još mnogo toga kako bi bila kompletni proizvod, dolazimo i do priče o njenoj kompatibilnosti. Naime, osim što aplikacija radi ispravno ona mora nastaviti raditi ispravno na bilo kojem od podržanih okruženja, neovisno o vremenu ili mjestu. Kod web aplikacije testirati kompatibilnost je višestruko teže nego kod recimo desktop aplikacija, pošto se kod web aplikacija treba uzeti u obzir puno više komponenti i distribuiranih arhitektura. Za jedan web sustav koji se sastoji od servera, baze podataka, web preglednika i mnogih drugih dijelova za svaki od njih moraju se razmotriti moguće nekompatibilnosti.

Pri početku izrade web aplikacije najbitnije je odrediti performanse i minimalne sistemske zahtjeve za uređaj na kojem će se aplikacija koristiti. Tek nakon toga u obzir dolaze web serveri, vatrozid, hardver, preglednik, operativni sustav itd. Svi testovi vezani za kompatibilnost fokusiraju se na probleme do kojih može doći prilikom promjene bilo koje od ovih konfiguracija. Nažalost, gotovo je nemoguće isprobati sve kombinacije do kojih može doći s raznim promjenama okruženja, zato se većinom uzimaju u obzir najkorištenije kombinacije ili najbolje dokumentirane. Još jedna bitna napomena kod ove vrste testiranja je da je vrlo poželjno krenuti ocjenjivati kompatibilnost aplikacije tek nakon što je većina funkcionalnosti utvrđena i testirana, u suprotnome bilo bi vrlo teško raspoznati koje od grešaka su uzrokovane zbog loše kompatibilnosti s okruženjem a koje s funkcionalnošću aplikacije.

4.4. Testiranje performansi

Najbitnija značajka web aplikacija je ta što omogućuje da istu uslugu pružimo velikom broju korisnika u isto vrijeme. Uz to također se javlja i velika odgovornost da taj isti sustav može izdržati takvo opterećenje. Testiranje performansi (*eng. Performance testing*) se odnosi upravo na to, vrijeme potrebno da se izvrši neka funkcija. Bilo to učitavanje neke stranice ili procesiranje uplate - testiranje performansi i opterećenja neizostavni su ako želimo da naša aplikacija može izdržati velike količine protoka informacija i služiti velikom broju korisnika unutar razumnih vremenskih okvira. Osim trenutnih potreba, bitno je obratiti pažnju i na skalabilnost aplikacije, ima li aplikacija mogućnosti prilagoditi se povećanju broja korisnika ili drugih opterećenja ?

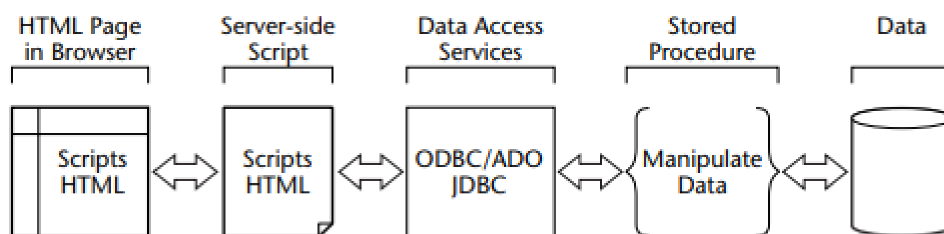
Jedna od najbitnijih značajki ove vrste testiranja je vrijeme odgovora (*eng. Response time*). Vrijeme odgovora je po ustaljenoj definiciji proteklo vrijeme između podnesenog zahtjeva i početka njegova odgovora. Vrlo je jednostavno zaključiti da manje čekanje na odgovor od strane aplikacije znači veće zadovoljstvo korisnika. Usko vezano i ponekad pogrešno korišteno kao sinonim je i latencija (*eng. Latency*). Bilo da govorimo o latenciji mreže ili servera, u oba slučaja mjeri se vrijeme obrade zahtjeva poslanog od strane korisnika. Latencija također može biti vezana i uz specifične uređaje, na primjer uz usmjerivače, koji mogu obraditi fiksne količine podataka po sekundi. Latencija označuje koliko vremena je potrebno da se informacija pošalje ako je taj broj podataka po sekundi nadmašen.

Sve ovo potrebno je da se odredi prag nakon kojeg aplikacija više nema dovoljno resursa da odgovori na zahtjeve pravovremeno. Taj prag posljedično omogućuje izvesti varijacije korisnih stresnih situacija i testova kako bi se otkrile slabe točke i moguća mjesta za napredak. Ako se zna da određena aplikacija ima dovoljno resursa za obraditi 1000 zahtjeva u sekundi, stres testiranjem će se pogurati daleko preko te brojke i analizirati ponašanje koje slijedi. Izuzetno je teško točno odrediti i predvidjeti količine opterećenja iz dana u dan pošto ta opterećenja često variraju kroz dane pa čak i sate. Upravo zbog toga je ova metoda testiranja jedna od najtežih za pravilno izvršavanje i greške pronađene su puno teže za rješavanje i optimiziranje.

4.5. Testiranje baze podataka

Svaka aplikacija će gotovo uvijek koristiti nekakve podatke koje na neki način trebamo pohraniti. Baze podataka u bilo kojem od svojih oblika služe upravo toj svrsi - spremanju potrebnih podataka kako bi ih web aplikacija onda mogla obrađivati i prikazati. No kao sa svakim dijelom web aplikacije i kod baza podataka može doći do grešaka. U radu s bazama podataka mjesta na kojima može doći do grešaka ima poprilično puno te je time i kvalitetno testiranje teško za postići.

