Usporedba testiranja performansi na strani klijenta i servera s praktičnim primjerima

Pintarić, Luka

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
V A R A Ž D I N

Luka Pintarić

Usporedba testiranja performansi na strani klijenta i servera s praktičnim primjerima

DIPLOMSKI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Luka Pintarić

Matični broj: 44892/16–R

Studij: Informacijsko i programsko inženjerstvo

Usporedba testiranja performansi na strani klijenta i servera s praktičnim primjerima

DIPLOMSKI RAD

Mentor/Mentorica:

Dr. sc. Marko Mijač

Varaždin, Veljača 2024.

Luka Pintarić

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Tema ovog rada odnosi se na testiranje softverskih performansi web aplikacija na klijentskoj i serverskoj strani. Rad započinje sa teoretskom podlogom u kojoj je opisano što su to performanse softvera, što je testiranje performansi, koje vrste takvog testiranja postoje i zašto je ono jedno od ključnih stavki kod razvoja softvera. Zatim slijede principi i tipovi testiranja performansi te aktivnosti koje se izvršavaju tijekom samog procesa testiranja. Teoretski dio završava razlikama u testiranju na serverskoj i klijentskoj strani te tehnologijama koje su dostupne za oba pristupa testiranja. U drugom djelu rada temeljem odabranih alata opisani su praktični primjeri testiranja performansi na serverskoj i klijentskoj strani. Također, kreiran je model usporedbe koji temeljem obrađenog teoretskog znanja i rezultata testova upućuje na sličnosti i razlike oba pristupa testiranja. Cilj ovog rada je objasniti što je to testiranje performansi softvera, koje su razlike klijentskog i serverskog testiranja te zašto su oba pristupa važna.

Ključne riječi: softver, testiranje, performanse, server, klijent

Sadržaj

1. Uvod

Web tehnologija u današnje vrijeme dosegla je iznimnu razinu što možemo vidjeti po broju raznih popularnih i korisnih web mjesta i aplikacija. Da bi web mjesto bilo uspješno mora imati neku svrhu i pozitivan efekt na korisničko iskustvo. Korisničko iskustvo uglavnom se odnosi na zadovoljstvo korisnika prilikom korištenja nekog web mjesta. Zadovoljstvo korisnika primarno se temelji na ponudi i broju funkcionalnosti u skladu sa svrhom web mjesta. Iako se neko web mjesto isprva može činiti kvalitetnim, temeljem funkcionalnosti i možda izgleda, postoje i druge osobine koje mogu utjecati na uspješnost. Gotovo je sigurno da smo svi ponekad tijekom dužeg korištenja nekog web mjesta primijetili da kretanje nije dovoljno fluidno, odnosno učitavanje novih web stranica je sporije od očekivanog. Takvu pojavu povezujemo sa performansama web mjesta, koje ovisno odazivu web mjesta mogu biti dobre ili loše. (Molyneaux, 2009)

Faktori koji mogu utjecati na performanse su brojni. Prvo na što bismo pomislili jest nestabilna i loša internetska konekcija korisnika ili zauzetost korisničkog računala drugim procesima koji bi tako usporili učitavanje sadržaja web mjesta. Suprotan slučaj ovome jest da odgovor od strane web mjesta traje duže zbog dužeg procesiranja zahtjeva. Takvo što može se desiti ako za web mjesto nisu predviđene neke rubne situacije, npr.:

- Rasprodaja ili crni petak na popularno web shopu. U to vrijeme web mjesto će primiti veliku količinu zahtjeva od strane korisnika, te će zbog nemogućnosti obrade odaziv biti uvećan, a ponekad se i samo web mjesto može srušiti i postati nedostupno.
- Do loših performansi može doći kod izdanja nove verzije web mjesta. Ponekad u novoj verziji zbog nedovoljno dobrog testiranja, propusti dolaze do produkcije i stvaraju duže obrade zahtjeva klijenata ili čak dolazi do grešaka pa se neki dijelovi web mjesta ni ne učitavaju.

Dakle, možemo zaključiti da loše performanse utječu na zadovoljstvo korisnika, a prema Google-ovom istraživanju iz 2017. (https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/) to je jedan od glavnih razloga da korisnik napusti i prestane koristiti web mjesto. No, loše ili dobre performanse utječu na mnoštvo drugih faktora uspješnosti (Mohan, 2022):

- Gubitak korisnika znači i pad promocije i prodaje što može utjecati na poslovanje tvrtke vezane uz web mjesto. 2018. američka tvrtka Amazon izgubila je između 72-

99 miliona dolara prihoda jer njihov web shop nije bio u mogućnosti podnijeti količinu korisničkih zahtjeva na dan *Prime Day* događaja. U ovakvom slučaju loše performanse mogu utjecati i na samu reputaciju tvrtke jer loše vijesti se šire brzinom svijetlosti u globalnom svijetu.

- Loše performanse mjerene su od strane raznih web pretraživačkih mašina kao što je *google.com*. Temeljem tih mjerenja Googleovi pretraživački optimizacijski algoritmi rangiranju lošije web mjesta loših performansi. Lošije rangiranje automatski znači da će posjećenost mjesta uvelike pasti i ponovo dovesti do financijskog deficita vlasnika web mjesta.
- Suprotno u odnosu na prvu točku, poboljšanje performansi može pridonijeti značajnom povećanju prihoda. Primjerice u 2016., britanska željeznička kompanija *The Tarinline* smanjila je prosječno učitavanje stranice za 0.3 sekunde i godišnji prihodi povećali su se za 8 miliona funti. Dugi primjer je tvrtka za *frontend* razvoj web aplikacija *Mobify* smanjenjem učitavanja samo početne stranice njihovog web mjesta za 0.1 sekundu povećala je prihode za 380 tisuća britanskih funti. Ovo ukazuje da poboljšanje performansi web mjesta pridonosi povećanju profita i većih i manjih tvrtki.

Danas se većina web mjesta pokreće pomoću Cloud servisa. Servisi osim što pokreću web mjesta, pružaju dodatnu podršku za praćenje grešaka, skaliranje, upravljanje novim verzijama, implementaciju testiranja i mnoge druge pogodnosti. Situacija sa web shop mjestom i lošim performansama, koju smo prethodno opisali, uglavnom se može izbjeći ako su konfiguracije servisa za skaliranje podešene pravilno. No, niti dobro skaliranje sofisticiranih Cloud servisa ponekad nije dovoljno ako sama implementacija web mjesta ima kritične propuste. Propuste performansi kod velikih projekata gotovo je nemoguće izbjeći u fazi razvoja novih funkcionalnosti web mjesta, zbog vrlo brzog tempa i kompleksnih zahtjeva klijenata. Zato je potrebno postaviti kvalitetan *CI/CD pipeline* te u njega ukomponirati adekvatne automatizirane testove za mjerenje performansi. Na taj način prije nego novo izdanje web mjesta dospije do produkcije, funkcionalnosti se testiraju i na taj način se mogu uočiti i otkloniti eventualni propusti u kodu, konfiguracijama i ostalim segmentima.

Također, bitno je napomenuti da kod web razvoja propusti u kodu nisu samo vezani uz *backend* kod koji se izvršava na strani servera. One se mogu desiti i na *frontend* ili klijentskoj strani. *Backend* kod striktno je vezan uz dohvaćanje sadržaja, izvršavanje poslovne logike, rad sa bazom podataka, povezivanje s mikro servisima i naravno propustima koji se mogu desiti u tim područjima. Sa druge strane *frontend* kod obuhvaća sve klijentske skripte koje su potrebne

web pregledniku kako bi dohvaćene podatke sa *backenda* prikazao na ekranu korisnika. U tim skriptama također može doći do propusta u logici, te izazvati neželjene poteškoće kod performansi web mjesta.

Stoga, u nastavku ovog rada krećemo sa opisivanjem osnovnih koncepata i značajka testiranja performansi te ćemo detaljno opisati razliku testiranja *backend* i *frontend* koda web mjesta. Nakon toga predstavit ćemo dva alata za testiranje web mjesta na obje strane (*backend/frontend*), izvesti praktični primjer na testnoj aplikaciji namijenjenoj za isprobavanje testova i naposlijetku napraviti usporedbu.

2. Osnovni koncepti i značajke testiranja performansi

2.1. Performanse softvera

Prije nego krenemo na koncept testiranja performansi vrijedilo bi definirati što su to zapravo performanse u kontekstu softvera. *Cambride Dictionary* (Meaning of performance, n.d.) web mjesto navodi nekoliko definicija za performanse:

- Koliko dobro osoba ili stroj obavlja neki zadatak ili aktivnost.
- Performanse se odnose na to koliko dobro je neka aktivnost ili posao izvršen

U kontekstu softvera performanse definiraju koliko efikasno neko softversko rješenje može ispuniti određeni zadatak. U današnje vrijeme softverska rješenja sastoje se od mnogo komponenti i vrlo su kompleksa pa je mjerenje performansi vrlo izazovan posao. (Sipriano, 2021)

Performanse su jedan od najvažnijih faktora koji direktno utječu na iskustvo korisnika tijekom korištenja neke vrste aplikacije, bila to web, desktop ili mobilna aplikacija. Stoga, performanse nisu jedini faktor koji utječe na korisničko iskustvo, već tu ubrajamo neke funkcionalne i nefunkcionalne karakteristike softvera, a o tome ćemo spomenuti nešto više u nastavku. Iz prethodno navedenog možemo zaključiti da testiranje performansi igra vrlo bitnu ulogu u ostvarenju kvalitete softvera i pomaže nam postaviti granicu prihvatljivosti kvalitete s perspektive korisnika. Dakle, zadatak ovog tipa testiranja jest izmjeriti i osigurati dobre performanse, no ovdje se javlja pitanje – kakve su to dobre performanse? Isto tako kod performansi teško je definirati generalno dobre ili loše, jer se one više baziraju na nekoj skali prihvatljivosti te mogu biti više ili manje dobre/loše. (Yorkston, 2021; Shivakumar, 2020)

Iz perspektive korisnika vrlo je jednostavno definirati razinu performansi, primjerice nekog web mjesta. Ako tijekom korištenja korisnik ima osjećaj da je kretanje web mjestom glatko i sadržaj se učitava gotovo trenutačno tada su performanse za korisnika vrlo dobre. Suprotno tome kada su performanse loše, vrlo je dobro pitanje, što je uzrok tome? Razloga može biti više:

- Loša internetska veza, npr. javne Wi-Fi mreže gdje je spojeno mnoštvo drugih uređaja, slab signal korisničkog uređaja itd.
- Fizički kvar korisničkog uređaja.

- Bitan događaj na web mjestu koji privlači mnoštvo dugih korisnika, npr. rasprodaja na web trgovini, izvanredne vijesti na portalu…
- Kvar na servisnoj ili hardverskoj infrastrukturi web mjesta.

Upravo zbog toga što na performanse s korisničke perspektive utjecaj može, ali i ne mora biti vezan uz infrastrukturu web mjesta, bitno je da inženjeri u procesu testiranja:

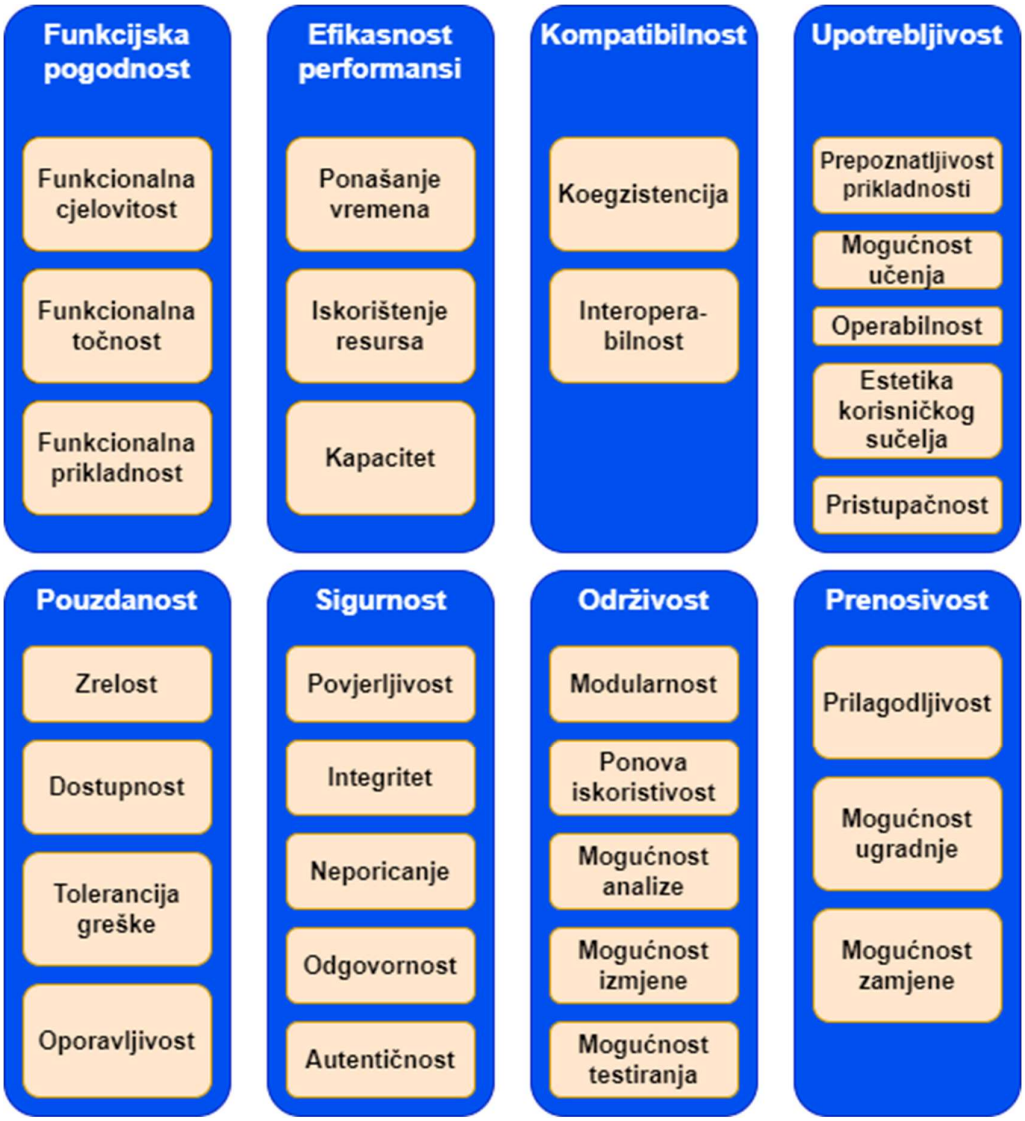
- Detaljno izlože problematiku testiranja klijentima i razvojnom timu, te na taj način istaknu faktore performansi koje testovi ne mogu predvidjeti
- Jasno definiraju vrstu i postupak mjerenja
- Jasno definiraju mjerljive zahtjeve, korisničke priče (*eng. User stories*, izraz za korisničke scenarije korištenja/kretanja web mjestom) te kriterije koji predstavljaju granicu zadovoljenih performansi

(Yorkston, 2021)

2.2. Performanse i model kvalitete

Prethodno smo spomenuli da kvaliteta softvera i korisničko iskustvo ne ovise samo o performansama sustava, te da postoje i drugi čimbenici. Zapravo, performanse su povezane i direktno utječu na ostale čimbenike kvalitete te je stoga iznimno bitno da su inženjeri testiranja performansi svjesni i educirani o tim poveznicama. Npr., loše performanse mogu smanjiti razinu upotrebljivosti, pouzdanosti, sigurnosti, a ponekad i funkcionalnosti sustava.

Kvaliteta softvera nije vezana samo uz web aplikacije nego i za ostale tipove aplikacija poput desktop ili mobilnih, te za razne servise i sisteme. 1991. godine definiran prvi značajni model kvalitete proizvoda (eng. *Product Quality Model*) ISO-9126 koji se sastojao od 6 osnovnih grupa karakteristika te je jedna od njih bila vezana uz performanse i nazvana je efikasnost/učinkovitost. No, kako je taj model bio izrađen u godinama kada nitko nije ni mogao pojmiti u kojem smjeru će se Internet razvijati, potrebno je bilo napraviti korekciju ovog modela kako bi i dalje bio relevantan uzevši u obzir nove tehnologije i mogućnosti. Primjerice jedan značajan problem koji se javio s vremenom kod modela ISO-9126 jest da je sigurnost bila samo funkcionalna karakteristika. U današnje doba vrlo je jasno da sigurnost više nije samo opcionalna funkcionalnost, nego nužna potreba svakog softverskog proizvoda. Stoga, niz godina model je bio ažuriran sve do kada nije definirana skupina standarda SQuaRE (Software product Quality Requirements and Evaluation) te je 2011. godine objavljen novi model nazvan ISO-25010. Ovaj model se sada sastoji od 8 karakteristika i ona vezana uz performanse se sada naziva *Efikasnost performansi*.



Slika 1, Model kvalitete ISO-25010 (Yorkston, 2021)

Testove pripremaju, izvode i interpretiraju rezultate inženjeri performansi koji su članovi tima za osiguranje kvalitete (*eng. Quality assurance*). Svaki inženjer performansi trebao bi biti vrlo dobro upoznat s ovim modelom kako bi bio svjestan kojim udjelom performanse utječu na kvalitetu proizvoda i kako su one povezane s drugim karakteristikama.

Efikasnost u modelu ISO-9126 (a vrlo slično i u ISO-25010) opisana je kao mogućnost softverskog proizvoda da pruži očekivane performanse korištenjem određene količine resursa pod zadanim uvjetima. Očekivane performanse odnose se na one koje utječu na pozitivno korisničko iskustvo (brže učitavanje). Možda najzanimljivija stavka iz definicije jesu korišteni resursi, jer pitanje jest što ako su tijekom testiranja primijećene loše performanse, a pritom je iskorištenost resursa bila gotovo 100 postotna? Prva stvar koju bi smo pomislili jest da kapacitet infrastrukture (jačina procesora, mreža, brzina hard diska…) jednostavno nije dovoljan i to nam

onemogućava postizanje dobrih performansi. Danas, pošto izgradnja moderne web aplikacije se temelji na Cloud servisima, pomoću konfiguracija može se u vrlo kratkom vremenu povećati raspoloživa infrastruktura i na taj način poboljšati performanse. No, što ako sam kod aplikacije nije dovoljno optimiziran i postoje propusti u kodu koji zauzimaju više resursa nego je to potrebno. Zato je vrlo bitno u kontekstu zauzeća resursa sagledati opcije optimizacije koda, konfiguracija, a tek onda povećati kapacitet infrastrukture jer to znači i povećanje troškova. Posljednja stavka ove definicije jesu zadani uvjeti koji se moraju precizno istražiti i definirati, kako bi bili racionalni i realni s obzirom na optimalne performanse i očekivanja.

Karakteristika efikasnosti performansi u modelu ISO-25010 se također, kao i kod starijeg modela, sastoji od 3 podkarakteristike: vremensko ponašanje (eng, *Time behaviour*), korištenje resursa (eng. *Resource utilization*) i kapacitet (eng. *Capacity*). U nastavku ćemo detaljno objasniti svaku od njih. (Yorkston, 2021; Molyneaux, 2009)

2.2.1. Vremensko ponašanje

Vremensko ponašanje najčešće je korištena metrika kod testiranja performansi. Ova metrika ispituje dali je neki sistem ili aplikacija sposobna dati odgovor u određenom vremenu pod zadanim uvjetima. Primjer ovakvog testa mogao bi biti da klijent pošalje zahtjev za učitavanjem početne stranice nekog web mjesta, i očekivano je da dobije odgovor unutar jedne sekunde, bez poruka pogreške i s cjelovitim sadržajem. Vrijeme odgovora može biti zadovoljavajuće za jednog klijenta ili jedan zahtjev, ali kada u jednoj sekundi sistem primi veći broj zahtjeva tada zbog ograničenih resursa vrijeme obrade, a tako i odgovora može se značajno povećati. Upravo je to i svrha mjerenja vremena u testovima performansi, da se ispitaju neki realni rubni i kritični slučajevi.

Kod vremenskog ponašanja od krucijalne je važnosti odrediti realne vremenske kriterije i uvjete. U uvodnom djelu spomenuli smo da je od strane Googleovih analiza preporučeno vrijeme odgovora web mjesta unutar pola sekunde (500ms). Unutar tog intervala korisnik ima osjećaj da je kretanje web mjestom fluidno i zadovoljavajuće. Često se testovi postavljaju da 95 posto odgovora bude u intervalu od pola sekunde jer primjerice zbog mogućih poteškoća u web mreži ponekad odgovor može biti malo dulji. No, ako od 100 zahtjeva 95 puta dobijemo odgovor u zadanim vremenskim intervalima, utjecaj na iskustvo će ostati pozitivan. Također, dvije korisne metrike jesu prosječno i maksimalno vrijeme odgovora. Prosječno vrijeme odgovora daje općenitu sliku performansi neke web stranice, odnosno daje prosječno vrijeme svih odgovora web stranice tijekom testiranja. Maksimalno vrijeme odgovora ukazuje često na rjeđe slučajeve kada vrijeme odgovora može biti veće od očekivanog i poželjnog. Razlog može

biti jedna loše konekcija prema web stranici te je to tada zanemarivo, ali i može ukazati na neke neočekivane propuste koji bi kasnije mogli dovesti do loših performansi. Druga bitna stavka kod mjerenja vremena jest kreiranje scenarija temeljem takozvanih korisničkih priča (eng. *User stories*). Scenariji definiraju kretanje korisnika web mjestom i akcije koje korisnik u stvarnom slučaju izvodi. Korisničke priče kreirane su od strane projektnog tima temeljem njihovih pretpostavki o korisničkim očekivanjima, te se ponekad razlikuju od stvarnih korisničkih scenarija. Stoga, kod pisanja testova treba uzeti u obzir korisničke priče, ali s dozom korekcije prema stvarnim korisničkim scenarijima. (Yorkston, 2021; Molyneaux, 2009)

2.2.2. Korištenje resursa

U uvodnom djelu 2.2. poglavlja dotaknuli smo se korištenja resursa te da se pomoću testova mogu otkriti poteškoće u neoptimalnom kodu i samoj arhitekturi web mjesta. Korištenje resursa kao metrika opisuje se kao udio korištenja raznih tipova resursa nekog sustava kada su performanse zadovoljavajuće. Zapravo, ovdje mjerimo koliko opterećenje (eng. *Load*) naš sustav ili aplikacija može podnijeti. (Yorkston, 2021). Bitno je postaviti realne zahtjeve testova na način da ispituju realne mogućnosti trenutnog stanja sustava, ali da i zahtjevi nisu previsoki s obzirom na infrastrukturu s kojom raspolažemo. Maksimalnim opterećenjem možemo dobiti određenu razinu performansi sustava, ali to ne znači da su one maksimalne, primjerice ako sam kod aplikacije nije optimiziran. Stoga, bitno je odraditi detaljnu analizu aplikacije, imati dovoljno znanja i iskustva o mogućnosti infrastrukture te o dobrim principima pisanja koda.