Pošto u dohvaćanju i slanju informacija sudjeluje više komponenti, kod svake poveznice između njih može doći do greške – klijentska strana, isto tako i programi i skripte sa server strane, servisi za pristup bazi podataka te sami podaci spremljeni u bazi podataka. Zbog toga je potrebno na svakoj od tih poveznica i točaka između različitih servisa obaviti određena testiranja.



Slika 2 poveznice između raznih elemenata uključenih u spremanje i dobavljanje podataka iz baze podataka (Hung Q. Nguyen 2003)

Većinu grešaka povezanih s bazama podataka možemo odvojiti u dvije grupe: integritet podataka (*eng. Data integrity*) i greške izlaznih podataka (*eng. Output errors*) . Kod integriteta podataka i sličnih situacija problem je s podacima koji se upisuju u bazu. Oni su ili krivi ili zapisani na kriva mjesta. Kod izlaznih podataka greške su vezane uz načine dohvaćanja ili manipuliranja s podatcima iako je sam izvor podataka točan. Oba tipa greški manifestiraju se slično te se jedino s *black-box* pristupom testiranja može uspješno odrediti koji je uzrok loše prikazanih podataka.

4.6. Testiranje sigurnosti

Niti jedna greška u drugim dijelovima web aplikacije ne može se mjeriti s razornošću koju može prouzročiti neriješena sigurnosna greška. Upravo zbog toga najopasnije i najbitnije područje koje svakako treba pokriti kod testiranja je sigurnosti (Hung Q. Nguyen 2003). Ovo područje zahtijeva široko znanje i stručnost izvan spektra znanja koje ima prosječni tester softvera. Iako vrlo bitno, gotovo je nemoguće aplikaciju učiniti 100% sigurnom. Zbog toga je potrebno analizirati i pokušati odrediti minimalni potrebni prag sigurnosti koji je prihvatljiv za korisnike i rad aplikacije. Taj prag se uglavnom pronalazi na mjestu gdje bi trošak za dodatnim mjerama sigurnosti nadmašio troškove izazvane potencijalnim sigurnosnim problemima koje ne rješava već postojeći sigurnosni sustav.

Neke od glavnih stavki kojima se treba baviti kod testiranja sigurnosti većinom se tiču aplikacijskog nivoa. Slabosti i curenje podataka (*eng. Data leaks*) povezanih sa programerskom praksom ili lošom konfiguracijom web servera neki su od njih. Ponovo, sve počinje od početnih zahtjeva klijenta. Ovisno o njima određuje se potreba i nivo sigurnosti. U slučaju male interne aplikacije s povjerljivim informacijama sigurnosni nivo mora biti daleko veći nego kod velike, *open source* aplikacije čiji je gotovo svaki dio dostupan velikom broju ljudi na dnevnoj bazi.

Nadalje, bitno pitanje je kontrola pristupa (*eng. Access control*), tko sve smije i može pristupiti aplikaciji? Postoje li posebne uloge koje dopuštaju određene privilegije ili pristupe? Osim specifikacija i pristupa bitna tema razgovora o sigurnosti aplikacije je i koje informacije moraju ostati privatne? Nakon određivanja ovih i sličnih pitanja fokus se skreće na konkretnije probleme poput *'backdoors'* ostavljenih u kodu, rukovanje iznimkama (*eng. exception handling*), održavanje lozinki ili pronalaženje mjesta mogućih curenja informacija (poruke o grešci koje otkrivaju imena servera, glavnih direktorija…). Sve ovo samo je početak potrebnog znanja i kompetencija za uspješno i kvalitetno testiranje sigurnosti pa je jasno zašto se u mnogim kompanijama formiraju cijeli timovi posvećeni isključivo tome.

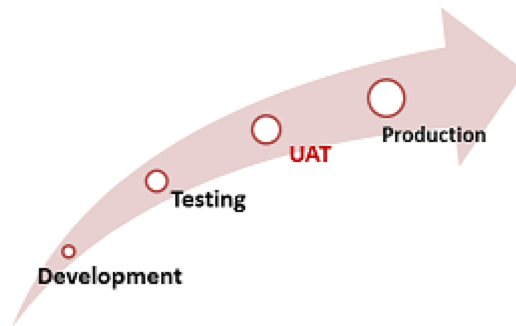
4.7. Masovno testiranje

Masovno testiranje *(eng. Crowd testing*) razlikuje se u tome što sve ranije spomenute metode i područja testiranja se uglavnom izvode od strane posebnog tima unutar kompanije koja proizvodi web aplikaciju. U slučajevima kada postoji jasna ciljna skupina za ili kada je jednostavno jeftinije - testiranje se prepušta određenim skupinama korisnika van kompanije. Jednom testnom timu teško je predvidjeti velike brojeve situacija i okruženja u kojima se aplikacija može koristiti i samim time oni su u stanju testirati samo mali dio. Naravno, aplikacija nikada neće biti u potpunosti 100% testirana no činjenica je da se s povećanim brojem individualnih testera povećava i pokrivenost te se nadmašuje ono što testni timovi mogu. (Global app testing 2022)

Još jedan veliki benefit su sami uređaji na kojima se web aplikacija koristi. Kompanija mora konstantno nabavljati nove uređaje na kojima se treba testirati što ispada vrlo skupo i ne uvijek moguće s najnovijim modelima uređaja. Također, kvaliteta testiranja raste ako jedan test riješi 100 ljudi, naspram slučaja gdje 100 testova riješi jedan čovjek. Uz sve ove benefite naravno da postoje i loše strane *masovnog* testiranja od kojih se odmah nameće problem sigurnosti i povjerljivosti. Teško je dopuštati korištenje aplikacije u razvoju ljudima izvan kompanije pogotovo ako ona sadržava bitne i delikatne informacije. Ne samo to, povjerenje u vanjske testere opada i kada se krene razmišljati o njihovoj kompetenciji da izvrše testiranje ispravno što nitko ne može garantirati.

5. Izrada aplikacije za osiguravanje kvalitete

User acceptance testing ili UAT (Hung Q. Nguyen 2003) je vrsta testiranja koju odrađuje korisnik ili krajnji klijent kako bi se provjerilo ili prihvatilo trenutno stanje aplikacije prije nego ona nastavi dalje u produkciju [].



Slika 3 Položaj odobrenja korisnika u razvojnom tijeku aplikacije (Guru99 2022)

U našem konkretnom primjeru razvojni tijek aplikacije često se 'verzionira', odnosno odvaja u verzije ili izdanja. Aplikacija nikada nije u potpunosti završena nakon prve verzije, naprotiv, imati će bar desetak ako ne i više verzija prije nego postane funkcionalna. Sa svakom novom verzijom definiraju se nove funkcionalnosti i zahtjevi korisnika. Ti zahtjevi dalje se odvajaju u pojedine zadatke (*eng. tasks)*. Te zadatke zatim rješavaju i implementiraju programeri. Prilikom testiranja tih zadataka oni se mogu slagati u takozvane *test suites* koji su samo skup testnih slučajeva koje treba testirati. Prije same produkcije, klijent mora odobriti svaku pojedinu verziju odnosno utvrditi kvalitetu ili nedostatke pojedinih zadataka ili funkcionalnosti. S tim na umu te uz mentorstvo Andreja Arbanasa u sklopu studentskog posla u tvrtki Juice d.o.o. razvila sam web aplikaciju koja olakšava upravo taj proces.

5.1. Korišteni alati

Za razvoj aplikacije korišteni su mnogobrojni alati i razvojni okviri. Nastavku je kratki opis svakog od njih.

5.1.1. Webstorm

WebStorm (JetBrains 2022) je integrirano razvojno okruženje (*eng. Integrated development environment*) proizvedeno od strane *Jet Brains-a*. Kao što su razne verzije razvojnih okruženja u proizvodnji *Jet Brains*-a često specifično napravljene za neki programski jezik tako je i *WebStorm IDE* specifično napravljen za JavaScript. Od *front-end* alata podržava Angular, React i Vue.js, dok za *back-end* i serversku stranu na raspolaganju ima Node.js i Meteor. Osim njih podržava i niz drugih razvojnih okvira i alata. Kao neke od glavnih značajki ističu se:

- mogućnosti automatske nadopune koda,
- pronalaženje i isticanje grešaka u kodu,
- jednostavno povezivanje s Git-om (platforma za verzioniranje koda),
- softver za praćenje izvršenja programa (*eng. Debugger).*

5.1.2. Angular

Angular (Angular 2022) kao razvojni okvir pisan je i baziran na programskom jeziku TypeScript te je izrađen i trenutno održavan od strane Google-a. Održavanje i doprinos nisu ograničeni samo na Google pošto je Angular tip razvojnog okvira otvorenog koda. Glavni ciljevi Angulara su olakšati programiranje (većinom) dinamičkih web aplikacija tako što olakšava rješavanje mnogih problema na koje bismo naišli da iste programiramo u čistom JavaScriptu zbog već gotovih funkcija i metoda različitih vrsta i primjena.

5.1.3. Typescript

Typescript (TypeScript 2022) je programski jezik koji zapravo nadopunjuje i olakšava korištenje JavaScripta. Ukratko, on definira set tipova za JavaScript i pomaže u pisanju i lakšem razumijevanju koda. Najviše se koristi uz Angular (no nikako nužno) pošto nudi bolju strukturu sintakse.

5.1.4. HTML

Kao što nam kratica daje naznačiti HTML (*eng. hypertext markup language*) nije programski jezik već takozvani jezik za označavanje. Koristi svoje mnogobrojne elemente raspoređene u oznakama (*eng. Tags)* kako bi prikazao strukturirane dokumente u pregledniku. (Wikipedia - HTML 2022)

5.1.5. CSS i SCSS

Kao nadogradnja na HTML, CSS (*eng. cascading style sheet)* pomaže u ljepšoj prezentaciji dokumenata napisanih u HTML-u. To postiže odvajanjem prezentacije i sadržaja te uvođenjem raznih rasporeda, boja i fontova. (Wikipedia - CSS 2022)

SCSS *(eng. Syntactically Awesome Style Sheet)* je nadogradnja za CSS koji nam omogućuje više opcija i pomaže kod uređivanja sa *Bootstrap4.* (Sass 2022)

5.1.6. NodeJS

Prebacujući se s *front-end* alata na *back-end* i *server-side* prvo nailazimo na NodeJS. Ovo je besplatna Javascript platforma otvorenog koda. Može se objasniti i kao razvojni okvir koji koristi međuprogramsku opremu kako bi olakšao izradu i upravljanje web serverom. (SimpliLearn 2022)