Postoji više vrsta resursa koji su korišteni od strane aplikacije u produkcijskom okruženju pa tako i na testnom za vrijeme testiranja, a to su slijedeći:

- **Procesor –** Procesorsko vrijeme vjerojatno je resurs na kojeg bi svatko prvo pomislio u kontekstu performansi i odnosi se na izvršavanje raznih instrukcija u nekom određenom vremenu. Neki standard o iskorištenju procesora jest da maksimalno zauzeće bi trebalo biti do nekih 80 posto, a ako su performanse i dalje nedovoljne potrebno je skalirati. Ponekad da bi se maksimalno iskoristio procesor, potrebno je optimizirati kod i koristiti višedretvenost kako bi se koristile sve dostupne jezgre procesora.
- **Memorija -** Memorija se odnosi na procesorsku *cache* memoriju i radnu memoriju (RAM). *Cache* i RAM su privremena vrsta memorije koje direktno utječu na izvođenje instrukcija procesora svojom veličinom i brzinom.

- **Diskovi za pohranu -** Diskovi za pohranu čine statičku memoriju u koju se pohranjuju podaci te je ovdje ključna brzina čitanja i pisanja koja također može imati utjecaj na brzinu izvođenja instrukcija.
- **Propusnost mreže –** Propusnost (eng. *bandwidth*) mreže određena je odabirom tipa servera ili *Cloud* servisa i jedna je od važnijih resursa. Ako propusnost nije dovoljno velika da prosljeđuje sadržaj na zahtjev većeg broja korisnika tada ni brz procesor i memorija neće doći do izražaja.

(Yorkston, 2021; Molyneaux, 2009)

Korištenje resursa vrlo je bitan čimbenik kod testiranja performansi jer nam govori koliko je trenutno iskorištenje resursa s obzirom na vrstu i količinu opterećenja nad sustavom/aplikacijom. Također, vrlo je koristan čimbenik za detekciju propusta u kodu aplikacije jer kada do propusta dođe, iskorištenje resursa biti će nerealno veće.

2.2.3. Kapacitet

Kapacitet kao karakteristika daje informaciju koji su maksimalni limiti sustava unutar kojih su još uvijek ispunjene zadovoljavajuće performanse sustava. Maksimalni kapacitet vrlo je važno izmjeriti kako bismo bili svjesni mogućnosti naše arhitekture, naravno uz optimiziran kod. Testiranje kapaciteta se postiže povećanjem broja virtualnih korisnika do granice zadovoljavajućih performansi. Da bismo definirali takav test potrebno je prethodno odrediti dvije vrste testnih profila:

- **Operacijski profil** – stvarni ili predviđeni uzorak korištenja komponente ili sistema
- **Profil opterećenja –** određuje broj virtualnih korisnika koje izvršavaju skup transakcija u nekom određenom vremenskom periodu na određenom komponentom ili sistemom.

Zapravo, operacijski profil određuje stvarni scenarij koji korisnik može izvršiti na produkciji, dok profil opterećenja određen je od strane inženjera testa te određuje koji scenariji se izvršavaju, s kojim brojem virtualnih korisnika i na koji način. Testovi se mogu konfigurirati na dva načina:

- Prvo definirati kapacitet koji želimo postići te ponavljajućim testiranjem odrediti odgovarajući operacijski profil virtualnog korisnika kako bi se postigao kapacitet
- Prvo definirati operacijski profil temeljem nekog stvarnog scenarija te tada podesiti profil opterećenja tako da se dobije maksimalni kapacitet opterećenja

(Yorkston, 2021; Molyneaux, 2009)

U ovoj sekciji spomenuli smo u više navrata opterećenje (eng. *Load*) sustava tijekom testiranja. Opterećenje u ovom kontekstu jest količina izvršenja procesa u sustavu koja je izazvana akcijama koje generira korisnik. No, ovdje je bitno nadodati još jednu stavku koja je poveznica između korisnika i izvršenja procesa, a to je programski kod sustava. Zapravo na svaku korisničku akciju poziva se izvršenje koda koje tada pokreće procese koji opet nadalje koriste resurse infrastrukture. (Yorkston, 2021)

Kapacitet zapravo povezuje prethodne dvije spomenute karakteristike na način da razmatra mogućnost sustava da podržava definirano opterećenje tako da generira stvarno opterećenje temeljem operacijskog i profila opterećenja, a pritom mjeri vremensko ponašanje i korištenje resursa. Zbog toga testiranje performansi često jest eksperiment u kojem se mjere i analiziraju parametri testa. Na taj način iterativnim postupkom mogu se donijeti odluke kako oblikovati arhitekturu sustava da bi se zadovoljila očekivanja interesenata.

2.3. Principi testiranja

Keith Yorkston (Yorkston, 2021) u svojoj je knjizi definirao pet osnovnih principa koje bi inženjeri trebali slijediti da bi testovi bili relevantni i korisni. Ovi principi vrijede za sve vrste testiranja, bilo funkcionalnih ili nefunkcionalnih. U nastavku slijede principi i njihovi opisi.

Testovi moraju biti u skladu sa definiranim očekivanjima različitih interesnih skupina kao što su korisnici, dizajneri sustava i operativno osoblje. Svaki veći projekt sastoji se od više skupina različitih interesnih grupa. Primjerice, to može biti tehnička grupa u koju spadaju administratori i developeri te ostale projektne role. Sa druge strane je to ne-tehnička grupa u koju spadaju poslovni korisnici i menadžment od strane klijenata. Naravno, svaka od ovih grupa i pa čak i podgrupa ima različite zahtjeve, te je prema tome potrebno definirati različite relevantne metrike. Inženjeri testiranja trebaju uzeti u obzir generalne zahtjeve i ciljeve projekta, one koji se tiču specifičnih korisničkih priča od različitih interesnih grupa te odrediti relevantne testove kako bi dokazali da su svi zahtjevi ispunjeni.

Testovi moraju biti definirani tako da se mogu ponavljati, te svaki puta uz nepromijenjenu infrastrukturu i arhitekturu koda moraju dati iste rezultate. Dakle, da bi testovi performansi kod svakog izvođenja dali iste rezultate bitno je da okruženje u kojem se izvode bude uvijek konzistentno. Produkcijsko okruženje vrlo je promjenjivo, pogotovo uz moderne Cloud servise koji nude vrlo brzo i efikasno skaliranje. To znači kada se primijeti značajno povećanje opterećenja (broja zahtjeva korisnika) raspoloživi resursu skalirat će se

prema gore. Zbog toga testiranje performansi zahtjeva i testna okruženja gdje je infrastruktura stabilna, bez vanjskih utjecaja i bez promjena konfiguracija. Na taj način kod svake nove verzije sustava, ako dođe do značajnih promjena u performansama, one su lako uočljive i prouzročene od strane nove verzije koda. No, to nije uvijek željeni oblik testnog okruženja jer ponekad zahtjevi testiranja su takvi da testna okruženja u nekoj mjeri moraju imitirati produkciju. To se može postići na način da generiranje opterećenja kod svakog pokretanja nije jednako ili da sistem ima mogućnost skaliranja kako bi se testirale dodatne mogućnosti same infrastrukture u kombinaciji sa kodom aplikacije. Dakle, inženjeri trebaju razlučiti koja vrsta testova najbolje odgovara u određenoj situaciji ili jesu li možda potrebne obje vrste testova.

Testovi moraju dati rezultate koji su razumljivi i mogu se usporediti sa zahtjevima interesnih grupa. Testovi moraju težiti tome da ostvare svoju svrhu, to jest da temeljem metrika daju sliku o tome da li trenutno stanje ispunjava zahtjeve interesnih grupa. Često rezultati nisu isprva razumljivi interesnim grupama pa je tako zadatak inženjera interpretacija rezultata, bilo opisom ili nekom vrstom grafa.

Testovi se mogu izvoditi gdje to resursi omogućuju bilo na gotovim ili polovičnim sustavima, testnim okruženjima koja imaju slične karakteristike produkcijskog okruženja. U prošlosti testiranje performansi izvodilo se na okruženjima doslovno jednakim produkciji. Takvo što je danas nemoguće jer produkcijska okruženja su često pogonjena, kao što smo već spomenuli, Cloud servisima koji su vrlo moćni, ali isto tako generiraju visoke troškove. To je uredu kada u obzir uzmemo produkciju, no troškovi za testna okruženja žele se uvijek minimalizirati. Sa druge strane kada testna okruženja nemaju osobine produkcije, vrlo je teško postaviti testove da budu relevantni. No, ipak iskusni inženjeri trebali bi prepoznati osobine produkcije i translatirati ih na testna okruženja u manjoj skali.

Npr. vrlo popularna tehnologija koja se u današnje vrijeme koristi u Cloudu jest Kubernetes. Ova tehnologija koristi kontejnere od kojih svaki predstavlja jednu instancu aplikacije. Kada je opterećenje povećano kreirat će se novi kontejneri. Na taj način obično funkcionira produkcija. Testno okruženje bi tada trebalo imati fiksni broj kontejnera, možda čak i samo jedan, te ako za taj jedan kontejner aplikacija radi optimalno, s obzirom na neko opterećenje, tada će i svaki kontejner optimalno raditi na produkciji, a naposlijetku i cijela grupa kontejnera. Na taj način može se zapravo izmjeriti koliko opterećenja podnosi svaki kontejner, predvidjeti skaliranje i troškovi te je li potrebna optimizacija.

Često nije jednostavno na testnim okruženjima izmjeriti osobine produkcije. Stoga, sljedeća ideja je izvoditi testiranje na manjim komponentama sustava kako bi se neželjene promjene lakše uočile.

Testovi moraju biti dostupni i izvršivi unutar nekog vremenskog intervala određenog od strane projekta. Testiranje performansi je neizbježno kod CI/CD (eng. *Constant integration and continuous delivery*) pristupa razvoju jer da bi bilo moguće isporučiti nove verzije u kratkim vremenskim periodima, one moraju biti sigurne od pogrešaka, tj. testirane. Bitno je da se testovi izvršavaju na što manjim promjenama, dakle što češće, ali to ponekad može prouzročiti visoke troškove. Testiranje također ulazi u konačni račun uloženog/dobivenog te je vrlo važno pomno odrediti koliki troškovi testiranja opravdavaju dobivene beneficije. Često se na velikim projektima dešava situacija da menadžment odluči da su troškovi testiranja preveliki, a to se onda odrazi na kvalitetu sustava i prouzroči još veće troškove ili značajno manje prihode zbog raznih rizika.

2.4. Tipovi testiranja performansi

Do sada smo spomenuli da testiranje performansi temeljem raznih metrika generira rezultate na ograničenoj infrastrukturi. No, testovi mogu biti različitih tipova i odabiru se s obzirom na potrebe projekta. Oblik i tip testa određuje se temeljem ciljeva, svrhe, korisničkih priča i zahtjeva. Na taj način definira se okvir za testiranje poslovnih procesa i njihovih operacijskih profila te na kraju dobivamo profil opterećenja (eng. *Load profile*). Testiranje performansi je generalni okvir koji se dijeli na više vrsta testova, koje ćemo detaljno obraditi u nastavku. (Yorkston, 2021)

2.4.1. Testiranje opterećenja

Testiranje opterećenja (eng. *Load testing*) daje nam rezultate kako se ponaša neki sustav kod generiranih različitih razina opterećenja. Kod ovog tipa testiranja pokušavaju se imitirati stvarni produkcijski uvjeti. Primjerice, kod testiranja web aplikacije cilj može biti generiranje 100 različitih tipova zahtjeva u sekundi u trajanju od 15 minuta. Na taj način postavljanjem relevantnih metrika i analizom rezultata vidimo je li web aplikacija spremna podnijeti određeno opterećenje, u našem slučaju 100 zahtjeva u sekundi. Često se takvi testovi izvode desetak minuta jer to nam daje do znanja je li aplikacija sposobna održati stabilno stanje pod navedenim opterećenjem više od nekoliko sekundi, a ako je spremna desetak minuta tada je spremna i višesatno. Testiranje opterećenja sastavni je dio testiranja performansi , a i možda najbitniji jer od njega proizlaze ostale vrste testiranja. Operacijski i profil opterećenja osnova su ove vrste testiranja (znani kao volumetrije, eng. *volumetrics*) i slijede slijedeća pitanja:

- **Tko?** Tko pripada grupi korisnika koja koristi trenutni sustav? Svaka grupa može izvoditi različite vrste akcija i posjedovati različita ovlaštenja.
- **Što?** Na ovo pitanje odgovor se dobiva identificiranjem nekog poslovnog procesa kojeg korisnik obavlja. Svaki poslovni proces može se podijeliti na zadatke, dok se oni opet mogu podijeliti na korake ili akcije. Na taj način u testiranju definiraju se scenariji sa detaljnim koracima koji se kao takvi mogu ponavljati. Ponavljanjem velikog broja scenarija u isto vrijeme generiramo opterećenje.
- **Gdje?** Kod ovog pitanja postoje dva aspekata. Prvi je da korisnici mogu koristiti sustav s jedne centralizirane lokacije ili s više različitih lokacija. Drugi aspekt govori o geolokaciji, odnosno o tome da opterećenje može putovati kroz više servera, servisa, komponenta i može biti generirano od različitih poslovnih procesa.
- **Kada?** U različita vremenska razdoblja opterećenje varira, stoga potrebno je odrediti testove različitih opterećenja kako bi se pokrili svi rubni slučajevi. Primjerice web trgovine imaju veće opterećenje tijekom vikenda nego u sredini tjedna.
- **Kako?** Bitno je uzeti u obzir na koji način korisnici izvode poslovne korake. Primjerice, novi korisnici sporije će se kretati nekim web mjestom i koristit će manje funkcionalnosti.

(Yorkston, 2021; Shivakumar, 2020)

2.4.2. Testiranje otpornosti na stres

Testiranje otpornosti na stres (eng. *Stress testing*) fokusira se na krajnje opterećenje koje sustav može podnijeti. Ova vrsta testa proizlazi iz testiranja opterećenja i ispituje razinu radnog opterećenja koja je iznad zadanog cilja ili predstavlja čak i limite sustava. Cilj testiranja je razina opterećenja izmjerena analitikom na produkcijskim sustavima, no to nije nikad limit sustava. Ovo testiranje često se smatra kao jedini cilj testiranja performansi jer ispituje maksimalne mogućnosti. Zapravo, identificira maksimalni kapacitet sustava i koji dio sustava će najprije pasti kod granice kapaciteta. (Yorkston, 2021; Shivakumar, 2020). Rezultati otpornosti na stres vrlo su korisni jer daju informaciju o maksimalnom podržanom opterećenju i planiranju promjena koje mogu biti optimizacija koda ili unaprjeđenje infrastrukture.

2.4.3. Testiranje skalabilnosti

Svrha testiranja skalabilnosti (eng. *Scalability testing*) jest mjerenje sposobnosti da sustav zadovolji zahtjeve performansi veće od trenutnih. Cilj je izmjeriti sposobnost sustava da

raste, tj. podnosi veći broj korisnika i obrađuje veće količine podataka, a da pritom zahtjevi performansi i dalje budu ispunjeni. Jednom kada su mogućnosti skaliranja izmjerene moguće je planirati promjenu konfiguracija skaliranja temeljem analitika sa produkcije kako bi se izbjegle loše performanse. (Yorkston, 2021)

U pravilu svaki sustav je skalabilan, samo je pitanje na koji način se skaliranje izvodi i koliko je efikasno. Skaliranje može biti vertikalno i horizontalno. Kod horizontalnog skaliranja povećava se broj instanci ili komponenata sustava dok, kod vertikalnog se unaprjeđuju snaga i kapacitet jedne instance ili komponente. U pogledu web razvoja, kod manjih projekata, web mjesta se pokreću na klasičnim virtualnim mašinama i serverima. Takvi sustavi vrlo se teško skaliraju i često su potrebne manualne akcije. Horizontalnim skaliranjem dodaju se novi serveri i virtualne mašine, dok vertikalnim skaliranjem slabija virtualna mašina bi se zamijenila jačom. Logično je za pretpostaviti da kod takvog vertikalnog skaliranja naš sustav će na neko vrijeme biti nedostupan, što i nije baš poželjno. Takve probleme danas rješavaju brojni Cloud servisi koji omogućuju automatska skaliranja. No, virtualne mašine i dalje su spore kod paljenja i gašenja, a upravo te probleme rješava već spomenuta kontejnerizacija i Kubernetes u kombinaciji sa Cloud servisima jer nudi vrlo fleksibilna automatska skaliranja, gdje se jedan kontejner (koji predstavlja jednu virtualnu mašinu) pali i gasi u nekoliko sekundi. (Kubernetes vs. Virtual Machines, Explained, n.d.)

2.4.4. Testiranje naleta opterećenja

Testiranje naleta opterećenja (eng. *Spike testing*) daje nam informacije koliko dobro sustav reagira na iznenadne nalete graničnog opterećenja i koliko se efikasno vraća u stabilno stanje. Ova vrsta testiranja korisna je jer pokazuje kako će se sustav ponašati kada opterećenje preskoči granicu predviđenog opterećenja na kratko vrijeme. Ovakav događaj može biti jednokratan, serija slučajeva jednakih vremenskih intervala ili serija nasumičnih slučajeva. (Yorkston, 2021; Shivakumar, 2020). Takva mjerenja i rezultati omogućuju testiranje rubnih slučajeva koji se mogu predvidjeti na temelju iskustava te ih je tada moguće izbjeći ili postaviti adekvatne konfiguracije za oporavak.

2.4.5. Testiranje izdržljivosti

Cilj testiranja izdržljivosti (eng. *Endurance testing*) jest dobiti rezultate stabilnosti sustava pod nekim određenim opterećenjem na duži vremenski interval. Rezultati daju informacije o potencijalnim problemima kapaciteta infrastrukture koji bi mogli dovesti do neželjenog pada performansi ili čak do rušenja sustava. (Yorkston, 2021). Ovim tipom testiranja

mogu se uočiti greške u kodu koje bi nakon nekog vremena mogle prouzročiti neželjeno popunjavanje kapaciteta sustava.

Razlika između klasičnog testiranja opterećenja i testiranja izdržljivosti jest u vremenu izvođenja, iako za oboje se definiraju isti ili slični profili opterećenja. Testiranje izdržljivosti često se izvodi nekoliko sati ili čak i dana kako bi bilo dovoljno vremena da se eventualni problemi s korištenjem resursa mogu detektirati. Oni kritičniji detektiraju se vrlo brzo jer kapaciteti resursa se tada brže popunjavaju i dolazi do pada performansi ili rušenja sustava. (Yorkston, 2021; Shivakumar, 2020). Duljim testiranjem moguće je detektirati i veći broj potencijalnih manjih nedostataka. Kod Cloud tehnologija manje je vjerojatno da će se sustav srušiti, ali zbog efektivnog skaliranja može prouzročiti znatno veće troškove.

2.4.6. Testiranje konkurentnosti

Kada je cilj testiranja izmjeriti performanse akcija ili događaja koji se izvode istovremeno koristi se testiranje konkurentnosti (eng. *Concurrency testing*). (Yorkston, 2021)

Klasičnim testiranjem opterećenja kreira se opterećenje generirano od većeg broja virtualnih korisnika koji izvode neki definirani scenarij ili skup zadataka i akcija. No, u ovom slučaju akcije istog tipa ne izvode se istovremeno, bar ne na dovoljnoj skali kao što to može biti pojava u produkcijskim okruženjima. Zato, bitno je testirati slučajeve kada se slične akcije, zadaci ili transakcije izvode u isto vrijeme jer na taj način testiraju se performanse pojedinačnih dijelova sustava.

Prema Keith Yorkstonu (2021.) postoje tri vrste testiranja konkurentnosti:

- **Aplikacijska konkurentnost –** kada istovremeno određen broj korisnika izvodi neki skup poslovnih procesa koju predstavljaju jednu funkcionalnost kao što su pretraživanje, kupnja, kreiranje računa…
- **Konkurentnost poslovnih procesa –** kada manji broj korisnika istovremeno izvodi jedan poslovni proces, npr. pretraživanje
- **Transakcijska konkurentnost –** određen broj korisnika istovremeno izvršava jedan zadatak/korak u poslovnom procesu kao što je klik na gumb pretraživanja koji tada u pozadini pokreće određen proces

Testiranje konkurentnosti daje informacije o performansama pojedinih dijelova sustava, detektira eventualne probleme i na taj način omogućava razvojnom timu da optimizira ili redizajnira kritične komponente.