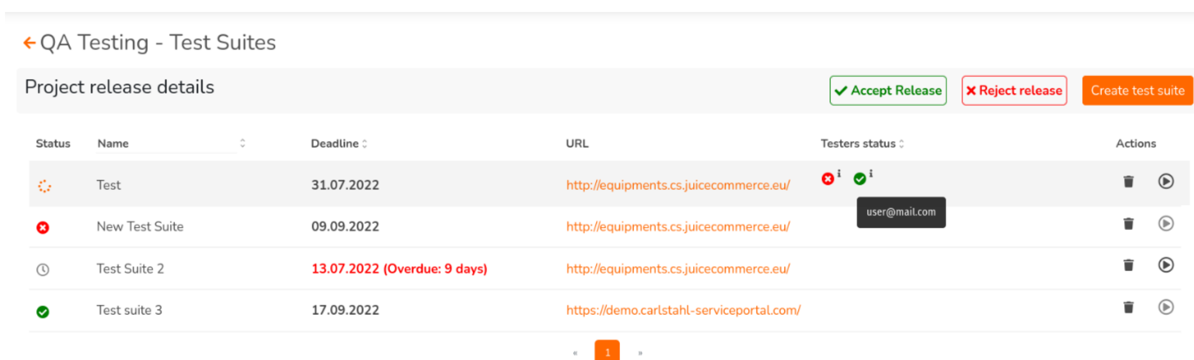
5.1.7. MongoDB

MongoDB (SimpliLearn 2022) je NoSQL program za baze podataka. Nudi velik nivo skalabilnosti i performansi te je prikladan za rad s konceptima poput kolekcija i dokumenata. Ti dokumenti većinom su u JSON ili sličnim formatima, a sam program pisan je u C++ programskom jeziku. Kolekcija u MongoDB bazi podataka je upravo skup dokumenata, dok su sami dokumenti setovi parova ključeva i vrijednost. Dokumenti mogu biti dinamični tj. unutar iste kolekcije različiti dokumenti mogu imati različite setove polja i struktura.

5.2. Načina rada Quality Assurance modula

Osnovni slijed korištenja aplikacije može se podijeliti među dva glavna pogleda: pogled admina, odnosno testera, i korisnika.

Tester ima uvid u listu svih projekata te unutar svakog projekta listu svih verzija koje su podijeljene u dvije kartice. Jedna s verzijama koje su u fazi testiranja (*eng. Active releases*) te druga s pregledom prijašnjih verzija koje su odobrene ili odbijene (*eng. Tested releases*) .
Za svaku od aktivnih verzija može kreirati *test suite* te ispuniti formu s imenom test suita, krajnji rok do kada bi se *test suite* trebao riješiti, odabrati na kojem stadiju se testiranje treba izvršiti te odabrati koji *taskovi* ulaze u taj *test suite*. Opcionalno za svaki od odabranih *taskova* može dodati opis radi lakšeg razumijevanja.
Nakon što je *test suite* kreiran, isti može riješiti tester, no još bitnije, sada svaki korisnik koji ima pristup određenoj organizaciji čijoj pripada projekt, također može riješiti *test suite*.
Sve dok nitko nije krenuo sa rješavanjem *test suita*, tester ga može ažurirati (dodavati *taskove*, mijenjati rokove, dodavati opise i medije za pojedine taskove…) ili obrisati. Osim svih početnih informacija na stranici pregleda svih *test suites*, tester također može vidjeti i statuse korisnika u obliku ikona koje mogu označavati je li korisnik u stanju trenutnog testiranja, je li odbio ili pak prihvatio verziju. Klikom na pojedinu ikonu korisnika otvara se *'reports tab'* u kojem tester može pogledati točno koji *taskovi* nisu prošli i zašto.



Slika 4 Pogled Testera

Klijent o svakom kreiranom *test suitu* za projekt u čiji ima uvid dobiva poruku na email kako bi znao da treba provjeriti aplikaciju i krenuti s testiranjem. Otvaranjem aplikacije njegov pogled se sačinjava od liste *test suites* koje mora riješiti te *switch* gumba koji u određenoj poziciji prikazuje sve završene odnosno riješene *test suites*. Lista trenutnih *test suites* prikazuje trenutni status pojedinog *test suita* koji može biti u stanju *pending* ako *test suite* nije započet, *in progress* ako je započet, *passed* ako je završen i odobren ili *rejected* ako je završen i odbijen. Nadalje, ima prikaz roka koji ako je prošao obavještava o zakašnjenju. Pokraj imena projekta, verzije i *test suita* također ima uvid i u napredak odnosno količinu riješenih testova unutar *test suita*. Naposljetku nalazi se gumb koji dalje vodi na pogled koji dijeli s testerom odnosno konkretno rješavanje *test suita*.

QA Testing - Test Suites

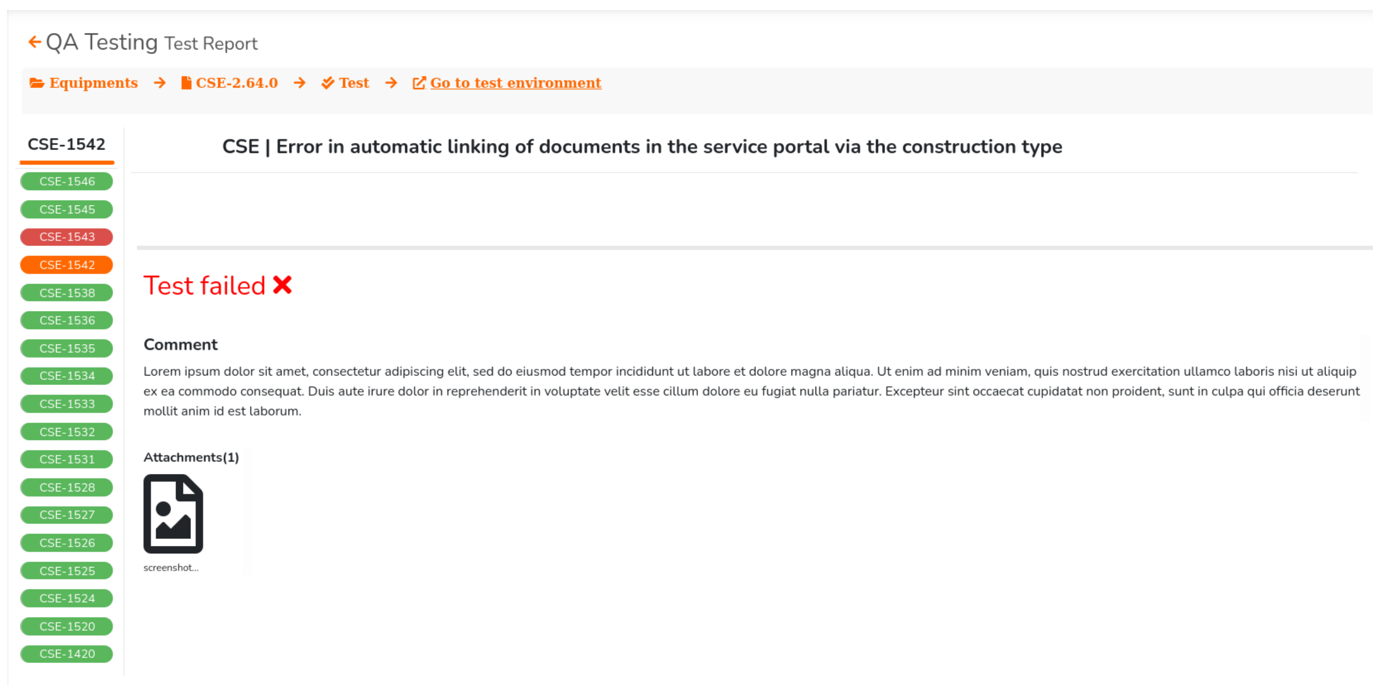
Test suites

Show finished test suites

Status	Deadline	End	Project	Release	Name	Progress	Action
	09.09.2022		Equipments - Staging	CSE-2.64...	Short Test Suite	1/3	
	13.07.2022 (Overdue: 9 days)		Equipments - Development	CSE-2.64...	Test Suite 2		
	31.07.2022		Equipments - Development	CSE-2.64...	Test	4/18	
	18.06.2022 (Overdue: 3 days)		Equipments - Staging	CSE-2.60...	late for 2		
	30.06.2022 (Overdue: 22 days)		Equipments - Development	CSE-2.60...	dddd		
	03.07.2022 (Overdue: 19 days)		Equipments - Development	CSE-2.60...	Image		
	20.08.2022		Equipments - Production	CSE-2.60...	Count ended test		
	19.11.2022	15.06.2022	Equipments - Production	CSE-2.60...	Image1	19/19	
	09.09.2022	19.07.2022	Equipments - Development	CSE-2.64...	New Test Suite	9/9	
	17.09.2022	22.07.2022	Equipments - Production	CSE-2.64...	Test suite 3	18/18	

Slika 5 Pogled klijenta

I tester i klijent rješavaju *test suite* na način da prolaze kroz zadane testove te imaju prikazan opis svakog pojedinog testa zajedno s priloženim medijima. Ako je test prošao pritiskom na '*passed'* gumb, otvara se sljedeći *task*. U slučaju da *task* ne funkcionira kako treba, treba se upisati komentar od minimalno 50 znakova kako bi se mogao kliknuti gumb '*failed'*. Istovremeno gumb '*passed'* postaje onemogućen za korištenje. Osim tog uvjeta ako korisnik (ili tester) dodaje medij, također se podrazumijeva da je test neuspješan te se gumb '*passed'* više ne može kliknuti. Tek nakon što su svi *taskovi* testirani, omogućuje se klik na gumb '*complete test suite'* koji završava testiranje i vraća korisnika na prethodni pogled.



Slika 6 Prikaz pregleda završenog test suita

QA Testing Run tests
Equipments → CSE-2.64.0 → Short Test Suite → Go to test environment
1/3
CSE-1545
CSE-1543
CSE-1534
Complete Test Suite
CSE-1543 Export needed data from database
Add comment
Minimum 50 characters must be written
0/50
Add media
Add media
Name Media Type Type File Name Actions
No data to display
Passed Failed

Slika 7 Pogled rješavanja test suite kojeg dijele i tester i klijent

5.3. Pojašnjenje koda *Quality Assurance* modula

Pošto je 'QA modul' dio mnogo veće aplikacije i povezan s puno drugih komponenti, zadržati ću se na pojašnjavanju par glavnih dijelova i funkcionalnosti na kojima sam osobno i samostalno radila.