2.4.7. Testiranje kapaciteta

Da bi se dobila precizna informacija koliko maksimalno korisnika i opterećenja koje oni generiraju podržava trenutni sustav, a da pritom performanse i dalje budu unutar zadovoljenih okvira provodi se testiranje kapaciteta (eng. *Capacity testing*). Iako veći broj korisnika će naravno prouzročiti veće opterećenje, bitno je uzeti u obzir i volumen transakcija, odnosno koliko opterećenje određena transakcija generira jer to također ima utjecaj na ukupno generirano opterećenje. (Yorkston, 2021)

Testiranje kapaciteta vrlo je slično testiranju otpornosti na stres i testiranju naleta opterećenja, ali ipak postoje razlike. Primjerice, testiranje otpornosti na stres generira opterećenje koje premašuje maksimalne kapacitete resursa kako bi uzrokovalo pad i ispitalo oporavak sustava. Na drugoj strani, testiranje kapaciteta premašuje trenutno stvarno opterećenje, ali ne prelazi kapacitet resursa jer je upravo to i cilj, ispitati kapacitet (maksimalan broj korisnika). Isto vrijedi i za testiranje naletu opterećenja gdje se u kratkim naletima generira opterećenje izvan kapaciteta sustava kako bi se promatralo ponašanje i pronašle slabosti. Rezultati testiranja kapaciteta od koristi su kod donošenja odluka o povećanju infrastrukture temeljem predviđanja rasta broja korisnika od strane organizacije/vlasnika. (Yorkston, 2021)

2.5. Aktivnosti u testiranju performansi

U prethodnoj sekciji obradili smo glavne vrste testiranja performansi, no testiranje se također može podijeliti u dva glavna tipa temeljem izvođenja aktivnosti: **statičko** i **dinamičko** testiranje. Također prethodno smo svaku vrstu opisali iz aspekta dinamičkog testiranja, tj. kako se testovi izvršavaju nad nekim testnim objektima uz predefinirane zahtjeve. No, postoji i statičko testiranje koje predstavlja proces procjene sustava temeljem njegove forme, strukture, sadržaja ili dokumentacije, bez izvršavanja testova. Ova dva tipa obradit ćemo detaljnije u nastavku ove sekcije i objasniti njihovu povezanost.

2.5.1. Statičko testiranje

Aktivnosti statičkog testiranja jednako su važne kao i dinamičko testiranje u pogledu testiranja performansi. Mnogo kritičnih defekata performansi često se pojavi već u samom početku kod arhitekture i dizajna sustava. Takvi propusti mogu se desiti zbog nerazumijevanja ili nedostatka znanja kod projektnih dizajnera i arhitekata. Isto tako, propusti se mogu pojaviti ako zahtjevi nemaju adekvatno definirano vrijeme odgovora, propusnost, iskorištenje resursa,

očekivano opterećenje i iskorištenje sustava. Upravo se takvi nedostaci rješavaju statičkim testiranjem koje može uključiti:

- Pregled zahtjeva sa fokusom na aspekt performansi i rizika
- Proučavanje shema baze podataka, dijagrama entiteta, metapodataka, procedura i upita
- Pregled sustava i mrežne arhitekture
- Pregled kritičnih dijelova u kodu sustava

(Yorkston, 2021)

Tradicionalno, statičko testiranje performansi nije se izdvajalo kao neka posebna cjelina i nije bilo posebno istaknuto. Razlog je bio to što se testiranje izvodilo u krajnjim fazama razvoja softvera i glavni naglasak bio je na dinamičke testove. (Yorkston, 2021)

No, softverski sustavi kroz vrijeme postali su sve kompleksniji, metodike razvoja postale su dinamičnije pa je i važnost na testiranju performansi postala sve veća. Upravo zato došlo je do spoznaje da je vrlo bitno potencijalne defekte uočiti u samim početcima razvoja i definirati realne zahtjeve koji će se koristiti kod dinamičkog testiranja u svakom novom razvojnom ciklusu.

Također, statičko testiranje otkriva defekte u memoriji, radu sa višedretvenim tehnologijama ili u nekim jednostavnim greškama u kodu kao beskrajne petlje, ne zatvaranje bespotrebnih objekata (konekcija na bazu podataka) i slično. (Yorkston, 2021). Danas su neki od ovih nedostataka pokriveni raznim dodacima u alatima za razvoj i pisanje koda (npr. *Sonarcube* za detekciju kritičnih grešaka u kodu koje bi mogle utjecati na performanse).

2.5.2. Dinamičko testiranje

Dinamičko testiranje sastoji se od izvršavanja različitih vrsta testova. Testiranje performansi također može se odvijati na više razina, a u prethodnoj sekciji spomenuli smo glavne vrste testiranja performansi koje su se odnosile na samo nekoliko viših razina koje se najčešće koriste. U nastavku ćemo spomenuti osnovne razine i njihove karakteristike.

Jedinično testiranje

Ovo je najniža razina testiranja koja počinje od testiranja najmanjih jedinica ili komponenti sustava. Komponenta predstavlja najmanju logičku cjelinu u sustavu i kod softvera to jeste neka logička komponenta koda (npr. klasa, servis…). Komponente se testiraju u izolaciji što znači da se za njih treba pripremiti testna okolina. Okolina se može sastojati od raznih lažnih/imitirajućih objekata, podataka, servisa ili drivera (eng. *Mock data*) koji oponašaju stvarne. Svrha je provjeriti performanse pojedinačne komponente zanemarujući

utjecaj okoline. Ovi testovi kreiraju se od strane developera u ranoj fazi razvoja neke funkcionalnosti, tj. u fazi pisanja koda.

Većinom se ovi testovi koriste kako bi se provjerila ispravnost određenih komponenti, ali se mogu mjeriti i performanse kao što je vrijeme izvođenja logike u kodu, spajanje na bazu i slično. Uglavnom obavlja se provjera iskorištenja resursa na vrlo detaljnoj razini. (Yorkston, 2021)

Integracijsko testiranje

Sljedeća razina testiranja jest integracijsko u kojem se testira međusobno djelovanje više komponenti ili jedne veće funkcionalne cjeline. Kod ove razine često je potrebno konfigurirati manje imitirajućih objekata, ali okolina i dalje mora biti definirana jer ovaj funkcionalni skup komponenta je i dalje zavisan o drugim dijelovima cjelokupnog sustava.

Ova razina testova također se kreira od strane developera, te se provjeravaju performanse različitih slučajeva korištenja i poslovnih tijeka. (Yorkston, 2021)

Testiranje sustava

Testiranje sustava odvija se na razni gdje inženjeri nemaju doticaja sa izvornim kodom. U agilnim metodikama razvoja zasebni sustavi testiraju se u svakom ciklusu. Sustavi su često povezani ili ovisni o više drugih sustava pa zahtijevaju izolirana okruženja kako bi se dobile individualne performanse. Kod ovog testiranja koriste se prethodno obrađene vrste testiranja generiranjem određene vrste i količine opterećenja da bi se ispitali rezultati temeljem zahtjeva.

Integracijsko testiranje sustava

Kao što integracijsko testiranje povezuje skup komponenata, na sličan način to radi i integracijsko testiranje sustava koje testira grupu međusobno povezanih ili ovisnih sustava. Ovakvo testiranje učestalo je u novije vrijeme gdje se razna softverska web rješenja sastoje niza različitih Cloud mikroservisa. Svaki mikroservis predstavlja jedan zaseban sustav te se integracijskim testiranjem mjere performanse njihove komunikacije i međudjelovanja. Najveći izazov jest konfigurirati testna okruženja (jer testiranje se mora odraditi prije produkcije) jer u cjelokupno rješenje možda je potrebno uključiti veći broj sustava od kojih neki mogu biti i vanjski. Na ovoj razini se izvode testiranja performansi generiranjem raznih vrsta opterećenja.

Testiranje prihvatljivosti

Posljednja razina jest prihvatljivost gdje je glavni cilj odrediti da li su određene funkcionalnosti sukladne zahtjevima raznih interesnih skupina. Prihvatljivost se može podijeliti na nekoliko podrazina:

- Testiranje korisničke prihvatljivosti – da bi se odredila prihvatljivost ciljanih korisnika sustava

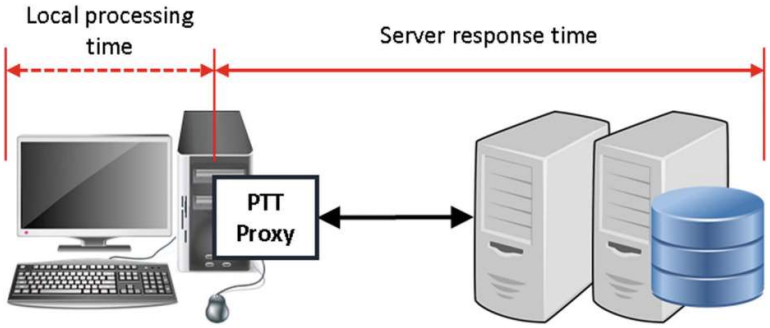
- Testiranje operativne prihvatljivosti – da bi se odredila prihvatljivost od strane operativnih i sistemskih administratora
- Testiranje dogovorene prihvatljivosti – da bi se odredila prihvatljivost sukladno dogovorenim zahtjevima projekta
- Testiranje regulativne prihvatljivosti – da bi se utvrdila prihvatljivost sukladno zakonima, pravilima i regulacijama

Inženjeri testova performansi dotiču se svih spomenutih podrazina testova prihvatljivosti, ali najviše pažnje obraćaju na testove operativne prihvatljivosti. Operativna prihvatljivost često je posljednja provjera izvedena prema zahtjevima administratora sustava koji najpreciznije mogu prepoznati kritične segmente sustava.

2.6. Razlike u testiranju performansi na strani klijenta i servera

Mjerenje rezultata performansi može se odvijati na dvije strane: serverskoj i klijentskoj. U sekciji 2.4. opisani su glavni tipovi testiranja i oni se direktno odnose na testiranje sa strane servera. U ovom poglavlju usporedit ćemo obje strane i objasniti glavne karakteristike na primjeru web mjesta gdje klijentska strana ima veliku ulogu kod kvalitete performansi.

Serverska strana kod generiranja opterećenja direktno koristi API (eng. *Application Programming Interface*) ili URL web mjesta da bi poslala zahtjeve te mjeri vrijeme od slanja zahtjeva pa sve do primitka odgovora. Ono što serverska strana ne pokriva jest lokalno procesiranje i izvođenje dobivenih skripti da bi se renderirala grafika korisničkog sučelja te ovaj dio nazivamo klijentska strana. S pogleda korisnika klijentska strana je jednako bitna kao i serverska jer ono što će korisnik vidjeti na ekranu jest zbroj vremena obje strane.



Slika 2, Vrijeme odgovora na serverskoj i klijentskoj strani (Yorkston, 2021).

2.6.1. Koncept generiranja opterećenja

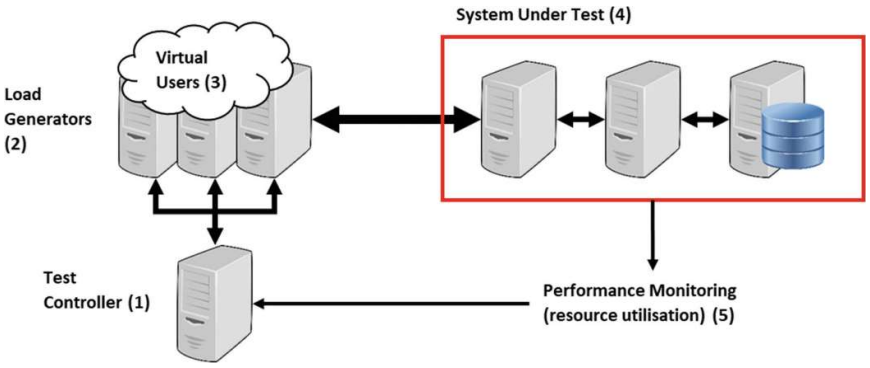
Profil opterećenja kod testiranja performansi vrlo je sličan ulaznim podacima koji se koriste kod funkcionalnog testiranja, ali se razlikuje u sljedećim točkama:

- Opterećenje testiranja performansi mora predstavljati veću količinu korisničkih ulaza/scenarija, a ne samo jedan
- Testiranje performansi u većini slučajeva zahtjeva posebno namijenjeni hardver i alate kako bi se velika količina opterećenja generirala
- Generiranje opterećenja je ovisno o funkcionalnim defektima sustava koji mogu utjecati na izvođenje i rezultat testa

Generatori opterećenja su mašine/infrastruktura na kojima se inicijalizira i upravlja generiranjem opterećenja nad nekim određenim sustavom koji predstavlja objekt testiranja. Na taj način testovi performansi se mogu skalirati i pomoću virtualizacije postići veću skalu opterećenja od samo jedne mašine/korisnika (kao što se to koristi kod funkcionalnog testiranja). Slika ispod pokazuje osnovni koncept i komponente koje predstavljaju cjelinu sustava testiranja performansi:

1) Kontroler izvodi testove performansi i prikuplja rezultate direktno iz izvršenja testova i promatranjem sustava. Zapravo, kontroler prosljeđuje skripte za testiranje performansi generatoru opterećenja koji te skripte izvodi.
2) Generator opterećenja izvodi skripte i šalje zahtjeve sustavu koji se testira te dohvaća rezultate odgovora.
3) Svaka skripta predstavlja jednog virtualnog korisnika koji temeljem instrukcija izvodi akcije i na taj način imitira stvarni poslovni proces nad sustavom
4) Sustav koji se testira reagira na opterećenje, odrađuje potrebne procese i generira odgovore koji predstavljaju neku metriku. Metrike se mogu detektirati na više mjesta, primjerice odgovor na zahtjeve biti će poslan do generatora opterećenja dok će se informacija o iskorištenju resursa poslati direktno kontroleru.
5) Vrijeme odgovora i iskorištenje resursa ciljani su rezultat testova performansi te se temeljem njih mogu detektirati defekti i problemi nastali u sustavu tijekom novog izdanja.

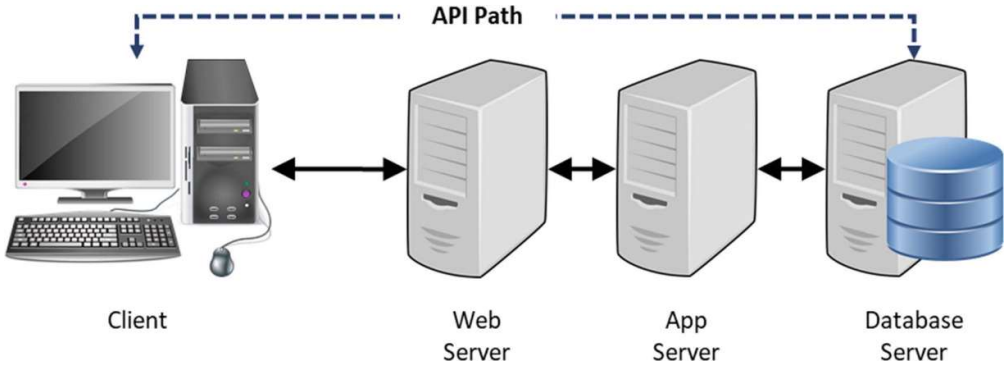
(Yorkston, 2021)



Slika 3, Koncept generiranja opterećenja (Yorkston, 2021)

2.6.2. Testiranje na strani servera

Testiranje performansi na strani servera odnosi se na generiranje opterećenja nad infrastrukturom sustava te se pritom mjeri iskorištenje resursa i vrijeme odgovora. Kod testiranja servera zahtjevi se šalju direktno putem API-ja umjesto simuliranja scenarija putem UI-a jer se na taj način izbjegava učitavanje i izvršavanje klijentskih skripti. Dakle, fokus je na serveru te se pomoću API poziva mogu kreirati scenariji za virtualne korisnike te generirati veliki broj zahtjeva. Ovo testiranje od krucijalne je važnosti u agilnim metodikama razvoja jer jednom definirani testovi daju korisne informacije o padu performansi i defektima sustava. Moderni sustavi često su građeni u više slojeva i temeljem mikroservisa. Tada na primjeru web aplikacije (slika ispod) pomoću API poziva može se generirati opterećenje na neki od slojeva kao što su web server, aplikacijski server ili baza podataka te na taj način testirati zasebne slojeve. Slična stvar je i sa mikroservisima, od kojih svaki ima konfiguriranu serversku instancu te se zahtjevi šalju također putem API.
(Yorkston, 2021; Shivakumar, 2020)



Slika 4, Višeslojna arhitektura web aplikacija (Yorkston, 2021)

Performanse na serverskoj strani definiraju se kao sposobnost usluživanja velikog broja korisnika u isto vrijeme bez velikih promjena u odnosu na prihvatljivo ponašanje. Kod

definiranja testova potrebno je definirati gornji broj korisnika te testiranjem provjeriti jesu li performanse kod tog opterećenja prihvatljive. (Mohan, 2022)

No, dobro je pitanje što je prihvatljivo ponašanje? Prema istraživanjima kada je odgovor neke web aplikacije manji od 0.1 sekunde, korisnik ima osjećaj da je odgovor trenutan. Unutar 1 sekunde korisnik osjeća malu usporenost ali i dalje ima osjećaj da je kretanje aplikacijom fluidno. Veće vrijeme odgovora od 1 sekunde daje osjećaj da je web aplikacija troma i može rezultirati lošim korisničkim iskustvo. Prema Googleovim istraživanjima ako je vrijeme odgovora veće od 3 sekunde tada postoji rizik od gubitka većine korisnika. Stoga, preporuka je da vrijeme odgovora ne smije biti veće od 2 sekunde (Molyneaux, 2009; Data Dome, n.d.)

Prethodno spomenuto vrijeme odgovora uključuje serversko i klijentsko vrijeme zajedno. Stoga, ako je poželjno da vrijeme odgovora koje primijeti korisnik jest unutar 2 sekunde, tada serversko vrijeme mora biti manje kako bi uz dodatno učitavanje na klijentskoj strani i dalje ostali unutar navedene granice. Obično je poželjno da je velika većina (preko 90%) zahtjeva unutar pola sekunde. Više detalja spomenuto je u praktičnom dijelu serverskog testiranja.

Faktori koji mogu utjecati na performanse na strani servera web aplikacije:

- **Dizajn arhitekture sustava** – Ako odgovornosti i ovisnosti između web servisa (mikroservisa) nisu jasno definirane to može dovesti do velikog broja API poziva i većeg vremena odgovora. Također ovdje ključnu ulogu igra i *caching* sustav koji ako nije adekvatno implementiran može izazvati vrlo loše performanse.
- **Izbor tehnologije –** Za različite potrebe ponekad su optimalne različite tehnologije. Kod web aplikacija/sustava prvo je potrebno odabrati glavni razvojni okvir, a nadalje i za ostale pomoćne aplikacije/servise. Ponekad će servisi napisani u node.js dati bolje performanse nego napisani u Javi, ali će imati možda manje mogućnosti.
- **Kompleksnost koda –** Loše napisan *backend* kod može prouzročiti ozbiljne degradacije u performansama, npr. kompleksni kod, dugačke operacije, validacije koje nedostaju ili su nepotrebno komplicirane…
- **Izbor baze podataka –** Postoji više vrsta baza podataka od kojih svaka može zadovoljiti neke zahtjeve u boljoj mjeri, ali i dohvaćanje podataka vrši u različitim brzinama. Stoga, za slučajeve kada je brzina dohvaćanja podataka krucijalna, potrebno je odabrati adekvatan tip baze, ali i organizirati podatke prema dobrim principima
- **Internet mreža –** Server mora biti pokrenut unutar adekvatnih konfiguracija propusnosti mreže kako bi mogao opsluživati veći broj korisnika. S druge strane, u obzir

se mora uzeti konekcija korisnika koja može biti nešto lošija na mobilnim uređajima pa je potrebno odraditi optimizacije da se primjerice prikazuju slike smanjenih rezolucija.

- **Geolokacija –** Popularne web aplikacije privlače korisnike diljem svijeta. Stoga, bitno je optimalno pozicionirati servere kako bi se izbjeglo mrežno kašnjenje. U tu svrhu koristi se CDN (eng. *Content delivery network*) kako bi se serveri koji stoje iza neke web aplikacije, rasporedili u više svjetskih lokacija ovisno o potrebi.
- **Infrastruktura -** Ovisno o veličini i potrebama web aplikacije potrebno je odabrati adekvatnu hardversku infrastrukturu. Danas, Cloud servisi nude vrlo sofisticirane funkcionalnosti, te osim velikog izbora tipa infrastrukture, postoji i mogućnost prilagođavanja infrastrukture tijekom rada aplikacije.
- **Integracije –** Web aplikacije često koriste gotove servise razvijene od drugih IT kompanija. Bitno je razmotriti kolike su latencije takvih servisa jer to također utječe na latenciju aplikacije koja ga koristi.

(Mohan, 2022)

Dakle, prethodno spomenuti faktori mogu utjecati na performanse web aplikacije koje je moguće detektirati pomoću nekoliko ključnih indikatora performansi (eng. *Key performance indicators - KPI*):

- **Vrijeme odgovora** (eng. *Response time*) – Vrijeme odgovora servera web mjesta na zahtjev korisnika glavni je pokazatelj performansi i ako dođe do nekih defekata, vrijeme odgovora će se povećati. Iako na vrijeme odgovora može utjecati veći broj zahtjeva korisnika, ono nam općenito ukazuje na propuste na strani servera koji mogu usporiti i samo jedan zahtjev.
- **Konkurentnost/propusnost** (eng. *Concurrency/throughput*) – Konkurentnost i propusnost ukazuju na to koliki je maksimalni broj korisničkih zahtjeva web aplikacija spremna podnijeti u nekom vremenskom intervalu, a da je pritom vrijeme odgovora unutar zadane granice.
- **Dostupnost** (eng. *Availability*) – Dostupnost mjeri da li je web aplikacija spremna pružati zadovoljavajuće vrijeme odgovora na zahtjeve korisnika kroz duži period. Primjerice, aplikacija prvih pola sata može pružati zadovoljavajuće vrijeme odgovora, ali nakon više sati može se osjetiti pad performansi zbog neefikasnog upravljanja memorijom ili nakupljanja nezavršenih procesa.