U svom najjednostavnijem obliku, projekt se može podijeliti na tri dijela: komponente, modeli i servisi.

Komponente su zapravo ono što sačinjava različite poglede naše aplikacije i njihove poveznice s logikom aplikacije. Modeli su datoteke u kojima određujemo 'izgled' naše kolekcije podataka u bazi (Slika 13 TestSuite.ts). Servisi su unutar Angular razvojnog okvira uglavnom klase kojima komponente pridaju određene zadaće, poput dohvaćanja podataka sa servera, validacije unosa, itd. Bitan dio servisa je *@Injectable* tag kojim taj servis sa svim svojim funkcijama postaje dostupan bilo kojoj komponenti unutar aplikacije. (Slika 12 QualityAssuranceService.ts)

Slika 8 Odnos 'injektiranja' servisa u komponentu (Angular 2022)

Za početnu klasu *QaTestingComponent* (Slika 9 qa-testing.component.ts) dodajemo *@Component* dekorator kojim naznačujemo da je to Angular komponenta. Samim time možemo lakše upravljati kako će se ta komponenta pokretati, procesirati itd.. Ova komponenta će biti naša baza ili takozvani '*parent*' aplikacije, a ostale komponente njeni '*child*'-ovi (Slika 10 qa-testing.route.module.ts).

```
import { Component, OnInit } from '@angular/core';
import {JuiceboxService} from '../../services/Juicebox.service';

@Component({
  selector: 'app-tests',
  templateUrl: './qa-testing.component.html',
  styleUrls: ['./qa-testing.component.scss']
})
export class QaTestingComponent implements OnInit {

  constructor(public juicebox: JuiceboxService) {}

  ngOnInit() {

  }

}
```

Slika 9 qa-testing.component.ts

```
(…)
const QaTestingRoute: Routes = [{
        path: '',
        component: QaTestingComponent,
        children: [
            {
                path: '',
                component: QaTestingListingComponent,
                redirectTo: 'project',
                canActivateChild: [AdminGuard]
            },
            {
                path: 'project',
                component: ProjectComponent,
                canActivate: [AdminGuard]
            },
(…)
```

Slika 10 qa-testing.route.module.ts

Pojednostavljeno, glavni zadatak servisa je da kreira jednu instancu klase koja se potom injektira u bilo koju drugu klasu koja to zatraži. Ovaj način zahtijeva takozvanog *provider*-a koji 'nabavlja' tu instancu komponentama unutar istog modula (Slika 11 qa-testing.module.ts).

```
import { NgModule } from '@angular/core';
(…)
@NgModule({
(…)
    providers: [
        QualityAssuranceService,
        QATestingTranslationPipe,
        AdminGuard
    ],
    (…)
})
export class QaTestingModule {

}
```

Slika 11 qa-testing.module.ts

```
import { Injectable } from '@angular/core';
import { Juice } from '../../../services/juice.service';
import {ISort} from '../../../interfaces/ISort';
import {ISearchTerm} from '../../../interfaces/ISearchTerm';
import { Result } from '../../../types/Result';
import { TestResult } from '../models/TestSuite';

@Injectable()
export class QualityAssuranceService {

    constructor(private juice: Juice) {
    }
(…)
fetchTestSuites(release_id: string, page: number, pageSize: number, options?:
{ sort: ISort, filter: Array<ISearchTerm>}): Promise<any> {
        return this.juice.request(
            "test:suite",
            "fetchTestSuites",
            [ release_id, page, pageSize, options]
        )
    }

    fetchTestSuiteById(testSuite_id): Promise<any> {
        return this.juice.request(
            "test:suite",
            "fetchTestSuiteById",
            [testSuite_id]
        )
    }

    deleteTestSuite(testSuite_id): Promise<any> {
        return this.juice.request(
            "test:suite",
            "deleteTestSuite",
            [testSuite_id]
        )
    }

    updateTestSuite(testSuite_id, testSuite): Promise<any> {
        return this.juice.request(
            "test:suite",
            "updateTestSuite",
            [testSuite_id, testSuite]
        )
    }
(…)
```
Slika 12 QualityAssuranceService.ts

```
export class TestSuite {

    _id: string;

    name: string;
    deadline: Date;

    //References
    release_id: string;
    project_id: string;
    application_id?: string;
    _projectName: string;
    _releaseName: string;

    stage: Stage;

    tests: Test[];

    url?: string;

    userTestAttempts: {
        [user_id: string]: {
            start: Date,
            end: Date,
            results: TestResult[]
        }
    };

    type?: any;

    media?: Array<any>;
}

export enum Stage {
    DEV = "Development",
    STAGING = "Staging",
    PROD = "Production"
}

export type Test = {
    key: string,
    name: string,
    description?: string
```

Slika 13 TestSuite.ts

Nakon postavljanja glavnih dijelova aplikacije kreiramo komponente koje se sastoje od TypeScript dokumenta s logikom i instancama servisa, HTML dokumenta s općenitim izgledom dokumenta i SCSS dokumenta za uređivanje njegova izgleda.

Za primjer uzeti ću ekran kojeg vidi korisnik s admin ulogom koji je vezan za dodavanje novog '*test suite'*-a.

Prilikom inicijalizacije *(ngOnInit())* pokrećemo dohvaćanje svih zadataka unutar trenutno odabranog izdanja te kreiranje takozvanog *FormGrup-a (*Slika 14 Kreiranje form grupe unutar select-add-test.component.ts*)*. Forme su još jedan vrlo koristan dio Angulara koji nam omogućuje upravljanje formama i njenim unosima čije se vrijednosti mogu mijenjati unutar nekog vremenskog perioda (Angular 2022). *Form group* je način za grupiranje pojedinih *form kontrola*, a svaka *form kontrola* je jedan unos. Kasnije se u HTML datoteci ovaj *form* može prikazati preko <form *[formGroup]="imeFormGrupe"*> oznake (Slika 16 form oznaka unutar select-add-test.component.html). Osim *form grupe* postoji i *FormArray* koji također grupira *form kontrole* no u obliku liste čije članove možemo dinamički dodavati i micati. Na ovaj način ne moramo definirati i imenovati svaku pojedinu *form kontrolu* pa je koristan kada imamo veliki i nedefinirani broj vrijednosti. U ovom primjeru *FormArray* nam je dobro došao u kreiranju potvrdnih okvira *(eng. Checkbox)* za odabir koji zadatci će ući u *test suite* koji kreiramo.

```
 get issuesFormArray() {
        return this.testSuiteForm.controls.tests as FormArray;
    }

    createFormGroup() {
        this.testSuiteForm = new FormGroup({
            name: new FormControl(this.testSuite ? this.testSuite.name : null, 
Validators.required),
            deadline: new FormControl(this.testSuite ? this.helper.dateForDa-
tepicker(this.testSuite.deadline) : null, Validators.required),
            stage: new FormControl(this.testSuite?.stage ?? null, Valida-
tors.required),
            tests: new FormArray([]),
        });
        this.addCheckboxes();
    }

    private addCheckboxes() {
        if (this.update) {
            this.testsData.forEach(test => this.issuesFormArray.push(new 
FormControl(this.testSuite.tests.findIndex(t => t.key === test.key) > -1)));
            this.testSuite.tests.forEach(t => this.additionalInfoMap[t.key] = 
{ description: t.description });
            //for every test from testsData (all tests) make a form control 
which will be true IF the test from testSuite (all checked tests) has the same 
key
        } else {
            this.testsData.forEach(() => this.issuesFormArray.push(new 
FormControl(true)));
        }
    }
```

Slika 14 Kreiranje form grupe unutar select-add-test.component.ts

Slika 15 Ekran za dodavanje test suite-a sa opisom odnosa form group i form control-a

```
<form [formGroup]="testSuiteForm">
       <div class="form-group">
           <label>{{'name'|translate}}*</label>
           <input class="form-control" type="text" placeholder="Name"
                  formControlName="name">
           <div *ngIf="testSuiteForm.controls['name'].hasError('required') &&
testSuiteForm.controls['name'].touched"
                class="alert alert-danger">You must include a name.
           </div>
       </div>
   </form>
```

Slika 16 form oznaka unutar select-add-test.component.html

Kod inicijalizacije pokrenuli smo i dohvaćanje svih zadataka odnosno *taskova*. Nakon što smo u konstruktoru definirali naš servis pod oznakom 'qa' sada ga možemo koristiti kao i sve funkcije unutar tog servisa. Konkretno, ovdje koristimo funkciju *getReleaseById* koja nam vraća izdanje koje odgovara proslijeđenim varijablama. Nakon toga iz dohvaćenih podataka uzimamo samo '*issues'* kao listu zadataka i spremamo u varijablu *testsData* (Slika 17 Dohvaćanje zadataka unutar select-add-test.component.ts*)*.

```
//constructor (private qa: QualityAssuranceService
async fetchAllTests() {
        const result = await this.qa.getReleaseById(this.release_id, { de-
mands: true });
        if (!result) {
            this.juicebox.showToast('error', 'No issues', result.error);
            return false;
        }

        if (result.issues == undefined)
            return false;

        this.testsData = result.issues;
        if (result.demands?.length)
            this.testsData.push(...result.demands.map(_d => {
                _d.summary = _d.description;
                _d.type = "SD";
                return _d;
            }));
```

Slika 17 Dohvaćanje zadataka unutar select-add-test.component.ts

```
getReleaseById(release_id, options?: any): Promise<any> {
        return  this.juice.request(
            "releases",
            "get",
            [release_id, options]
        );
    }
```

Slika 18 Funkcija getReleaseById unutar QualityAssuranceService.ts

```
@Remotable(["objectid", "json"],
new RemotableUserSessionFilter("releases:role"))
    async get(release_id: ObjectId, options?: any): Promise<Release> {
        const release = await Release.findOne<Release>({ _id: release_id });
(…)
        return release;
    }
```

Slika 19 get funkcija na back-end strani za dohvaćanje izdanja

Kod prikazivanja dohvaćenih zadataka dolazimo do još jedne komponente Angulara a to je *ngx-datatable*. Ovaj format tablice uvelike olakšava prikaz, sortiranje i filtriranje sadržaja. Za njeno korištenje dovoljna nam je jednostavna *<ngx-datatable>* oznaka. Tablica se sastoji od *<ngx-datatable-column>* oznaka od kojih svaka zatim ima *<ng-template>* dio (Slika 20 Primjer ngx-datatable unutar select-add-test.component.html).

```
<ngx-datatable class="bootstrap fullscreen"
                           [rows]="issuesFormArray.controls"
                           [columns]="columns"
                           [columnMode]="'flex'"
                           [columnMode]="'force'"
                           [externalSorting]="true"
                           [sorts]="[{ prop: 'index', dir: 'asc' }]">

                <ngx-datatable-column prop='key' name="{{ 'key' | translate
}}" [width]="10">
                    <ng-template let-rowIndex="rowIndex" ngx-datatable-cell-
template>
                        <span style="width: 100%"><strong
                            *ngIf="testsData[rowIndex].type === 'SD'">[Servi-
ceDesk] </strong>{{ testsData[rowIndex].key }}</span>
                    </ng-template>
                </ngx-datatable-column>
            </ngx-datatable>
```