Temeljem ovih indikatora formiraju se različiti testovi performansi spomenuti u sekciji 2.4. Kod formiranja testova bitno je uzeti u obzir realnost te izbjeći neke scenarije koji se nikada u

stvarnosti neće desiti. Npr. ako u je u produkciji izmjereno maksimalno 200 zahtjeva u sekundi, nema potrebe testirati njih 400. Isto tako korisnički scenariji se moraju odvijati u realnom vremenu jer nemoguće je da stvarni korisnik u nekoliko milisekundi odradi kretanje kroz 10 stranica web mjesta, zapravo će mu trebati nekoliko sekundi da razmisli o svom sljedećem koraku.
(Mohan, 2022; Shivakumar, 2020; Molyneaux, 2009)

2.6.3. Testiranje na strani klijenta

Pomoću alata za testiranje slanjem zahtjeva putem API-a i HTTP zahtjeva nije moguće u potpunosti izmjeriti stvarno vrijeme odgovora i performanse iz perspektive korisnika. Razlika je u tome što alati ne simuliraju web preglednike koji prilikom dobivanja odgovora izvršavaju *frontend (FE)* kod koji renderira grafičko sučelje temeljem:

- HTML koda koji predstavlja glavni okvir za izgled sučelja
- CSS kod temeljem kojeg se uključuju stilovi
- JS (*Javascript*) skripte koje omogućuju izvršavanje raznih dinamičkih akcija

Sa korisničke strane dodatno na performanse utječe učitavanje raznih medijskih sadržaja kao što su slike i video materijali, razni dokumenti (PDF) i ostali dodaci. Također JS kod može također uspostavljati konekcije do baza podataka ili pozivati razne API-eve što također može utjecati na konačno vrijeme odgovora. (Mohan, 2022)

Stoga ako odgovor sa servera traje 1 sekundu, dodatni procesi na strani klijenta mogu trajati bar još 1 sekundu ili čak i dulje. Ponekad da bi se performanse poboljšale na strani klijenta uvodi se koncept *Lazy Loading* na način da se prvo učita osnovni kostur stranice (HTML), a tek onda se skriptama učitava detaljni sadržaj ili se učita samo vidljivi dio stranice, a ostatak kada korisnik pomiče stranicu, npr. prema dolje.
Faktori koji utječu na klijentske performanse:

- **Kompleksnost FE koda –** JS kod web aplikacije može se sastojati od mnogo API zahtjeva od kojih svaki može trajati koliko i URL zahtjev za dohvaćanje skripti što može uvelike utjecati na klijentske performanse
- **Mreža za distribuciju sadržaja** (eng. *Content delivery network - CDN)* – CDN predstavlja grupu servera na različitim geolokacijama. Na njima se pohranjuje dio sadržaja web aplikacije kako bi se smanjila vremena odgovora na različite zahtjeve od strane FE skripti.

- **DNS** (*Domain name server*) – Svaki zahtjev korisnika za određenom web aplikacijom može se keširati u DNS serverima. Pa tako nakon prvog zahtjeva dohvaćanje IP adrese servera sprema se u lokalne DNS servere te su sljedeći odgovori znatno brži.
- **Mrežno kašnjenje –** Mrežno kašnjenje može ovisiti o propusnosti mreže korisnika, ali isto tako i količini podataka koje web aplikacija dohvaća pomoću FE skripti.
- **Cache preglednika –** Današnji web preglednici imaju vrlo napredne mogućnosti kratkoročnog spremanja sadržaja web aplikacije kao što su slike, video, kolačići i slično. Na taj način kod novog zahtjeva već posjećene stranice određeni sadržaj bit će unaprijed dostupan lokalno.
- **Prijenos podataka –** Ako se tijekom odvijanja određenih akcija događaju prijenosi velike količine podataka tada može doći do znatnih kašnjenja u učitavanju web stranice.

Također i za klijentski stranu postoje metrike pomoću kojih se mjeri kvaliteta performansi web aplikacije:

- **Učitavanje prvog sadržaja –** Odnosi se na vrijeme koje je potrebno da bi se renderirao prvi element DOM-a koji predstavlja smisleni sadržaj (slika, tekst…). Drugim riječima vrijeme unutar kojeg korisnik može vidjeti prve akcije učitavanja sadržaja.
- **Vrijeme do interakcije** – Vrijeme unutar kojeg korisnik može uspostaviti prvu interakciju sa web aplikacijom. Ponekad se interaktivni elementi mogu učitati, ali ne reagiraju na akcije što može izazvati frustraciju.
- **Učitavanje najzahtjevnijeg sadržaja –** Vrijeme koje je potrebno da se najopsežniji elementi učitaju (velike sekcije teksta, velike slike)
- **Pomicanje sadržaja –** Mjeri koliko često dolazi do pojave pomicanja stranice ili sadržaja prilikom učitavanja. Ponekad se na web stranicama sadržaj može pomaknuti nakon inicijalnog učitavanja što uzrokuje loše iskustvo korisnika.

Postoje naravno još i neke dodatne detaljne metrike osim ovih navedenih, a Google je izdvojio troje kao ključne: učitavanje najzahtjevnijeg sadržaja, vrijeme do prve interakcije te pomicanje sadržaja. Većina klijentskih alata pa tako i Googleov *Lighthouse* upravo ove tri metrike mjeri i izdvaja kao ključne. (Mohan, 2022; Shivakumar, 2020)

2.7. Dostupni alati i tehnologije

Na tržištu softvera postoje brojni alati za testiranje performansi. Iako svi imaju zajedničku svrhu postoje neke razlike i specifičnosti kod svakog alata. Stoga, potrebno je dobro razmotriti specifikacije kako bi odabrani alat zadovoljio potrebe projekta u odnosu na složenost i troškove. Također, svaki alat je namijenjen za serversku ili za klijentsku stranu.

2.7.1.Alati za testiranje na serverskoj strani

Alati za testiranje performansi na serverskoj strani još se nazivaju i alati za generiranje opterećenja (eng. *Load performance testing tools*). Dakle, kreiranjem testova pomoću ovih alata osigurava se provjera performansi s obzirom na zadano opterećenje, te se otkrivaju potencijalni defekti i greške u *backend* kodu i infrastrukturi servera.

JMeter

Slika 5, Logo JMeter alata (Kartaca, n.d.)

Trenutno najpopularniji *open source* alat za serversko testiranje. Ovaj višedretveni okvir omogućava testiranje performansi na statičkim i dinamičkim resursima i web aplikacijama. Omogućava simuliranje velike količine opterećenja za neki određeni ili grupu servera i mreža. Podržava različite tipove generiranja opterećenja i protokole: *HTTP, HTTPS, SOAP/REST, FTP, LDAP*… (Galeza, 2022)

Prednosti:

- Testovi se kreiraju jednostavno bez naprednog znanja programiranja jer uz komandnu liniju postoji i opcija rada s grafičkim korisničkim sučeljem.
- Nakon izvršenja testova moguće je generirati razne izvještaje i analize podataka
- Proširiv je uz razne ekstenzije (*plugin*) koje se mogu dodati pomoću *Plugin Managera*

- Može izvršavati testove opterećenja nad raznim vrstama aplikacija jer se mogu slati zahtjevi na API, baze podataka, redove poruka (eng. Message Queues).
- Podržava distribuirane testove opterećenja
- Postoji velika baza korisnika pa tako i velika podrška

Nedostaci:

- Kod generiranja opterećenja uz puno virtualnih korisnika zauzeće resursa može biti iznimno veliko
- Ne podržava JavaScript i AJAX zahtjeve
- Nije skriptni alat te zbog toga automatizacija može biti vrlo zahtjevna

Gatling

Slika 6, Gatling logo (TestMatick, n.d.)

Open-source alat za testiranje performansi i nudi bilježenje korisničkih tokova. Posebnost je ta da uz to što se testovi mogu pisati u popularnom jeziku Java, nudi pisanje istih u jeziku *Scala* što pojednostavljuje samu strukturu koda. Uz to nudi i vrlo detaljne izvještaje te se vrlo lako integrira u CI/CD *pipeline.* Podržava slijedeće protokole: *HTTP(s)/1, JMS, SOAP, MQTT, HTTP/2, JDBC*, i *WebSockets*. (Galeza, 2022)

Prednosti:

- Analiza testova je vrlo detaljna i prikazana grafički
- Izvještaji su spremljeni u obliku HTML datoteke
- Moguće je dodati ekstenzije za testiranje *Kafka, RabbitMQ* i *JDBC*
- Najnovija verzija uz Javu i Scalu omogućava i pisanje testova u Kotlinu

Nedostaci:

- Osoba zadužena za pisanje testova mora poznavati osnove jednog od jezika (Scala, Java, Kotlin)
- Neke od korisnih funkcionalnosti su dostupne samo u plaćenoj verziji (*Gatling Enterprise*): distribucijski način, *live* izvještaji, modificiranje izvještaja

K6

Slika 7, K6 logo (Github Grafana, n.d.)

Relativno novi alat na tržištu koji je u usponu te je označen kao moderni alat za developere. Alat je izrađen od strane *Grafana Labs* i k6 zajednice u jeziku *Go*, ali testovi se pišu u JavaScriptu. Podražava *HTTP/1.1, HTTP/2, WebSockets,* and *gRPC* protokol. (Galeza, 2022)

Prednosti:

- CLI alata sa vrlo jednostavnim API-em
- Testovi se pišu u JavaScriptu s podrškom lokalnih i udaljenih modula
- Sa provjerama i granicama (eng. *Thresholds*) osigurava jednostavno i testiranje okrenuto cilju
- Moguće je definirati specifične metrike
- Drugačiji je od ostalih alata za testiranje jer efikasnije koristi hardverske resurse

Nedostaci:

- Zadani izvještaji nisu detaljni, ali je moguće to donekle nadomjestiti sa dodatkom *k6-reporter*
- Nema distribuiranih testova, ali se zato veća opterećenja mogu postići na plaćenoj *K6 Cloud* platformi
- Potrebno je solidno poznavanje JavaScript/TypeScript jezika

2.7.2.Alati za testiranje na klijentskoj strani

Za velike projekte najbitniji dio je testirati opterećenje za serversku stanu. No, za potpunu provjeru performansi sustava potrebno je testirati i klijentsku stranu. Klijentske/*frontend* skripte ponekad čine i do 90% vremena učitavanja web stranice. Mnogi problemi kod klijentske strane se mogu vrlo lako uočiti kod testiranja od strane developera (kod inicijalne provjere

funkcionalnosti) i primjenom dobrih principa pisanja koda, ali klijentsko testiranje služi kao dodatna provjera i može ukazati na vrlo male detalje.

WebPageTest

Slika 8, WebPageTest logo (Github WebPageTest, n.d.)

Postoji *online* verzija koja omogućava pokretanje testova sa različitih lokacija koristeći stvarne web preglednike, a nudi i prilagođavanje mrežnih uvjeta. Analiza testiranja daje ocjene za kritična područja koja utječu na performanse (detalje o vremenu učitavanja te veličini web stranice). Plaćena verzija alata omogućava brojne zanimljive pogodnosti kao što je testiranje gotovih slučajeva koji dolaze kao prijedlog. (Galeza, 2022)

Lighthouse

Slika 9, Google Lighthouse logo (Justia, n.d.)

Open-source alat izrađen od Googlea koji omogućava analizu performansi, pristupačnost, najbolje prakse, SEOP (eng. *Search optimization*) web mjesta. Integriran je direktno u *Chrome DevTools*, a može se i instalirati kao dodatak za web preglednik (*eng. plugin*). Uobičajeno testiranje ovim alatom se implementira u CI/CD procese gdje su tada generirani razni komentari kod dodavanja koda ako dođe do nekih poteškoća te mogu biti pohranjeni na Cloud platforme. (Galeza, 2022)

Sitespeed

Slika 10, Sitespeed logo (Sitespeed, n.d.)

Predstavlja grupu manjih alata za testiranje i baziran je na Docker kontejnerima. Zapravo, predstavlja platformu koja za testiranje koristi Lighthouse, Google PageSpeed Insights i WebPageTest. Također, može se prilagođavati i integrirati s alatima za analitiku testova kao što je Grafana. Postoji i opcija da se koriste i ostali alati za testiranje, osim navedenih iznad. (Galeza, 2022)

3. Praktična usporedba testiranja

U ovom poglavlju slijedi praktični dio rada gdje ćemo pomoću dva odabrana alata za testiranje performansi odraditi testiranje na klijentskoj i serverskoj strani te napraviti usporedbu. Opisat ćemo odabrane alate i web mjesto koje će nam poslužiti u svrhu testiranja, definirati ćemo model za usporedbu, izvesti samo testiranje na obje strane te naposlijetku odraditi analizu dobivenih rezultata i zaključiti usporedbu.

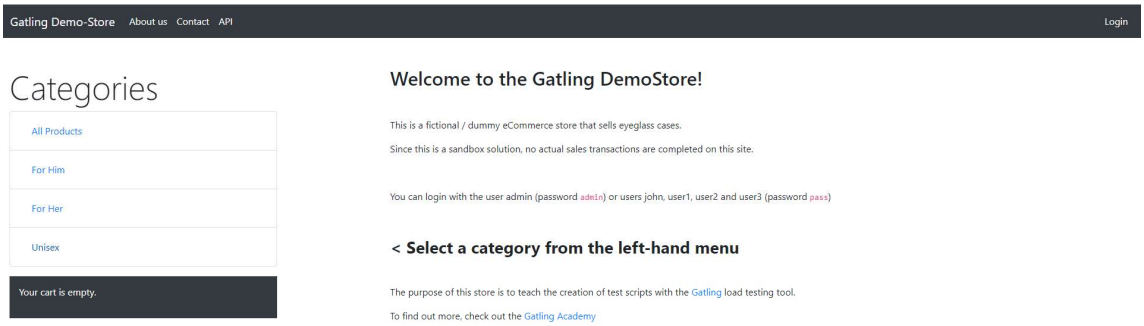
3.1. Odabrani alati i resursi

Za testiranje performansi na serverskoj strani odabran je alat Gatling. Alat je odabran jer je jedan od popularnijih na tržištu i pruža mnoge napredne mogućnosti za konfiguriranje testova. Testovi se mogu pisati u 3 vrlo popularna jezika, a u ovom radu odabrana je Scala jer pisanje samog koda pojednostavljeno je u odnosu na Javu, ali opet pruža napredne objektno-orijentirane mogućnosti te je zbog toga vrlo pogodan za pisanje testova.

Za testiranje performansi na klijentskoj strani odabran je vrlo poznati Google Lighthouse. Vrlo napredan alat koji se može pronaći u Google Chrome kao alat za developere. Vrlo je lagan za koristiti i daje vrlo detaljnu statistiku učitavanja klijentskih skripti i same vizualne prezentacije web mjesta. Postoji i ekstenzija za JavaScript pomoću koje se uz definiranje jednostavnih skripti možemo dobiti rezultate kao i kod korištenja Chrome alata što je od velike koristi kod automatizacije samog procesa.

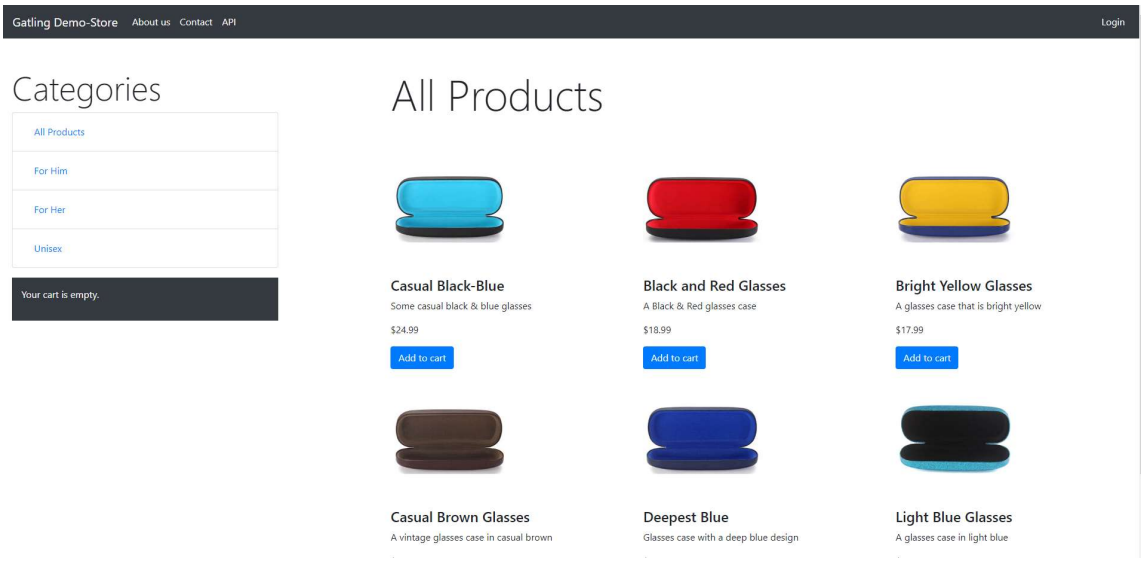
Objekt testiranja za obje strane jest web mjesto kreirano specijalno za testiranje opterećenja od strane Gatling tima *Gatling Demo-Store* (URL: https://demostore.gatling.io/). Ovo je web mjesto odabrano jer je posebno dizajnirano da izdrži nešto veće opterećenje i testiranje neće biti identificirano kao prijetnja. Mogli bismo testirati i neko realno web mjesto, ali tada ćemo prije ili kasnije biti blokirani od strane anti-bot sistema koji su implementirani u produkcijskim okruženjima. Zato se većinom web mjesta testiraju u testnim okruženjima koja nisu dostupna javnom, ne sadrže zaštitne sisteme i prilagođena su specijalno za testiranja. Web mjesto Gatling tima zbog mogućnosti podnošenja većeg opterećenja bit će idealno za testiranje serverske strane, ali isto tako moguće je testirati i klijentsku stranu jer je dovoljno kompleksno i sadrži dovoljnu količinu klijentskih skripti. Isto tako da bi usporedba bila moguća obje strane moraju testirati isto web mjesto.

Web mjesto sastoji se od početne stranice.

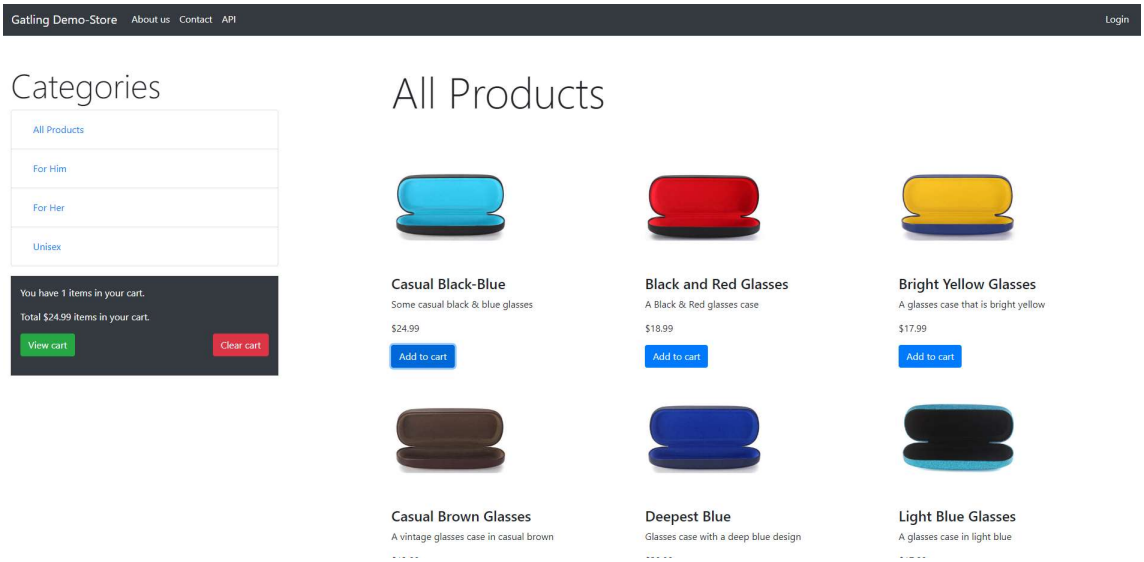


demostore.gatling.io/category/unisex

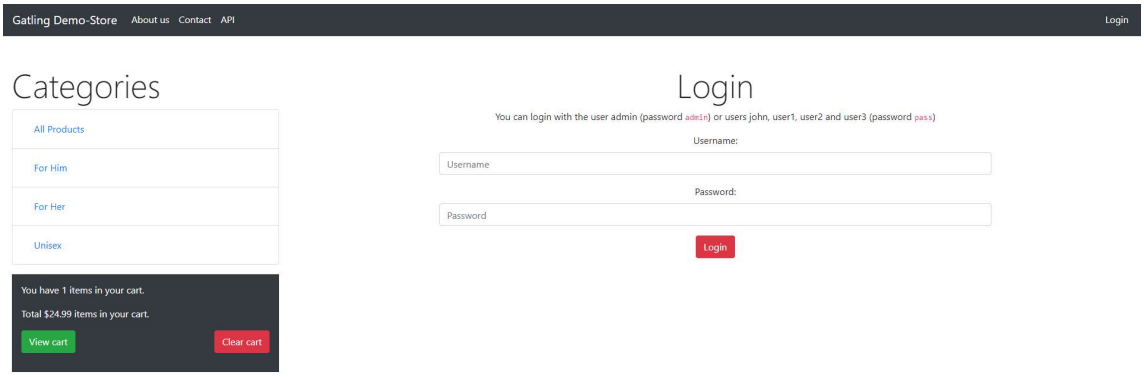
Na početnoj stranici moguće je iz desnog izbornika odabrati nekoliko kategorija proizvoda.



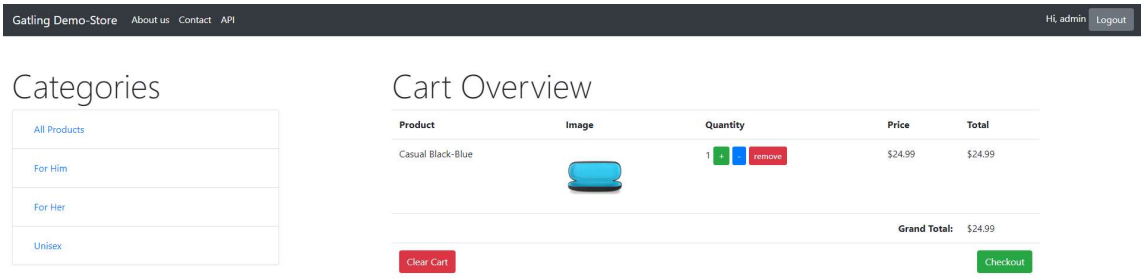
Proizvodi se mogu detaljno pregledati klikom na neku stavku iz liste ili se mogu dodati u košaricu.



Košarica se može pregledati, ali prethodno se potrebno prijaviti. Na stranici za prijavu u opisu ispisani su podaci za nekoliko testnih korisničkih računa.



Nakon prijave moguće je odraditi pregled košarice i potvrditi kupnju. Nakon što je kupnja potvrđena simulira se plaćanje.



Web mjesto je dizajnirano specijalno za svrhu testiranja i iz tog razloga vrlo je jednostavno, ali otporno na veća opterećenja i dovoljno kompleksno kako bi se pokazale karakteristike oba alata i napravila usporedba između klijentske i serverske strane.

3.2. Model usporedbe pristupa i alata

Za usporedbu serverske i klijentske strane testiranja performansi kreirat ćemo model usporedbe. Svrha ovog modela je da usporedimo neke zajedničke značajke obje strane kako bi istaknuli razlike među njima. Nakon što izvedemo testiranje na obje strane dobit ćemo jasnu sliku koje su sličnosti/razlike i prednosti/mane između ova dva pristupa testiranja performansi. Također, pomoću rezultata usporedbe moći ćemo donijeti zaključak u koja vrsta testiranja je prikladnija s obzirom na vrstu i opseg projekta razvoja web sustava.

U prvom dijelu usporedbe model će se sastojati od vremena odgovora testiranih web stranica. Model će zapravo biti tablica usporedbe (Tablica 1). Oba pristupa testiranja se odnose na testiranje performansi pa možemo zaključiti da zasigurno postoje neke sličnosti temeljem kojih se može izvršiti direktna usporedba. Stoga, u tablici usporedbe navedene su web stranice *Gatling Demo-Store* web mjesta odabrane za testiranje. Svaka od web stranica može biti testirana u oba pristupa, a rezultat usporedbe biti će vrijeme odgovora koje bi kako možemo zaključiti iz poglavlja 2.6, trebalo biti veće na strani klijenta. Ovim modelom to želimo dokazati i na primjeru detaljno objasniti što je uzrok toj razlici te koja je svrha mjerenja različitih vremena.

Vrijeme odgovora (sekunde)	**Serverska strana – Gatling**	**Klijentska strana – Lighthouse**
Home page		
About Us		
All Products		
For Him		
For Her		
Unisex		
Product		
Login		

Tablica 1 Model usporedbe vremena odgovora klijentskog i serverskog pristupa testiranju.

Drugi dio modela usporedbe su općenite značajke i funkcionalnosti. Dakle, ovdje ćemo navesti listu značajki koje su specifične za određeni pristup, a neke od njih mogu biti karakteristika oba pristupa. Rezultat se bilježi ocjenom zadovoljenja određene značajke ili funkcionalnosti. S obzirom u kojoj mjeri je određena značajka zadovoljena ocjena može biti:

- Ne podržava
- Minimalno
- Prosječno
- Zadovoljavajuće
- Odlično

Rezultate svih značajki/funkcionalnosti ćemo kod analize (nakon sekcije izvođenja testova) detaljno opisati i istaknuti koje mogućnosti svaka značajka pruža te koje vrijednosti dodaje procesu testiranja performansi.

Značajke i funkcionalnosti (Ocjena 0-5)	**Serverska strana – Gatling**	**Klijentska strana – Lighthouse**
Testiranje opterećenja		
Testiranje klijentskih skripti		
Mogućnost integracije u CI/CD		
Testiranje API-a		
Mogućnost automatizacije		
Dokumentacija i podrška		

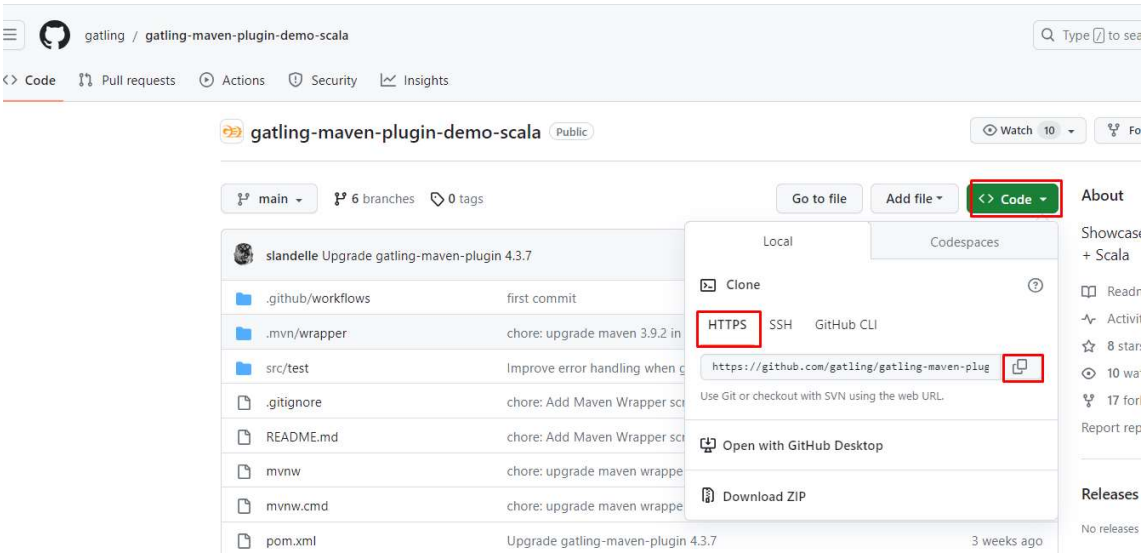
Jednostavnost implementacije		
Mogućnost prilagodbe testova		
Podržani jezici programske implementacije		

Tablica 2, Model usporedbe općenitih karakteristika/značajki klijentskog i serverskog pristupa testiranju.

3.3. Serversko testiranje – Gatling

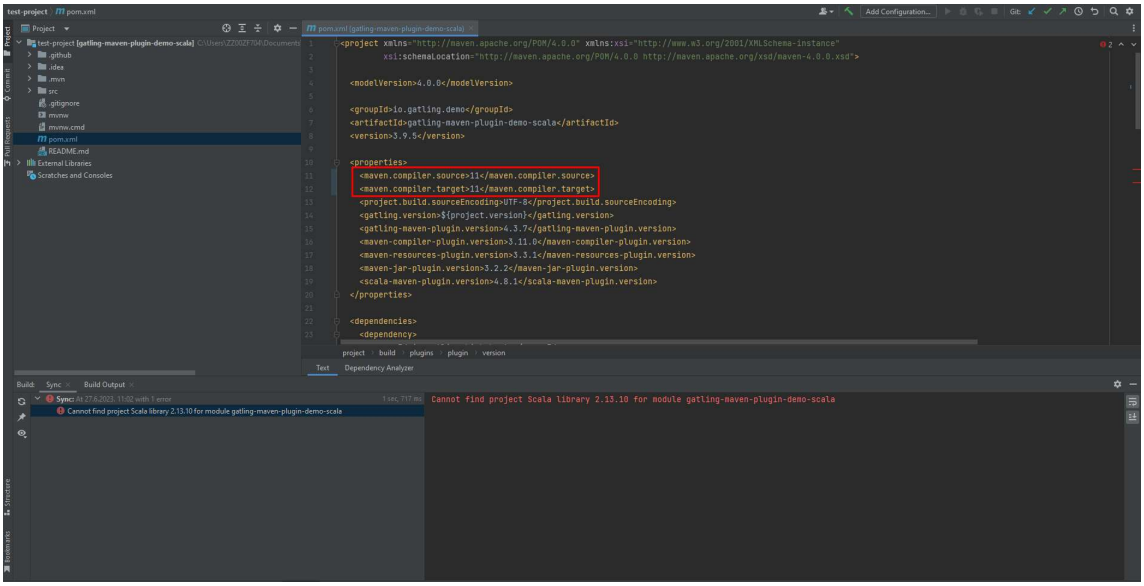
3.3.1. Inicijalizacija projekta

Kako bismo postavili Gatling projekt potrebno je preuzeti starter projekt sa sljedeće poveznice: https://github.com/gatling/gatling-maven-plugin-demo-scala. Ovaj projekt sadrži sve potrebne postavke i ovisnosti pa je stoga nakon preuzimanja odmah moguće početi sa pisanjem testova. Preuzeti starter projekt moguće je kloniranjem GitHub repozitorija ili preuzimanjem ZIP datoteke.

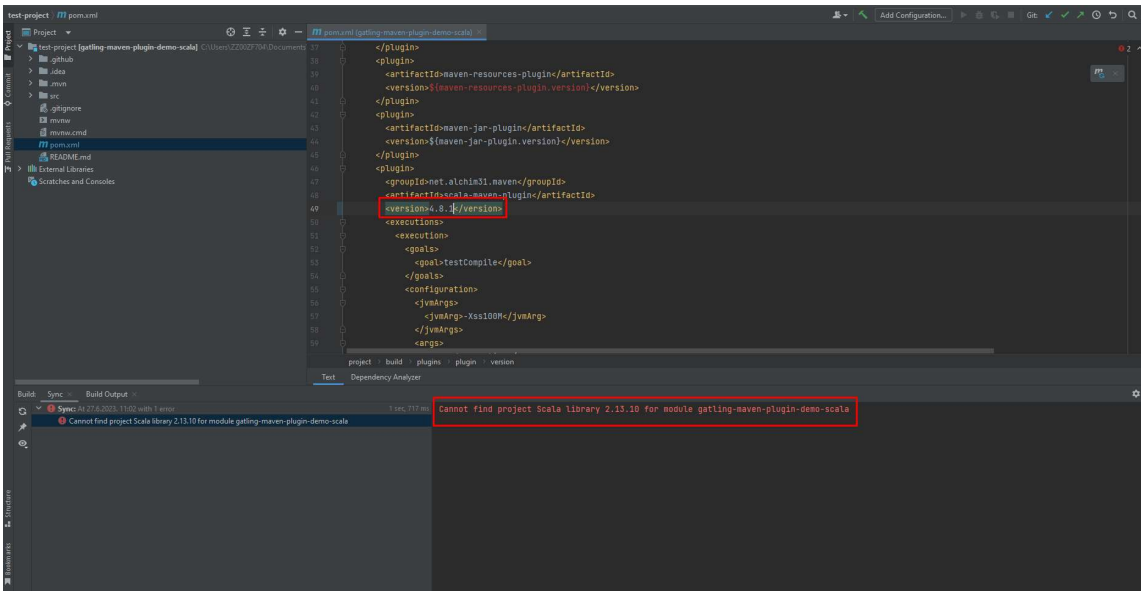


Za kloniranje odaberemo direktorij na lokalnom računalu te izvršimo komandu *git clone https://github.com/gatling/gatling-maven-plugin-demo-scala.git test-project*. Projekt otvorimo u nekom IDE (eng. *Integrated development environment* – alat razvojnog okruženja), u ovom slučaju to će biti IntelliJ IDE.

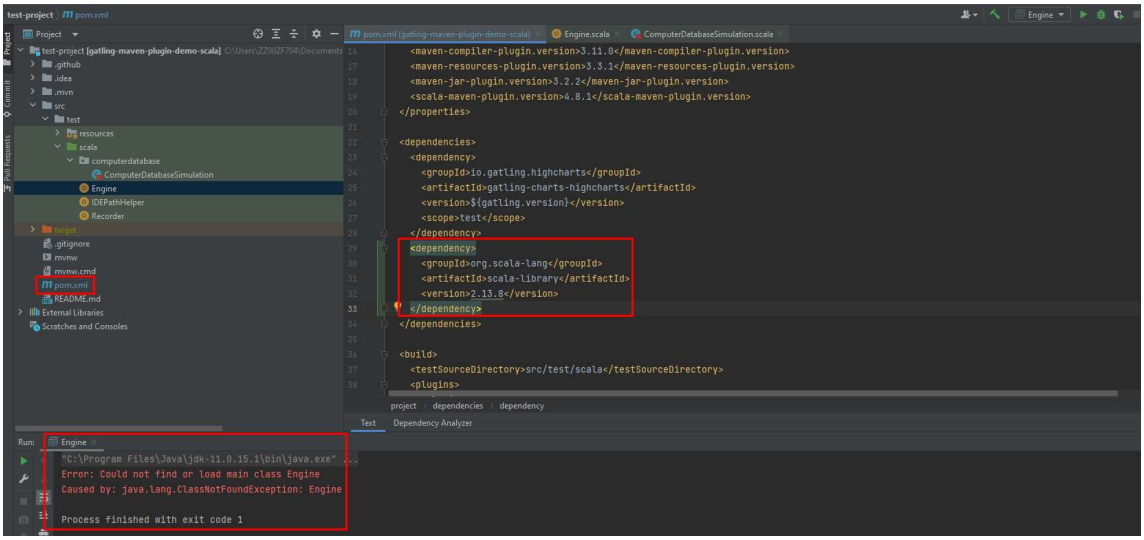
Nakon što je projekt otvoren odradit ćemo nekoliko osnovnih koraka kako bi bili sigurni da su sve postavke ispravno podešene. Prvo otvorimo *pom.xml* datoteku te provjerimo da li je Java verzija 11. Isto tako treba se pobrinuti da je ista verzija Jave instalirana i na računalu.



Ako se prilikom sinkronizacije projekta pojavi greška da se ne može pronaći *Scala library,* ponovo u *pom.xml* treba promjeniti verziju *maven-scala-plugin* u konkretnu (ovdje je to bila najnovija 4.8.1).



Ako kod pokretanja testa pomoću *Engine* klase dođe do greške jer klasa nije pronađena, potrebno je dodati novi *dependency* za *scala-library* u *pom.xml* unutar sekcije <*dependencies*>.



Nakon ovih promjena ne bi više trebalo biti poteškoća i sve je spremno za pisanje prvih testova te za njihovo uspješno pokretanje.

3.3.2. Struktura projekta

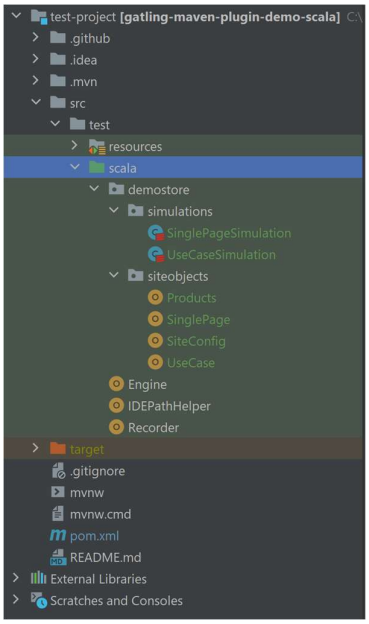
Gatling projekt za testiranje serverske strane web aplikacije sastoji se od dvije vrste testova: statičkih i dinamičkih.

Prva skupina testova sa serverske strane bit će statički testovi. Naziv im je *statički* jer će svaki testni slučaj slati zahtjeve na samo jednu specifičnu web strancu *Gatling Demo-Store-a.* Upravo na taj način želimo ispitati koliku količinu zahtjeva neka web stranica može podnijeti, a da pri tome prosječno vrijeme odgovora bude zadovoljeno sukladno zadanim kriterijima.

Druga skupina testova jesu dinamički testovi koji sadrže definirane korisničke slučajeve korištenja. Drugim riječima, to su testovi koji oponašaju korisnika prilikom korištenja web mjesta. Također, i u ovom slučaju definirat ćemo veći broj korisnika kako bismo usporedili vremena odgovora sa zadanim kriterijima. Kriteriji za obje vrste testova bit će detaljno objašnjeni u nastavku.

Ispod na slici možemo vidjeti strukturu samog Gatling projekta. Paket u kojem se nalazi sav kod jest *src/test/scala/demostore.* U ovom direktoriju nalazi se glavna i predefinirana *Engine.scala* datoteka koju smo dobili preuzimanjem GitHub starter projekta. U ovoj datoteci definiran je *Engine* koji služi za pokretanje aplikacije i automatski detektira sve definirane testove te pruža opcije za pokretanje. Sljedeća Scala datoteka jest *IDEPathHelper.scala* koja sadrži definirane vrijednosti za razne putanje koje koristi *Engine.* Treća predefinirana datoteka jest *Recorder.scala* koji služi kao mini alat koji omogućuje generiranje koda testova temeljem mrežnog prometa kojeg korisnik odradi. Primjerice, na taj način možemo uključiti *Recorder* i

u nekom web pregledniku kretati se web mjestom te će pritom taj promet biti zabilježen. Zatim će testovi biti generirani temeljem zabilježenog prometa.



Nadalje na slici možemo također vidjeti da *demostore* paket sadrži slijedeća dva paketa: *simulations* i *siteobjects*. U s*imulations* paketu nalaze se datoteke koje predstavljaju Scala simulacije. Prva je pod nazivom *SinglePageSimulation* i u njoj su definirani prethodno spomenuti statički testovi. U simulaciji *UseCaseSimulation* definirani dinamički testovi koji imitiraju korisničke slučajeve korištenja. U *siteobjects* paketu sadržane su datoteke s pomoćnim objektima koji su korišteni u simulacijama. Simulacije i pomoćne objekte detaljnije ćemo opisati u nastavku.

3.3.3. Statički testovi

Statičkim testovima testiramo odabrane web stranice zasebno. Time želimo provjeriti da li je trenutno serversko okruženje sposobno zadovoljiti zadane zahtjeve (zadano opterećenje). Također ovakvi testovi služe i za provjeru stabilnosti *backend* logike u kodu i obično se izvršavaju na nižim okruženjima prije dolaska nove verzije aplikacije na produkciju. Na taj način žele se otkloniti eventualne pogreške u logici koda koje bi mogle prouzročiti financijsku štetu, pa čak i pad sustava.

U sljedećem isječku koda nalazi se definirana klasa *SinglePageSimulation* koja nasljeđuje klasu *Simulation* iz *io.gatling.core* paketa. Na taj način naša klasa je prepoznata kao simulacija kod pokretanja testova pomoću *Engine.* Simulacija se generalno sastoji od dva dijela: definicije testova/scenarija i postavljanja scenarija za izvršavanje.

```
1.  package demostore.simulations
2.
3.  import demostore.siteobjects.{Products, SinglePage}
4.  import demostore.siteobjects.SiteConfig._
5.  import io.gatling.core.Predef._
6.  import io.gatling.core.structure.ScenarioBuilder
7.
8.  class SinglePageSimulation extends Simulation {
9.
10.   def homePage: ScenarioBuilder = scenario("HomePage")
11.     .exec(SinglePage.homePage)
12.
13.   def aboutUsPage: ScenarioBuilder = scenario("AboutUsPage")
14.     .exec(SinglePage.aboutUsPage)
15.
16.   def allProductsPage: ScenarioBuilder = scenario("AllProducts")
17.     .exec(SinglePage.allProductsPage)
18.
19.   def forHimPage: ScenarioBuilder = scenario("ForHimPage")
20.     .exec(SinglePage.forHimPage)
21.
22.   def forHerPage: ScenarioBuilder = scenario("ForHerPage")
23.     .exec(SinglePage.forHerPage)
24.
25.   def unisexPage: ScenarioBuilder = scenario("UnisexPage")
26.     .exec(SinglePage.unisexPage)
27.
28.   def productPage: ScenarioBuilder = scenario("ProductPage")
29.     .exec(SinglePage.productPage(Products.casualBlackBlueProduct.name))
30.
31.   def loginPage: ScenarioBuilder = scenario("LoginPage")
32.     .exec(SinglePage.loginPage)
33.
34.   setUp(
35.     homePage.inject(basicInjectionConfig)
36.     .andThen(
37.       aboutUsPage.inject(basicInjectionConfig)
38.     ).andThen(
39.       allProductsPage.inject(basicInjectionConfig)
40.     ).andThen(
41.       forHimPage.inject(basicInjectionConfig)
42.     ).andThen(
43.       forHerPage.inject(basicInjectionConfig)
44.     ).andThen(
45.       unisexPage.inject(basicInjectionConfig)
46.     ).andThen(
47.       productPage.inject(basicInjectionConfig)
48.     ).andThen(
49.       loginPage.inject(basicInjectionConfig)
50.     )
51.   ).protocols(httpProtocol)
52.
53. }
```

Definirano je 8 statičkih scenarija od kojih svaki definira slanje zahtjeva na specifičnu stranicu *Gatling Demostore-a.* Scenarij se definira na slijedeći način, uzmimo primjerice *HomePage* sa linije 10:

- Poziva *scenario()* funkcija koja poprima parametar naziv web stranice. Ova funkcija ima ulogu konstruktora i vraća objekt *ScenarioBuilder*

- Zatim se poziva funkcija *exec()* u koju u našem slučaju stavljamo *homePage* objekt iz *SinglePage* datoteke koja sadrži pomoćne objekte. *homePage* objekt (isječak koda ispod, linija 7) jest *exec()* funkcija koja sadrži definirani HTTP zahtjev prema korijenskoj *HomePage* stranici.

```
1.  package demostore.siteobjects
2.
3.  import io.gatling.core.Predef._
4.  import io.gatling.http.Predef._
5.
6.  object SinglePage {
7.    def homePage = exec(
8.        http("HomePage")
9.          .get("/")
10.     )
11.
12.   def aboutUsPage = exec(
13.       http("AboutUsPage")
14.         .get("/about-us")
15.     )
16.
17.   def allProductsPage = exec(
18.       http("AllProducts")
19.         .get("/category/all")
20.     )
21.
22.   def forHimPage = exec(
23.       http("ForHimPage")
24.         .get("/category/for-him")
25.     )
26.
27.   def forHerPage = exec(
28.       http("ForHerPage")
29.         .get("/category/for-her")
30.     )
31.
32.   def unisexPage = exec(
33.       http("UnisexPage")
34.         .get("/category/unisex")
35.     )
36.
37.   def productPage(productName: String) = exec(
38.       http("ProductPage")
39.         .get(s"/product/$productName")
40.     )
41.
42.   def loginPage = exec(
43.       http("LoginPage")
44.         .get("/login")
45.     )
46. }
```

Na isti način definirani su i ostali statički scenariji, a razlikuju se po nazivu i putanji do određene web stranice definiranoj u *SinglePage* pomoćnom objektu.