Slika 20 Primjer ngx-datatable unutar select-add-test.component.html

Većina ostalih pogleda sastoji se od istih komponenata i radi na vrlo sličan način te bi njihovo daljnje objašnjavanje bilo ponavljanje uglavnom temeljno istog koda s manjim izmjenama.

6. Zaključak

Retrospekcijom na ideju i izvedbu testiranja jasno je da se kao jedan od standardnih dijelova u životnom ciklusu web aplikacije testiranje nastavlja rapidno razvijati. Od skromnih početaka vezanih za industriju i fizičke objekte do današnjih apstraktnih specifikacija unutar raznih web servisa, testiranje se prilagođavalo i razvijalo jednakom brzinom kao i sve tehnologije uz koje je vezano. Odvajanje i sistematizacija raznih tehnika i metoda potrebna je radi lakše organizacije testiranja no u stvarnim primjenama jedna ili dvije tehnike gotovo nikad nisu dovoljne te pravi testni stručnjaci gotovo će uvijek koristiti više njih i to simultano i isprepleteno.

Kroz primjer aplikacije za testiranje koji je dan u završnom radu, još je jasnije koliko je testiranje integrirano i bitno za kvalitetnu isporuku web aplikacije kao proizvoda i zadovoljstvo klijenta i korisnika. Ne samo da je bitno uključiti klijente u razvoj web aplikacije, već je bitno uključiti ih u svaki pojedini korak koji sačinjava taj razvoj - pa i testiranje.

7. Popis Slika

Slika 1 Hijerarhija testiranja, osiguravanja kvalitete i inženjerstva kvalitete softvera 6
Slika 2 poveznice između raznih elemenata uključenih u spremanje i dobavljanje podataka iz baze podataka (Hung Q. Nguyen 2003) 17
Slika 3 Položaj odobrenja korisnika u razvojnom tijeku aplikacije (Guru99 2022)............... 20
Slika 4 Pogled Testera.. 23
Slika 5 Pogled klijenta.. 24
Slika 6 Prikaz pregleda završenog test suita... 25
Slika 7 Pogled riješavanja test suite kojeg dijele i tester i klijent 25
Slika 8 Odnos 'injektiranja' servisa u komponentu (Angular 2022) 26
Slika 9 qa-testing.component.ts 27
Slika 10 qa-testing.route.module.ts.. 27
Slika 11 qa-testing.module.ts.. 28
Slika 12 QualityAssuranceService.ts 29
Slika 13 TestSuite.ts... 30
Slika 14 Kreiranje form grupe unutar select-add-test.component.ts.............................. 32
Slika 15 Ekran za dodavanje test suite-a sa opisom odnosa form group i form control-a 33
Slika 16 form oznaka unutar select-add-test.component.html 33
Slika 17 Dohvaćanje zadataka unutar select-add-test.component.ts 34
Slika 18 Funkcija getReleaseById unutar QualityAssuranceService.ts............................ 34
Slika 19 get funkcija na back-end strani za dohvaćanje izdanja 34
Slika 20 Primjer ngx-datatable unutar select-add-test.component.html 35

8. Literatura

1. *Angular*. 2022. https://angular.io/ (pokušaj pristupa 7. Srpanj 2022).
2. *Global app testing*. 2022. https://www.globalapptesting.com/blog/crowdsourced-testing (pokušaj pristupa 6. Srpanj 2022).
3. *Guru99*. 2022. https://www.guru99.com/user-acceptance-testing.html (pokušaj pristupa 6. Srpanj 2022).
4. Hung Q. Nguyen, Bob Johnson, Michael Hackett, Robert Johnson. *Testing Applications on the Web: Test Planning for Mobile and Internet-Based Systems, Second Edition.* Wiley Publishing, 2003.
5. *JetBrains*. 2022. https://www.jetbrains.com/webstorm/ (pokušaj pristupa 4. Srpanj 2022).
6. *Sass*. 2022. https://sass-lang.com/ (pokušaj pristupa 5. Srpanj 2022).
7. *SimpliLearn*. 2022. https://www.simplilearn.com/tutorials/nodejs-tutorial (pokušaj pristupa 7. Srpanj 2022).
8. Tian, Jeff. *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement.* Wiley-IEEE Computer Society Press, 2005.
9. *TypeScript*. 2022. https://www.typescriptlang.org/ (pokušaj pristupa 7. Srpanj 2022).
10. *Webomates*. 2022. https://www.webomates.com/blog/software-testing/evolution-of-software-testing/ (accessed Srpanj 5, 2022).
11. *Wikipedia - CSS*. 2022. https://en.wikipedia.org/wiki/CSS (pokušaj pristupa 7. Srpanj 2022).
12. *Wikipedia - HTML*. 2022. https://en.wikipedia.org/wiki/HTML (pokušaj pristupa 7. Srpanj 2022).