Drugi dio simulacije odnosi se na postavljanje izvršavanja testova (*SinglePageSimulation*, linija 34). Testovi se izvršavaju redom, a svaki sljedeći test započinje

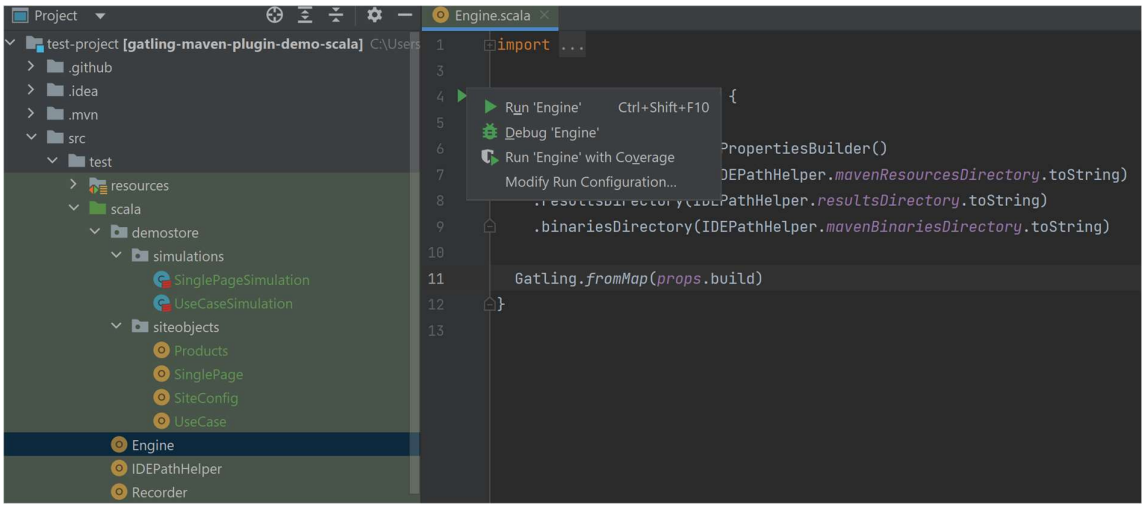
izvršavanje kada prethodni završi. Dakle, funkcija *setUp()* kao parametar prima lanac povezanih scenarija funkcijom *andThen()*. Iza svakog scenarija slijedi i funkcija *inject()* koja kao parametar prima konfiguraciju generiranja opterećenja. U ovom slučaju ta konfiguracija jest nazvana *basicInjectionConfig* i definirana je u pomoćnom objektu *SiteConfig*.

```
1.  package demostore.siteobjects
2.
3.  import io.gatling.core.Predef._
4.  import                    io.gatling.core.controller.inject.open.{ConstantRateOpenInjection,
    OpenInjectionStep}
5.  import io.gatling.http.Predef._
6.  import io.gatling.http.protocol.HttpProtocolBuilder
7.
8.  import scala.concurrent.duration._
9.  import scala.language.postfixOps
10.
11. object SiteConfig {
12.
13.   val httpProtocol: HttpProtocolBuilder =
14.     http.baseUrl("http://demostore.gatling.io/")
15.
16.   val smallPause: FiniteDuration = 1 seconds
17.
18.   val bigPause: FiniteDuration = 3 seconds
19.
20.   val basicInjectionConfig: ConstantRateOpenInjection =
21.     constantUsersPerSec(10).during(10)
22.
23.   val rampUsersConfig: OpenInjectionStep =
24.     rampUsers(10).during(10.seconds)
25.
26. }
```

SiteConfig objekt sadrži osnovne vrijednosti koje koriste simulacije. Konkretno *basicInjectionConfig* definira opterećenje sa konstantnim brojem korisnika (10) kroz 10 sekundi. To znači da u 10 sekundi u scenarij će se ubaciti 100 korisnika od kojih će svaki izvršiti zadani scenarij. Pošto je scenarij za *homePage* jednostavan i šalje samo jedan zahtjev prema početnoj stranici, možemo zaključiti da ćemo u 10 sekundi generirati 100 zahtjeva. Ista stvar će se ponoviti za svaku stranicu i scenarij definiran unutar *setup()* metode.

U nastavku *setup()* metode dolazi .protocols() u kojeg prosljeđujemo također objekt iz *SiteConfig* (linija 12) pod imenom *httpProtocol*. On definira protokol koji će se koristiti za slanje zahtjeva i samu domenu web mjesta na koje šaljemo zahtjeve.

Testiranje se pokreće tako da otvorimo *Engine* datoteku i kliknemo na *Play* ikonu/*Run 'Engine'*.



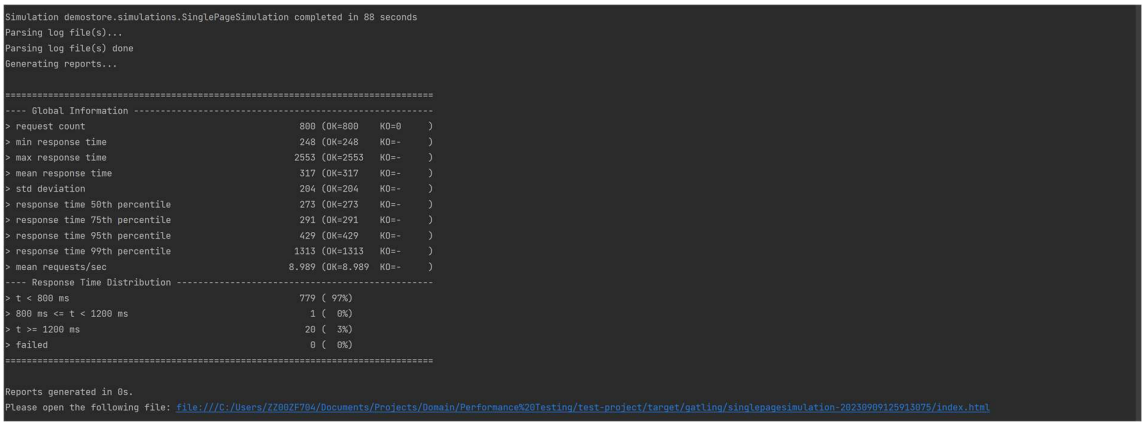
Zatim će se u konzoli pojaviti izbornik sa detektiranim simulacijama (koje se nalaze unutar *simulations* paketa). Odabiremo opciju 0 jer se ona odnosi na prethodno opisanu simulaciju i pritisnemo *Enter*. Zatim unesemo neki opis testa i ponovo *Enter*.

```
"C:\Program Files\Java\jdk-11.0.15.1\bin\java.exe" ...
Choose a simulation number:
     [0] demostore.simulations.SinglePageSimulation
     [1] demostore.simulations.UseCaseSimulation
0
Select run description (optional)
Testing
```

Simulacija započinje i svakih 5 sekundi ispisuje informaciju o poslanim zahtjevima, aktivnim korisnicima i korisnicima na čekanju.

```
Simulation demostore.simulations.SinglePageSimulation started...

================================================================================
2023-09-09 14:59:18                                           5s elapsed
---- Requests ------------------------------------------------------------------
> Global                                                   (OK=47     KO=0     )
> HomePage                                                 (OK=47     KO=0     )

---- ProductPage ---------------------------------------------------------------
[                                                                          ]  0%
          waiting: 100    / active: 0      / done: 0
---- HomePage ------------------------------------------------------------------
```

Nakon što se simulacije/testovi završe dobijemo statistiku poslanih zahtjeva.



Prema dobivenoj statistici možemo izvući neke bitne stavke:

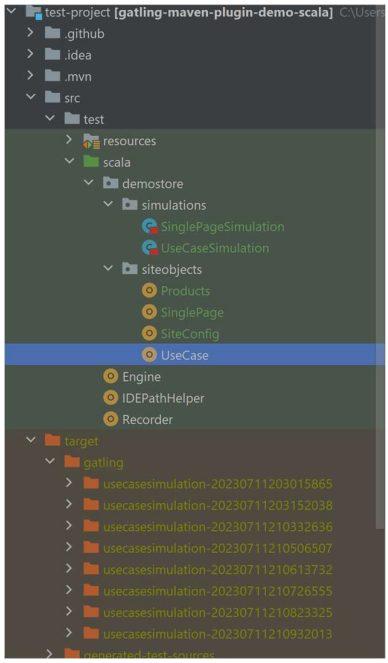
- u ovom slučaju možemo vidjeti da je poslano točno 800 zahtjeva prema *Gatling Demostore* web mjestu. Pošto smo imali 8 scenarija u kojima smo kroz 10 sekundi poslali 100 zahtjeva, dobivamo broj od ukupno 800 zahtjeva.
- Minimalno vrijeme odgovora je 248ms (milisekundi)
- Maksimalno vrijeme odgovora je 2553ms
- Prosječno vrijeme odgovora je 317ms
- 95 posto odgovora je unutar 429ms
- 99 posto odgovora je unutar 1313ms
- Prosječan broj zahtjeva po sekundi je ~ 9
- Nije bilo pogrešaka prilikom slanja zahtjeva, tj. nismo dobili grešku kao odgovor
- Link na detaljno grafičko izvješće

Ako se vodimo nekim općim standardom, također preporučenom od Google-a, da bi se web mjesto smatralo dobro ocjenjenim tada bi 95 posto odgovora trebalo biti unutar pola sekunde (500ms) i 99 posto odgovora unutar 2 sekunde (2000ms). Možemo vidjeti da *Gatling Demostore* zadovoljava ove uvjete. Kao što smo spomenuli bitno je da web mjesto zadovoljava ove uvjete jer tada pozitivno utječe na korisničko iskustvo, a također je i bolje rangirano od strane raznih tražilica.

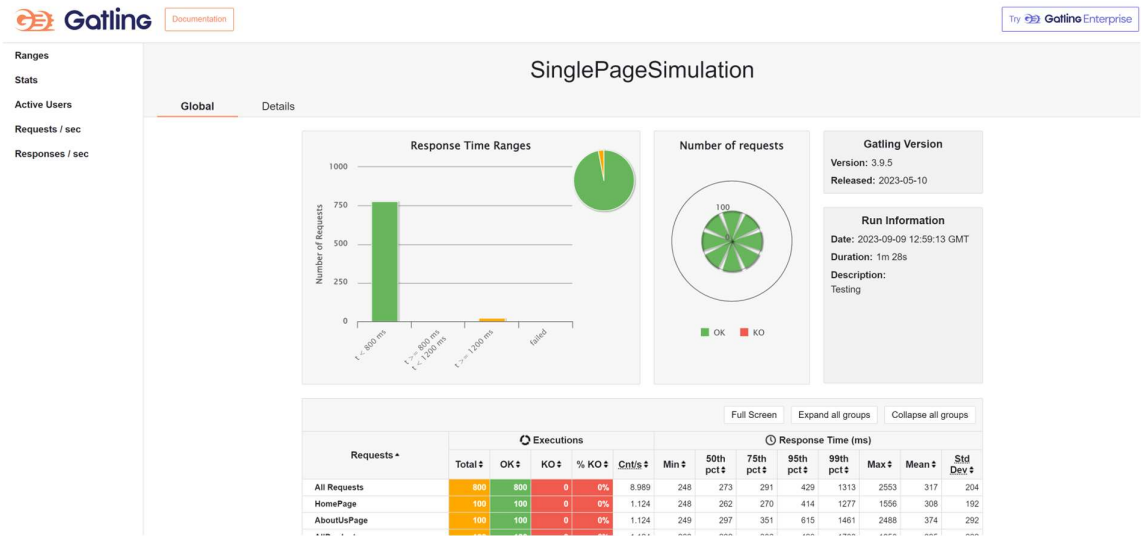
Isto tako, možemo zaključiti da kada je opterećenje maksimalno 10 zahtjeva u sekundi tada će ovo web mjesto pružati zadovoljavajuću uslugu. Ako mjerenjem na produkciji primijetimo da je promet veći tada bi se sama serverska infrastruktura trebala skalirati prema gore. Ovakvi testovi mogu služiti kao kontrola za nova izdanja web mjesta. Primjerice, na početnu stranicu dodamo novu funkcionalnost dohvaćanja podataka iz baze (recimo neke novosti) koja je implementirana neoptimalno. Prije nego novo izdanje dođe do produkcije, pokrenuli bismo testove za testno okruženje. Ako prilikom izvršenja testova primijetimo pad u

performansama (95 posto zahtjeva je unutar 800ms, a mora biti unutar 500ms), tada znamo da je došlo do nekog propusta i potrebno je to ispraviti prije nego nova verzija stigne do produkcije.

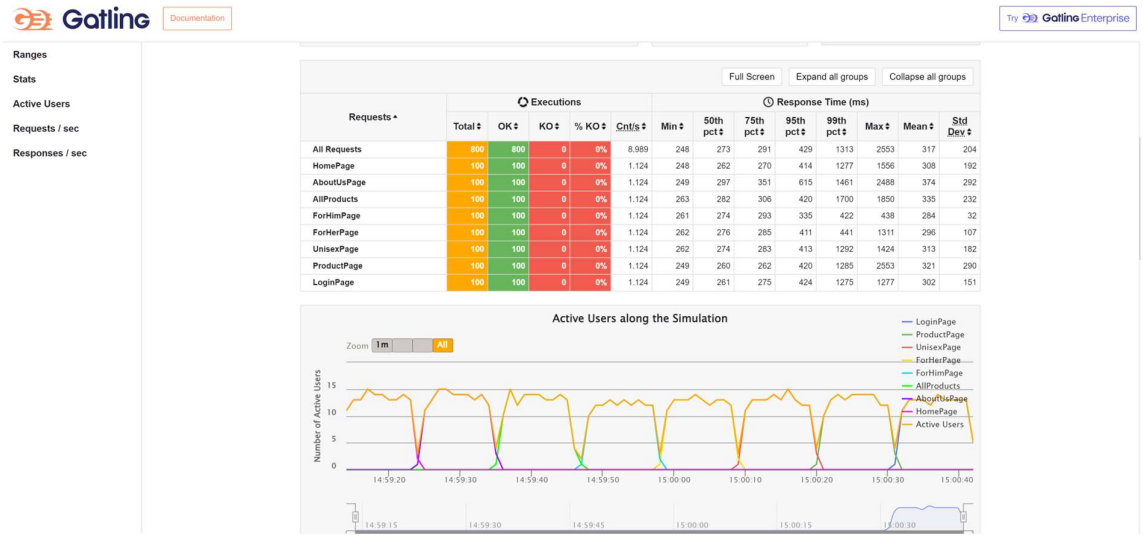
Dodatno *Gatling* automatski generira izvješća u obliku statičnih HTML datoteka. Ta izvješća generiraju se po završetku procesa testiranja i spremaju se u direktorij */target/gatling*.



Pritiskom na ponuđeni link kojeg dobijemo u završnom izvješću otvorit će se statička web stranica u lokalnom pretraživaču.



U *Global* sekciji možemo pronaći detaljne informacije o vremenima odgovora za svaku stranicu pojedinačno. Također niže se nalaze razni dijagrami koji pokazuju statistiku o aktivnim korisnicima, vremenu odgovora, broju poslanih zahtjeva i slično kroz vremenski period testiranja.

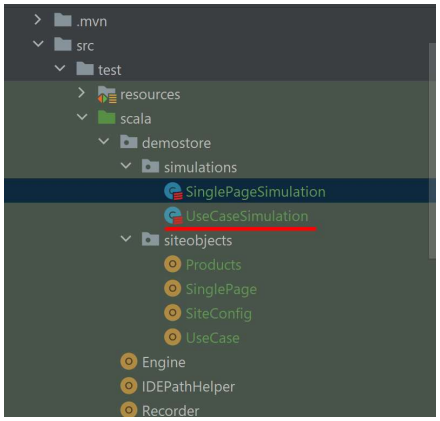


Također prethodno navedene detalje moguće je vidjeti u sekciji *Details* gdje se u lijevom izborniku može odabrati željeni scenarij.

3.3.4. Dinamički testovi

Kod dinamičkih testova opterećenje se generira pomoću scenarija korisničkih slučajeva korištenja. Drugim riječima pomoću ovih testova pokušat ćemo imitirati stvarne scenarije koji se odvijaju tijekom stvarnog korištenja nekog web mjesto od strane više korisnika.

Spomenuli smo prethodno da u Gatling projektu jest definirana i druga simulacija pod imenom *UseCaseSimulation.scala.*



UseCaseSimulation je također klasa koje nasljeđuje klasu *Simulation.* U ovoj klasi definirano je slijedeće:

- Scenarij koji imitira stvarno korištenje web mjesta
- Postavljanje scenarija

```
1.  package demostore.simulations
2.
3.  import demostore.siteobjects.SiteConfig.{httpProtocol, rampUsersConfig}
4.  import demostore.siteobjects.UseCase
5.  import io.gatling.core.Predef._
6.  import io.gatling.core.structure.ScenarioBuilder
7.
8.  class UseCaseSimulation extends Simulation {
9.
10.   val scn: ScenarioBuilder = scenario("User Activity")
11.     .exec(UseCase.browseBasicPages)
12.     .exec(UseCase.loginIntoProfile)
13.     .exec(UseCase.browseAllProductsPage)
14.     .exec(UseCase.browseForHimPage)
15.     .exec(UseCase.browseForHerPage)
16.     .exec(UseCase.browseUnisexPage)
17.     .exec(UseCase.checkout)
18.     .exec(UseCase.logout)
19.
20.   setUp(
21.     scn.inject(rampUsersConfig)
22.   ).protocols(httpProtocol)
23.
24. }
```

U ovom slučaju kreiran je jedan scenarij (linija 16) koji se sastoji od više koraka, točnije od više zahtjeva prema različitim web stranicama i API-ima. Scenarij pod imenom *User Activity* sastoji se od 8 koraka, a svaki korak predstavlja jedno izvršavanje niza akcija. Koraci i akcije definirani su datoteci *UseCase* (isječak koda ispod). Stoga ovaj scenarij predstavlja jedan mogući i detaljan slučaj korištenja. Prvo korisnik pregledava osnovne stranice (Početna i O nama). Zatim vrši prijavu u korisnički račun pretražuje nekoliko stranica sa kategorijama proizvoda, otvara proizvode i dodaje ih u košaricu. Zatim pregledava košaricu i vrši kupnju. Na kraju se odjavljuje.

```
1.  package demostore.siteobjects
2.
3.  import io.gatling.core.Predef._
4.  import io.gatling.core.structure.{ChainBuilder, ScenarioBuilder}
5.  import io.gatling.http.Predef._
6.
7.  object UseCase {
8.
9.    def browseBasicPages: ChainBuilder =
10.     exec(SinglePage.homePage)
11.       .pause(SiteConfig.smallPause)
12.       .exec(SinglePage.aboutUsPage)
13.       .pause(SiteConfig.bigPause)
14.
15.   def browseAllProductsPage: ChainBuilder =
16.     exec(SinglePage.allProductsPage)
17.       .pause(SiteConfig.bigPause)
18.       .exec(SinglePage.productPage(Products.casualBlackBlueProduct.name))
19.       .pause(SiteConfig.bigPause)
20.       .exec(addProductToCart(Products.casualBlackBlueProduct))
21.       .pause(SiteConfig.smallPause)
22.
23.   def browseForHimPage: ChainBuilder =
24.     exec(SinglePage.forHimPage)
25.       .pause(SiteConfig.bigPause)
26.       .exec(SinglePage.productPage(Products.casualBlackBlueProduct.name))
27.       .pause(SiteConfig.bigPause)
28.       .exec(addProductToCart(Products.casualBlackBlueProduct))
29.       .pause(SiteConfig.smallPause)
30.
31.   def browseForHerPage: ChainBuilder =
32.     exec(SinglePage.forHerPage)
33.       .pause(SiteConfig.bigPause)
34.       .exec(SinglePage.productPage(Products.perfectPinkProduct.name))
35.       .pause(SiteConfig.bigPause)
36.       .exec(addProductToCart(Products.perfectPinkProduct))
37.       .pause(SiteConfig.smallPause)
38.
39.   def browseUnisexPage: ChainBuilder =
40.     exec(SinglePage.unisexPage)
41.       .pause(SiteConfig.bigPause)
42.       .exec(SinglePage.productPage(Products.curvedBrownProduct.name))
43.       .pause(SiteConfig.bigPause)
44.       .exec(addProductToCart(Products.curvedBrownProduct))
45.       .pause(SiteConfig.smallPause)
46.
47.   def loginIntoProfile: ChainBuilder =
48.     exec(
49.       http("Load Login Page")
50.         .get("/login")
51.         .check(css("#_csrf", "content").saveAs("csrfValue"))
52.     ).pause(SiteConfig.bigPause)
53.       .exec(
54.         http("Customer Login Action")
55.           .post("/login")
56.           .formParam("_csrf", "${csrfValue}")
57.           .formParam("username", "admin")
58.           .formParam("password", "pass")
59.       ).pause(SiteConfig.smallPause)
60.
61.   def checkout: ChainBuilder =
62.     exec(
63.       http("Load Cart")
64.         .get("/cart/view")
65.     )
66.       .pause(SiteConfig.bigPause)
67.       .exec(
68.         http("Checkout")
69.           .get("/cart/checkout")
70.       ).pause(SiteConfig.smallPause)
```

```
71.
72.   def logout: ChainBuilder =
73.     exec(
74.       http("Logout")
75.         .post("/logout")
76.         .formParam("_csrf", "${csrfValue}")
77.     ).pause(SiteConfig.smallPause)
78.
79.   private def addProductToCart(product: Products.Product): ChainBuilder = {
80.     exec(
81.       http("Add Product To Cart")
82.         .get(s"/cart/add/${product.id}")
83.     )
84.   }
85.
86. }
```

Definirani koraci unutar *UseCase* objekta zapravo predstavljaju skup akcija slanja zahtjeva i na određene web stranice *Gatling Demostorea* i odrađivanja pauzi. Pauze se odrađuju kako bi se realnije imitirale korisničke akcije, te služe kao vrijeme u kojem korisnik razmišlja o slijedećoj akciji. One su definirane unutar *SiteConfig* datoteke u sekciji 3.3.3. Imamo definirane sljedeće korake (isječak koda gore, počevši od linije 9):

- **browseBasicPages** – imitira pregled osnovnih stranica. Prvo šalje zahtjev na početnu stranicu, radi malu pauzu, šalje zahtjev na *aboutUs* stranicu te radi veliku pauzu.
- **browseAllProductsPage** – pregled stranice sa svim proizvodima otvaranje jednog proizvoda i dodavanje u košaricu
- **browseForHimPage** – pregled stranice sa muškim proizvodima, otvaranje i dodavanje jednog u košaricu
- **browseForHerPage** – pregled stranice sa ženskim proizvodima, otvaranje i dodavanje jednog u košaricu
- **browseUnisexPage** – pregled stranice sa neutralnim proizvodima, otvaranje i dodavanje jednog u košaricu
- **loginIntoProfile** – prijava u korisnički profil kako bi bilo moguće dovršiti kupovinu proizvoda. Ovdje se kod učitavanja stranice za prijavu sprema CSRF token koji je generiran od strane servera i šalje se ponovo kod slanja POST metode serveru zajedno sa korisničkim imenom i lozinkom
- **checkout** – Pregled stavki u košarici i izvršavanje kupnje.
- **logout** - odjava korisničkog računa te se prilikom odjave također koristi CSRF token spremljen u sesiju prilikom prijave.

UseCase također sadrži još jednu privatnu metodu za dodavanje proizvoda u košaricu (linija 79) koju koriste neke od prethodno navedenih metoda.

Drugi dio *UseCaseSimulation* jest definicija izvršavanja testa. Ovdje se ponovo u *setUp()* metodu prosljeđuje parametar s konfiguracijom opterećenja i naposlijetku se dodaje protokol. Ovaj puta konfiguracija jest *rampUsersConfig* koja se može pronaći u datoteci *SiteConfig*. Ova konfiguracija će u periodu od 10 sekundi ubaciti 10 korisnika od kojih će svaki izvršiti definirani scenarij. Sada je scenarij složeniji i sastoji se od više zahtjeva u odnosu kako je to bilo kod statičkih testova (SinglePageSimulation).

Simulaciju pokrećemo kao i kod statičkog testa. Pokrećemo *Engine* objekt, odabiremo ovaj put opciju 1 te dodajemo opis. Testiranje će započeti i svakih 5 sekundi do završetka testa prikazat će se trenutna statistika.

```
"C:\Program Files\Java\jdk-11.0.15.1\bin\java.exe" ...
Choose a simulation number:
     [0] demostore.simulations.SinglePageSimulation
     [1] demostore.simulations.UseCaseSimulation
1
Select run description (optional)
Testing
```

Primjerice u 15. sekundi svih 10 korisnika je aktivno, izvršeno je 56 zahtjeva ukupno. Vidimo da *HomePage, AboutUsPage i LoginPage* su izvršene 10 puta što znači da su ih svi korisnici izvršili. Ostale stranice nisu izvršene još 10 puta jer su neki korisnici ubačeni u proces nešto kasnije.

```
================================================================================
2023-09-09 19:39:32                                          15s elapsed
---- Requests ------------------------------------------------------------------
> Global                                                   (OK=56     KO=0     )
> HomePage                                                 (OK=10     KO=0     )
> AboutUsPage                                              (OK=10     KO=0     )
> Load Login Page                                          (OK=10     KO=0     )
> Customer Login Action                                    (OK=8      KO=0     )
> Customer Login Action Redirect 1                         (OK=8      KO=0     )
> AllProducts                                              (OK=7      KO=0     )
> ProductPage                                              (OK=3      KO=0     )

---- User Activity -------------------------------------------------------------
[--------------------------------------------------------------------------]  0%
          waiting: 0      / active: 10     / done: 0
================================================================================
```

Na posljednjem izvještaju u 53. sekundi vidimo da je ukupno izvršeno 230 zahtjeva u 53 sekunde. Dakle, svaki korisnik je izvršio ukupno 23 zahtjeva.

```
================================================================================
2023-09-09 19:40:11                                          53s elapsed
---- Requests ------------------------------------------------------------------
> Global                                                   (OK=230    KO=0     )
> HomePage                                                 (OK=10     KO=0     )
> AboutUsPage                                              (OK=10     KO=0     )
> Load Login Page                                          (OK=10     KO=0     )
> Customer Login Action                                    (OK=10     KO=0     )
> Customer Login Action Redirect 1                         (OK=10     KO=0     )
> AllProducts                                              (OK=10     KO=0     )
> ProductPage                                              (OK=40     KO=0     )
> Add Product To Cart                                      (OK=40     KO=0     )
> ForHimPage                                               (OK=10     KO=0     )
> ForHerPage                                               (OK=10     KO=0     )
> UnisexPage                                               (OK=10     KO=0     )
> Load Cart                                                (OK=10     KO=0     )
> Load Cart Redirect 1                                     (OK=10     KO=0     )
> Checkout                                                 (OK=10     KO=0     )
> Checkout Redirect 1                                      (OK=10     KO=0     )
> Logout                                                   (OK=10     KO=0     )
> Logout Redirect 1                                        (OK=10     KO=0     )

---- User Activity -------------------------------------------------------------
[########################################################################]100%
          waiting: 0      / active: 0      / done: 10
================================================================================
```

U izvještaju o testu možemo vidjeti slijedeće informacije:

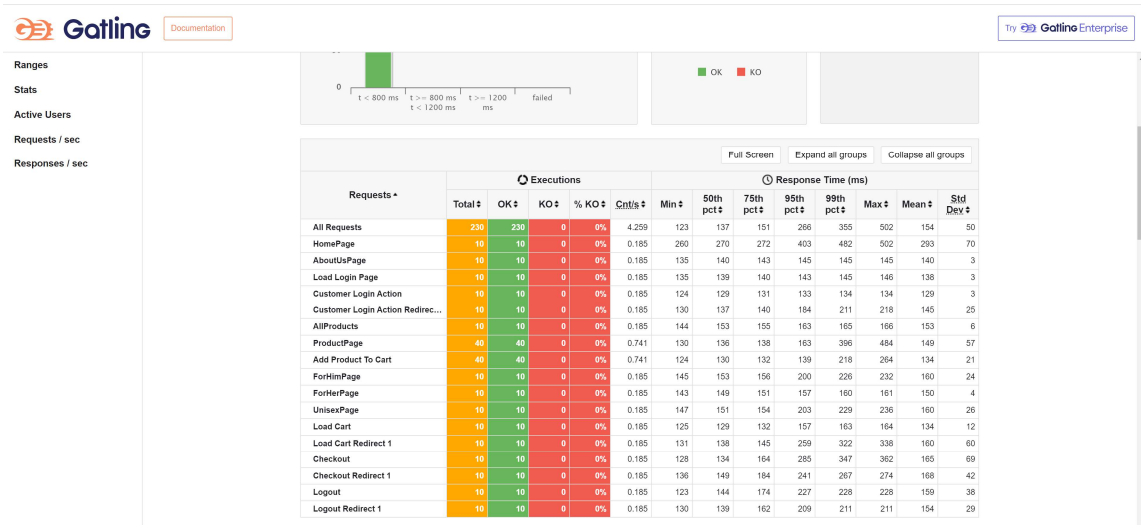
- Broj poslanih zahtjeva 230
- Minimalno vrijeme odgovora 123ms
- Maksimalno vrijeme odgovora 502ms
- Prosječno vrijeme odgovora 154ms
- 95 posto odgovora je unutar 266ms
- 99 posto odgovora je unutar 355ms
- Prosječan broj zahtjeva u sekundi ~ 4,2
- Nije bilo grešaka kod slanja zahtjeva

```
================================================================================
---- Global Information --------------------------------------------------------
> request count                                    230 (OK=230    KO=0     )
> min response time                                123 (OK=123    KO=-     )
> max response time                                502 (OK=502    KO=-     )
> mean response time                               154 (OK=154    KO=-     )
> std deviation                                     50 (OK=50     KO=-     )
> response time 50th percentile                    137 (OK=137    KO=-     )
> response time 75th percentile                    151 (OK=151    KO=-     )
> response time 95th percentile                    266 (OK=266    KO=-     )
> response time 99th percentile                    355 (OK=355    KO=-     )
> mean requests/sec                              4.259 (OK=4.259  KO=-     )
---- Response Time Distribution ------------------------------------------------
> t < 800 ms                                       230 (100%)
> 800 ms <= t < 1200 ms                              0 (  0%)
> t >= 1200 ms                                       0 (  0%)
> failed                                             0 (  0%)
================================================================================

Reports generated in 0s.
Please open the following file: file:///C:/Users/ZZ00ZF704/Documents/Projects/Domain/Performance%20Testing/test-project/target/gatling/usecasesimulation-20230909173917114/index.html

Process finished with exit code 0
```

Možemo uočiti da su vremena odgovora bolja u odnosu na rezultate kod statičkih testova. Razlog tome jest da sada možda jesmo izvršili više zahtjeva, ali kroz dulji vremenski interval

pa je prosječan broj zahtjeva u sekundi bio i više nego duplo manji, 4.2 dok je kod statičkih testova bio 9.

Ako otvorimo generirani izvještaj možemo vidjeti da sve stranice u prosjeku imaju jednako prosječno vrijeme odgovora oko 150ms, jedino *HomePage* odstupa sa skoro 300ms. Uglavnom i ovdje su kriteriji zadovoljavajući jer ukupno 95 posto zahtjeva je unutar 500ms, dok je 99 posto zahtjeva unutar 2000ms.

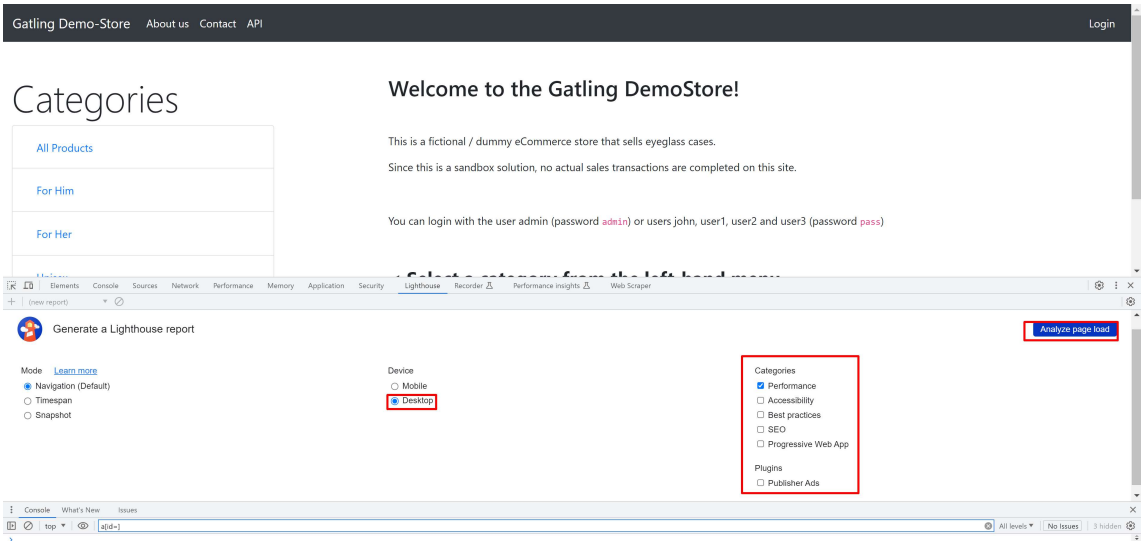


Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
All Requests	230	230	0	0%	4.259	123	137	151	266	355	502	154	50
HomePage	10	10	0	0%	0.185	260	270	272	403	482	502	293	70
AboutUsPage	10	10	0	0%	0.185	135	140	143	145	145	145	140	3
Load Login Page	10	10	0	0%	0.185	135	139	140	143	145	146	138	3
Customer Login Action	10	10	0	0%	0.185	124	129	131	133	134	134	129	3
Customer Login Action Redirec...	10	10	0	0%	0.185	130	137	140	184	211	218	145	25
AllProducts	10	10	0	0%	0.185	144	153	155	163	165	166	153	6
ProductPage	40	40	0	0%	0.741	130	136	138	163	396	484	149	57
Add Product To Cart	40	40	0	0%	0.741	124	130	132	139	218	264	134	21
ForHimPage	10	10	0	0%	0.185	145	153	156	200	226	232	160	24
ForHerPage	10	10	0	0%	0.185	143	149	151	157	160	161	150	4
UnisexPage	10	10	0	0%	0.185	147	151	154	203	229	236	160	26
Load Cart	10	10	0	0%	0.185	125	129	132	157	163	164	134	12
Load Cart Redirect 1	10	10	0	0%	0.185	131	138	145	259	322	338	160	60
Checkout	10	10	0	0%	0.185	128	134	164	285	347	362	165	69
Checkout Redirect 1	10	10	0	0%	0.185	136	149	184	241	267	274	168	42
Logout	10	10	0	0%	0.185	123	144	174	227	228	228	159	38
Logout Redirect 1	10	10	0	0%	0.185	130	139	162	209	211	211	154	29

3.4. Klijentsko testiranje – Lighthouse

3.4.1. Chrome Lighthouse alat

U sekciji 3.1 spomenuli smo da je Lighthouse alat koji se može pronaći kao dio razvojnih alata u Google Chrome pregledniku. Pomoću tog ugrađenog alata možemo također analizirati *Gatling Demostore* web mjesto. Za primjer uzmimo početnu stranicu: http://demostore.gatling.io/. Otvorimo ovu stranicu u Chrome pregledniku -> desni klik -> *Inspect.* Na taj način otvorit ćemo prozor razvojnih alata (*eng. Developer tools*). U tom prozoru pronađemo sekciju *Lighthouse* te ćemo dobiti prizor kao na slici ispod.



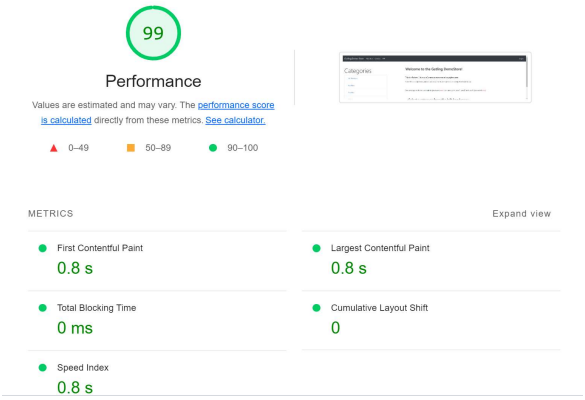
Lighthouse nije samo alat za testiranje i analizu performansi web stranice nego i ostalih područja popust pristupačnosti, najboljih praksi, optimizacije za pretraživanje i slično. U otvorenom prozoru možemo odabrati kategorije koje želimo analizirati. U našem slučaju to jesu performanse. Odaberemo da uređaj bude stolno računalo (eng. *Desktop*) i na desnoj strani kliknemo gumb za početak analize (*Analyze page load*). Nakon što analiza završi dobit ćemo vrlo detaljan izvještaj o performansama web stranice. Performanse Lighthouse ocjenjuje na skali 0-100 te postoje 3 kategorije rezultata (Lighthouse performance scoring, n.d.):

- Ocjena 0-49 – Vrlo loše performanse
- Ocjena 50-89 – Potrebna su poboljšanja
- Ocjena 90-100 – Zadovoljavajuće performanse

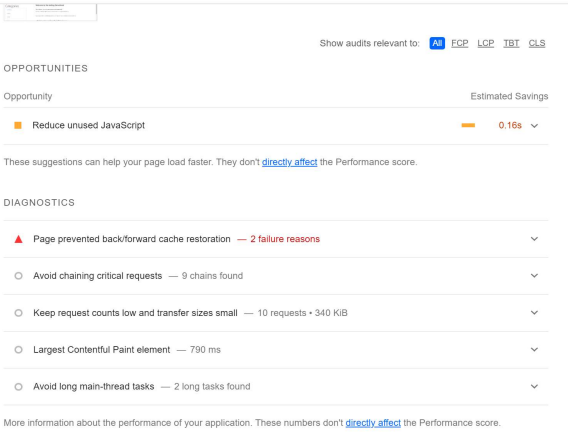
U našem slučaju izvještaj pokazuje da performanse za početnu stranicu imaju ocjenu 99 što znači da trenutne performanse pozitivno utječu na korisničko iskustvo. Također u izvještaju možemo pronaći detalje o metrikama temeljem kojih je to izračunato (Lighthouse performance scoring, n.d.):

- **Pojava prvog sadržaja** (eng. *First Contentful Paint*) – Predstavlja vrijeme od početka učitavanja web stranice do pojave prvog sadržaja (tekst, slika, bilo koji vidljivi element). U našem slučaju to je bilo 0.8 sekundi što je ocjenjeno pozitivno i prema Google-u poželjno je da ova vrijednost bude manja od 1.8 sekundi
- **Učitavanje najvećeg sadržaja** (eng. *Largest Contentful Paint*) – Vrijeme od početka učitavanja stranice do završetka renderiranja i vidljivosti najvećeg elementa. Prema Google preporuci ova vrijednost bi trebala biti manja od 2.5 sekundi. U našem slučaju ta vrijednost je 0.8 što je izuzetno dobro.

- **Ukupno vrijeme blokiranja** (eng. *Total Blocking Time*) – Vrijeme od pojave prvog sadržaja do vremena mogućnosti prve interakcije. Preporuka je da ovo vrijeme bude manje od 200ms (0.2 sekunde). U izvještaju ta vrijednost je 0ms, što znači da od pojave prvog vidljivog sadržaja korisnik je odmah mogao vršiti interakciju, primjerice klik na poveznicu navigacije.
- **Ukupno pomicanje sadržaja** (eng. *Cumulative Layout Shift*) – Ovo je mjera najveće vrijednosti (neočekivanog) pomicanja sadržaja (eng. Layout shift score) koje se može pojaviti tijekom pregleda sadržaja stranice. Preporučeno je da ta mjera bude manja od 0.1, a u našem slučaju ona jest 0 što znači da je to izvrsno i nema pomicanja sadržaja na ovoj web stranici.
- **Indeks brzine** (eng. *Speed Indeks*) – Pokazuje koliko je brzo sadržaj stranice popunjen i vidljiv korisniku. Preporučena vrijednost jer 3.4 sekunde ili manje. Prema tome i ova mjera je zadovoljena za testiranu web stranicu i iznosi 0.8 sekundi.



Niže u dobivenom izvještaju možemo pronaći preporuke pomoću kojih akcija se performanse mogu poboljšati. Iako je su performanse testirane stranice vrlo dobre postoji još malo mjesta za poboljšanje kod učitavanja smanjenjem neiskorištenog koda JavaScripta. Ta promjena bi ocjenu performansi podigla na 100.



Google (Lighthouse performance scoring, n.d.) također navodi da za dobre performanse dovoljno je imati ocjenu 90 naviše. Podići ocjenu od 80-90 ponekad može zahtijevati jednake resurse kao i za podići ocjenu od 90-94 ili 99-100. Naravno da bi ocjena 100 dodala još neka mala poboljšanja kod korisničkog iskustva, ali to je ponekad teško postići zbog kompleksnosti zahtjeva web mjesta. Stoga, bitno je postaviti optimalne ciljeve u kontekstu klijentskih performansi.

Pomoću Google Lighthouse analizom web stranice dobili smo vrlo detaljne rezultate performansi. Ovo sada bilo odrađeno pomoću ugrađenog Chrome alata, ali Lighthouse je moguće integrirati pomoću JavaScript biblioteka i API-ja te na taj način postići automatizaciju prethodno pokazanih testova. Integracija pomoću JavaScripta slijedi u nastavku.

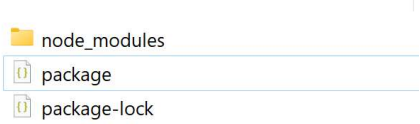
3.4.2. Postavljanje projekta

Projekt koji će automatizirati Lighthouse testiranje performansi vrlo jednostavno je napisati u pomoću JavaScripta i node.js. Postavljanje projekta pomoću node.js vrlo je jednostavno i dobro dokumentirano na *npm* (*Node package manager*) repozitoriju (https://www.npmjs.com/package/playwright-lighthouse). Automatizacija je bazirana na biblioteci *playwright-lighthouse*, gdje se *playwright* koristi za pokretanje preglednika u pozadini, a *lighthouse* naravno za provođenje testiranja.

Dakle za postavljanje projekta potrebno je prvo instalirati node.js i *npm. Zatim* kreirati korijenski direktorij, otvoriti terminal i izvršiti slijedeću komandu *npm install --save-dev playwright-lighthouse playwright lighthouse*. Nakon izvršenja komande u direktoriju ćemo dobiti slijedeće:

- **node_modules** – *playwright* i *lighthouse* moduli koje ćemo koristimo za kreiranje i izvođenje testova

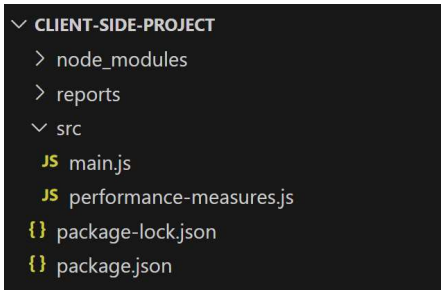
- **package** i **package-lock** – informacije o aplikacijama, modulima, paketima, verzijama i slično



Sada projekt je inicijaliziran i možemo kreirati testni kod što slijedi u sljedećoj sekciji.

3.4.3. Testiranje

Kod za pokretanje *Lighthouse* klijentskih testova puno je jednostavniji nego što je to bilo kod *Gatlinga*. To je upravo tako jer cijela logika testiranja vrlo je dobro odrađena od strane *Lighousea* kojeg mi kod kreiranja testova pozivamo kao spreman JS modul putem API-ja. Struktura projekta sa dodanim kodom izgleda kao na slijedećoj slici.



Dodali smo direktorij *src* u kojemu će se nalaziti kod aplikacije te direktorij *reports* u kojeg ćemo spremati izvještaje testiranja (slično kao kod *Gatinga*).

Sav kod za testiranje smješten je unutar JS datoteke *performance-measures.js* (isječak koda ispod). Ova skripta sastoji se od slijedećih dijelova:

- Započinje sa uključivanjem modula potrebnih za izvođenje testiranja.
- Zatim slijedi definicija varijable za URL *Gatling Demostore* web mjesta te glavna funkcija *testSite()* u kojoj pozivamo testove za željene web strance pojedinačno. Ovim putem testiramo web stranice koje smo prethodno testirali i u *Gatling* statičkim testovima.
- Svaka stranica se testira zasebno sekvencijalnim redom. Testiranje određene stranice započinje se pozivanje funkcije *testPage()* i prosljeđivanjem parametara putanje i naziva/oznake stranice.
- Funkcija *testpage()* započinje kreiranjem instance preglednika pomoću *playwrighta.* Zatim se učitava stranica sa definiranom putanjom. Samo testiranje/analiza web stranice se odvija pomoću komande *playAudit()*. U ovu komandu prosljeđujemo postavke:

konfiguracija desktop preglednika, učitana stranica, granice ocjene testova, port i postavke za generiranje izvještaja.

- Naposlijetku zatvaramo preglednik

```
import { playAudit } from 'playwright-lighthouse'
import playwright from 'playwright'
import lighthouseDesktopConfig from 'lighthouse/core/config/lr-desktop-config.js';

var siteUrl = "http://demostore.gatling.io"

export async function testSite(){
    await testPage("/", "HomePage");
    await testPage("/about-us", "AboutUsPage");
    await testPage("/category/all", "AllProducts");
    await testPage("/category/for-him", "ForHim");
    await testPage("/category/for-her", "ForHerPage");
    await testPage("/category/unisex", "UnisexPage");
    await testPage("/product/casual-black-blue", "ProductPage");
    await testPage("/product/perfect-pink", "ProductPage");
    await testPage("/login", "LoginPage");
}

async function testPage(pagePath="", pageName="") {
    const browser = await playwright['chromium'].launch({
        args: ['--remote-debugging-port=9222'],
    });
    const page = await browser.newPage();
    await page.goto(siteUrl + pagePath);

    await playAudit({
        config: lighthouseDesktopConfig,
        page: page,
        thresholds: {
            performance: 50
        },
        port: 9222,
        reports: {
            formats: {
              html: true,
            },
            name: pageName,
            directory: `reports`,
          }
    });

    await browser.close();
}
```

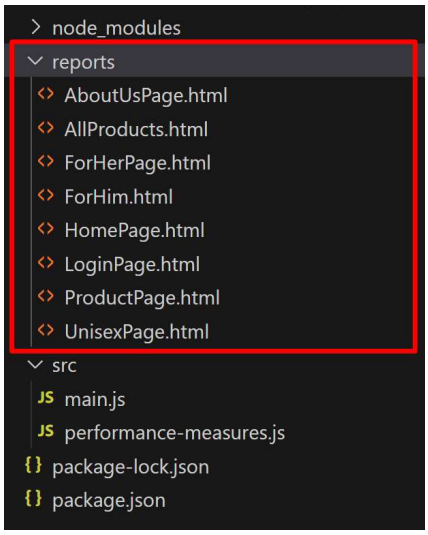
Duga JS datoteka pod nazivom *main.js* sadrži uključivanje testne JS datoteke i pozivanje funkcije *testSite()*. Pomoću ove datoteke pokrenut ćemo testove

```
import { testSite } from './performance-measures.js'

testSite();
```

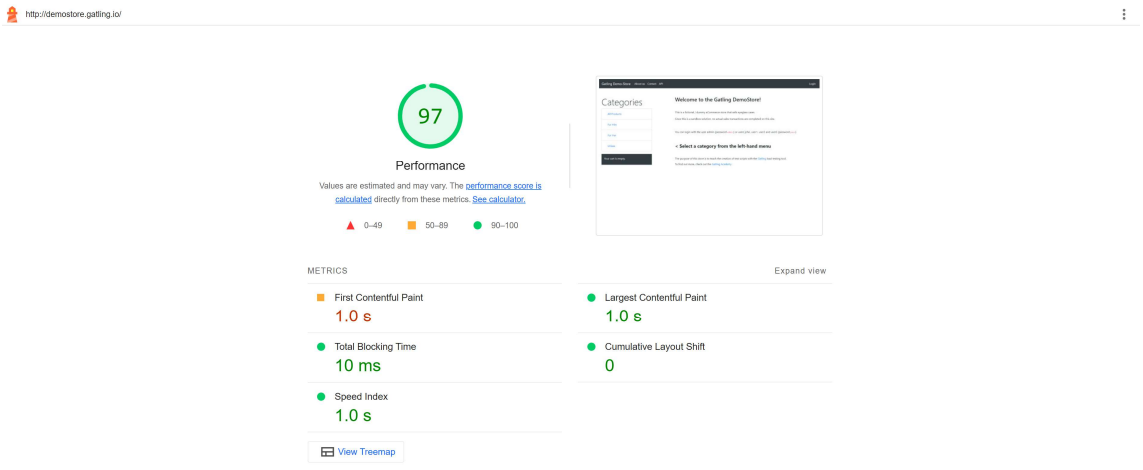
Testovi se pokreću pomoću komande *node .\src\main.js*. Tijekom izvršenja testova nakon svakog izvršenog testa dobit ćemo povratne informacije o ocjeni performanse i dali one prelaze zadani prag.

```
-------- playwright lighthouse audit reports --------

performance record is 97 and desired threshold was 50

-------- playwright lighthouse audit reports --------

performance record is 99 and desired threshold was 50
```

Detaljnije izvještaje možemo pronaći u *reports* direktoriju gdje se za svaki test generiraju HTML datoteke.



Otvaranjem pojedine HTML datoteke prikazat se izvještaj jednak onome kojeg smo prethodno analizirali pomoću Chrome preglednika.



Ovim pristupom Lighthouse testovi mogu se jednostavno i uz malo programskog koda automatizirati što je vrlo korisno kod velikih projekata gdje je broj stranica za testiranje velik. Isto tako dobivene rezultate moguće je usporediti sa definiranim pragovima zadovoljavajućih

performansi i ako dolazi do pada performansi na određenim stranicama napraviti automatsko upozorenje.

3.5. Analiza rezultata

U ovoj sekciji analizirat ćemo i objasniti rezultate dobivene tijekom testiranja prema modelu usporedbe definiranom u sekciji 3.2.

U prvom dijelu modela usporedit ćemo vremena odgovora koja su dobivena tijekom procesa testiranja kod oba pristupa. U Tablica 3 možemo vidjeti dobivena vremena odgovora od svih testiranih web stranica *Gatling Demostorea.* Prosječno na serverskoj strani vrijeme odgovora je 0.3 sekunde, dok je na klijentskoj strani to 1 sekunda. Na serverskoj strani također vidimo da najveće odstupanje ima stranica *About Us* čiji se zahtjev nešto dulje obrađuje na serveru. Na klijentskoj strani uzeti su rezultati mjere *Učitavanje najvećeg sadržaja* koja nam daje vrijeme kada se učitao i najzahtjevniji element pa je tada stranica u potpunosti učitana. Odstupanja kod klijenta nisu precizno izmjerena od strane *Lighthousea* ko što je to kod *Gatlinga* (milisekunde) pa ovdje možemo vidjeti da je najveće odstupanje na stranici *All Products* koja se učitava 1.2 sekunde.

Generalno možemo zaključiti da je prosječna razlika vremena odgovora kod oba pristupa 0.7 sekundi. Kod oba pristupa mjerenje vremena odgovora započinje kada se šalje zahtjev prema serveru. Kada odgovor dospije od strane servera tada je to vrijeme odgovora koje mjerimo na serverskoj strani. Kod klijenta ovo vrijeme završava kada se i posljednji element učita na stranici. Stoga, vrijeme odgovora na klijentskoj strani mogli bismo bolje nazvati vrijeme učitavanja sadržaja web stranice.

Vrijeme odgovora (sekunde)	**Serverska strana – Gatling**	**Klijentska strana – Lighthouse**
Home page	0.308	1
About Us	0.374	1
All Products	0.335	1.2
For Him	0.284	1
For Her	0.296	1
Unisex	0.313	1
Product	0.321	1.1
Login	0.302	0.8

Tablica 3, Rezultati vremena odgovora

U drugom dijelu modela kao rezultat unesene su ocjene za određene značajke i funkcionalnosti oba pristupa testiranju. Ove ocjene dobivene su temeljem usporedbe pristupa i izvođenjem testova. U nastavku slijedi tablica rezultata i obrazloženje.

Značajke i funkcionalnosti (Ocjena 0-5)	**Serverska strana – Gatling**	**Klijentska strana – Lighthouse**
Testiranje opterećenja	Odlično	Ne podržava
Testiranje klijentskih skripti	Minimalno	Odlično
Mogućnost integracije u CI/CD	Prosječno	Odlično
Testiranje API-a	Odlično	Ne podržava
Mogućnost automatizacije	Odlično	Zadovoljavajuće
Dokumentacija i podrška	Odlično	Zadovoljavajuće
Jednostavnost implementacije	Minimalno	Odlično
Mogućnost prilagodbe testova	Odlično	Minimalno
Podržani jezici programske implementacije	Odlično	Prosječno

Tablica 4, Ocjene značajki i funkcionalnosti

Za **testiranje opterećenja** serverska strana dobila je maksimalnu ocjenu jer glavni cilj ovog pristupa i jest generiranje raznih tipova opterećenja temeljem kojeg je moguće mjeriti vrijeme odgovora klijenta. Iako na klijentskoj strani također mjerimo vrijeme odgovora i vrijeme učitavanja elementa, to se ne postiže slanjem više zahtjeva nego se temelji na slanju jednog zahtjeva. Dakle, možemo zaključiti da kod klijentske strane se ne generira opterećenje, što znači da ta funkcionalnost nije podržana.

Testiranje klijentskih skripti odnosno klijentskog koda koji se izvršava tijekom učitavanja elementa u pregledniku, testirali smo sa *Lighthouseom.* Pomoću klijentskog testiranja dobivamo detaljne rezultate pa je ocjena maksimalna. Prije nego sam preglednik može početi sa izvršavanjem klijentskih skripti, mora ih preuzeti sa servera, i to će trajati za određeni vremenski interval. Preuzimanje se također može testirati i sa serverske strane što je korisno kako bi se kontrolirala veličina datoteka klijentskih skripti ili propusnost koju server pruža tijekom velikog broja zahtjeva. Pošto sa serverske strane možemo testirati samo vrijeme preuzimanja ocjena je *Minimalno*.

Pošto su Lighthouse klijentske skripte pisane u JavaScript jeziku njih je vrlo jednostavno **integrirati u CI/CD** procese i ocjena je stoga *Odlično*. Sa druge strane *Gatling* skripte se mogu pisati u Java ili Scala programskom jeziku. Ovi jezici kao što smo spomenuli pružaju bolju kontrolu nad pisanjem koda i posjeduju pogodne karakteristike za definiranje

testova. Zbog toga da bi se integrirali u CI/CD potrebno je nešto više napora, ali jest moguće pa je ocjena za serversku stranu *Zadovoljavajuće*.

Kod serverskog testiranja dinamičkim testovima proveli smo testiranje za akcije prijave i kupnje proizvoda sadržanih u košarici. Tom prilikom prije nego se učitala sljedeća web stranica poslali smo zahtjev na *Demostore* API kako bi izvršili prijavu/kupnju. To je bilo **testiranje API-a**. Serversko testiranje zadovoljava sve potrebe pa je ocjena *Odlično.* API kao odgovor ne vraća HTML kod web stranice, nego podatke u JSON formatu. Stoga, API-e nije moguće testirati pomoću *Lighthousea* zato jer klijentski alati mjere vrijeme renderiranja UI elementa koji su definirani u HTML kodu i skriptama. Dakle, testiranje API-a nije podržano.

Testovi kreirani u *Gatlingu* i općenito na serverskoj strani namijenjeni su za **automatizaciju**. U našem slučaju kod *Gatlinga* mogli smo vidjeti da u jednom testu možemo definirati više scenarija koji se tada izvađaju automatizirano prema određenoj konfiguraciji za generiranje opterećenja, tako da ovdje serverskoj strani dodijeljujemo maksimalnu ocjenu. *Lighthouse* je primarno alat u sklopu *Google Chrome* preglednika. Pomoću kombinacije *Node.js* modula moguće je automatizirati i *Lighthouse* testove, ali imamo nešto manju kontrolu što se tiče izvođenja testova, biranja mjera i slično. Stoga kod automatizacije *Lighthouse* dobiva ocjenu 4.

Dokumentacija za *Gatling* specifikaciju detaljna je i lako dostupna. Na njihovoj službenoj stranici (https://gatling.io/) mogu se pronaći i besplatni mini tečajevi koji vrlo detaljno opisuju mogućnosti ovog razvojnog okruženja. *Gatling* je ovdje dobio maksimalnu ocjenu. *Lighthouse* je *open-source* projekt te kod i dokumentacija dostupni su na javnom *GitHub* repozitoriju (https://github.com/GoogleChrome/lighthouse). Primjeri i video materijali mogu se pronaći pretraživanjem, no ne postoje službeni. Stoga, ocjena za dokumentaciju bit će *Zadovoljavajuće*.

Temeljem provedenih testiranja možemo zaključiti da je **implementacija** u velikoj mjeri jednostavnija za *Lighthouse* u Node.js-u. Kod *Gatlinga* implementacija testova je kompleksnija i potrebno je postaviti detaljne parametre i konfiguracije, ali zato pruža više mogućnosti. Stoga ocjena za *Lighthouse* je maksimalna dok je za *Gatling* minimalna.

Slijedeća karakteristika je usko povezana uz jednostavnost implementacije. Kao što smo naveli, *Gatling* testovi su kompleksniji za implementirati, ali je zato **mogućnost prilagodbe** znatno veća nego što je to kod *Lighthouse*. Stoga, ocjena za je maksimalna. Kod *Lighthousea* ocjena je niža jer kao što smo spomenuli opcije za definiranje pojedinosti kod testova su minimalne.

Gatling implementaciju testova podržava u čak 3 **programska jezika**: *Scala, Java* i *Kotlin.* Iako su sva tri jezika slična i bazirana na *Javai*, ipak nude više mogućnosti s obzirom na preferencije testera pa ovdje dajemo maksimalnu ocjenu. Kod *Lighthousea* trenutno implementacija automatizacije i prilagodbe testova je podržana u *JavaScriptu*, tj. razvojnom okviru *Node.js*, stoga je dodijeljena ocjena *Prosječno*.

3.6. Kombiniranje serverskog i klijentskog testiranja

Oba pristupa testiranju jednako su važna jer svako od njih u različitim aspektima daje određenu vrijednost testiranju performansi. Analiza rezultata usporedbe u sekciji 3.5 ukazuje na to da svaki pristup ima određene kvalitete kod pojedinih značajka. Testiranje web aplikacija, uz testiranje performansi, sastoji se i od ostalih vrsta testiranja koje dodaju vrijednost osiguranju kvalitete programskog proizvoda. Stoga, ovisno uvjetima koji vladaju na projektu razvoja web sustava, prioriteti mogu biti različiti.

Klijentsko testiranje provjerava performanse aplikacije na razini korisničkog sučelja te mjeri u kojem će se vremenu elementi pojaviti na ekranu. Ovi testovi identificiraju probleme na mikro razini jer mjere performanse za jednu instancu tj. jednog korisnika. Zbog toga je klijentsko testiranje limitirano u pogledu obujma i ne može dati dovoljno informacija o višestrukim komponentama od kojih se sastoji serverski sustav. Manje web aplikacije generiraju manje opterećenje te stoga, serversko testiranje opterećenja i nije nužno jer kritični problemi performansi lako su uočljivi i bez automatiziranih serverskih testova (npr. sporo učitavanje početne stranice). Zbog svoje jednostavnosti implementacije klijentski testovi pogodni su za manje projekte koji posjeduju i manje budžete, a također prisutni su i kod velikih projekata.

Serversko testiranje u odnosu na klijentsko pridonosi drukčijim aspektima testiranja kao što su:

- Daje informacije o performansama različitih komponenti koje čine sustav.
- Daje informacije o performansama kada se broj korisnika skalira što uzrokuje određeno opterećenje u odnosu na klijentsko koje je usmjereno samo na jednog korisnika
- Serverske ili *backend* komponente sustava mogu se testirati neovisno o klijentskom kodu, stoga na strani servera testiranje je moguće u ranijim fazama razvoja.

Prethodno navedene stavke igraju važnu ulogu kod velikih projekata gdje su performanse ključne i testiranje se izvodi u više faza razvoja. Serversko testiranje zbog svoje složene implementacije iziskuje više angažmana što uzrokuje veće troškove implementacije testova.

Projekti s velikim budžetima često osiguraju dovoljna sredstva za kompleksna testiranja jer je testiranje neizbježno, a pogotovo testiranje performansi. U pravilu, mali projekti izbjegavaju serversko testiranje zbog velikih troškova, osim ako projekt nije uvelike ovisan o mikroservisima i API-ima.

4. Zaključak

U razvoju programskih rješenja testiranje performansi u današnje vrijeme neophodno je i jednako važno kao i razvoj sustava, osobito kada su u pitanju projekti velikih razmjera. Testiranjem performansi osiguravamo da programsko rješenje bude u skladu sa standardima koji su definirani sukladno očekivanjima korisnika. To je bitno jer financijski i opći uspjeh nekog projekta ovisi naravno o zadovoljstvu korisnika programskim rješenjem.

Testiranje se može odvijati na serverskoj i klijentskoj strani iz čega proizlaze dva pristupa. Oba pristupa su jednako važna, ali ipak pridonose različitim aspektima testiranja. Klijentski pristup nam daje informacije koliko je u kojem vremenu korisnik dobije sadržaj na svojem ekranu koji ovisi o vremenu dohvata podatka sa servera, izvršavanja klijentskih skripti, a ovisi i o samom uređaju na kojem se izvodi. Serverski pristup izostavlja klijentske aktivnosti, ali nam daje mogućnosti testiranja različitih serverskih komponenti sa različitim intenzitetom opterećenja što nam dodatno daje i sliku o okruženju sustava. Također, serverska testiranja moguća su u ranijim fazama razvoj programskog rješenja jer ne ovise o klijentskom kodu. (van der Hoeven, n.d.)

U praktičnom dijelu prikazali smo kreiranje i izvođenje testova za oba pristupa pomoću *Gatlinga* i *Lighthousea*. Kreiranje serverskih testova puno je kompleksnije jer je sama problematika testiranja složenija, a također i testovi se kreiraju u potpunosti manualno. *Lighthouse* kao klijentski okvir za testiranje sadrži već gotovu implementaciju testiranja prema najboljim principima preporučenima od *Googlea* te je programski potrebno samo definirati automatizaciju testova. (van der Hoeven, n.d.)

Oba pristupa daju određenu vrijednost projektu razvoja programskog rješenja, ali je potrebno razmotriti kada se implementacija pojedinog pristupa isplati te kako ih kombinirati. Projekti sa manjim budžetima u svoj će proces uključiti klijentske i eventualno neke jednostavne serverske testove. Na većim projektima implementiraju se klijentski, ali i složeni serverski testovi performansi jer takvi projekti sastoje se od mnogo razvojnih ciklusa te je potrebno osigurati kvalitetu softvera u ranim fazama i održati ju kada kompleksnost sustava postaje veća.

Popis literature

van der Hoeven, N. (n.d.). *Frontend vs. backend: How to plan your performance testing strategy*. Preuzeto 8. 2 2024 iz https://grafana.com/blog/2023/04/03/frontend-vs.-backend-how-to-plan-your-performance-testing-strategy/

Data Dome. (n.d.). Preuzeto February 2024 iz https://datadome.co/learning-center/how-to-reduce-server-response-time/#:~:text=If%20your%20server%20response%20time,1%20second%20is%20too%20slow.

Galeza, A. (2022). *9 Recommended Performance Testing Tools in 2022*. Preuzeto 5. May 2023 iz https://www.netguru.com/blog/performance-testing-tools

Github Grafana. (n.d.). Preuzeto 8. 9 2023 iz https://github.com/grafana/k6

Github WebPageTest. (n.d.). Preuzeto 8. 9 2023 iz https://github.com/catchpoint/WebPageTest.docs

Justia. (n.d.). Preuzeto 8. 9 2023 iz https://onward.justia.com/website-metrics-with-google-lighthouse

Kartaca. (n.d.). Preuzeto 8. 9 2023 iz https://kartaca.com/en/the-importance-of-testing-and-apache-jmeter/

Kubernetes vs. Virtual Machines, Explained. (n.d.). Preuzeto 8. 2 2024 iz https://portworx.com/blog/kubernetes-vs-virtual-machines/

Lighthouse performance scoring. (n.d.). Preuzeto 10. 9 2023 iz https://developer.chrome.com/docs/lighthouse/performance/performance-scoring/

Meaning of performance. (n.d.). Preuzeto 6. September 2023 iz Cambridge Dictionary: https://dictionary.cambridge.org/dictionary/english/performance

Mohan, G. (June 2022). *Full Stack Testing.* O'Reilly Media, Inc.

Molyneaux, I. (2009). *The Art of Application Performance Testing.* O'Reilly Media, Inc.

Shivakumar, S. K. (2020). *Modern Web Performance Optimization: Methods, Tools, and Patterns to Speed Up Digital Platforms.* Apress.

Sipriano, R. (2021). *What actually is software performance?* Preuzeto 6. September 2023 iz https://www.linkedin.com/pulse/what-actually-software-performance-rodrigo-sipriano

Sitespeed. (n.d.). Preuzeto 8. 9 2023 iz https://www.sitespeed.io

TestMatick. (n.d.). Preuzeto 8. 9 2023 iz https://testmatick.com/best-load-testing-tools/gatling-logo/

Yorkston, K. (2021). *Performance Testing: An ISTQB Certified Tester Foundation Level Specialist Certification Review.* Apress.

Popis slika

Popis tablica