# Napredni razvoj programske potpore za web

**- predavanja -**

**2021./2022.**

## Skalabilnost web-aplikacija

# Creative Commons

- slobodno smijete:
  - **dijeliti** — umnožavati, distribuirati i javnosti priopćavati djelo
  - **prerađivati** djelo
- pod sljedećim uvjetima:
  - **imenovanje:** morate priznati i označiti autorstvo djela na način kako je specificirao autor ili davatelj licence (ali ne način koji bi sugerirao da Vi ili Vaše korištenje njegova djela imate njegovu izravnu podršku).
  - **nekomercijalno:** ovo djelo ne smijete koristiti u komercijalne svrhe.
  - **dijeli pod istim uvjetima:** ako ovo djelo izmijenite, preoblikujete ili stvarate koristeći ga, preradu možete distribuirati samo pod licencom koja je ista ili slična ovoj.

*U slučaju daljnjeg korištenja ili distribuiranja morate drugima jasno dati do znanja licencne uvjete ovog djela.*
*Od svakog od gornjih uvjeta moguće je odstupiti, ako dobijete dopuštenje nositelja autorskog prava.*
*Ništa u ovoj licenci ne narušava ili ograničava autorova moralna prava.* *Tekst licence preuzet je s http://creativecommons.org/*

# Sadržaj

1. Skalabilnost web aplikacija – poslužitelj
2. Skalabilnost web aplikacija – klijent
    1. Apache JMeter
        - ispitivanje performansi web poslužitelja
    2. Ostale aplikacije
        - potpora razvoju web aplikacija

# Uvod (1)

- **Skalabilnost je sposobnost prilagođavanja kapaciteta sustava učinkovitom ispunjavanju zahtjeva**
  - Skalabilnost opisuje sposobnost rukovanja većim brojem korisnika, klijenata, podataka, transakcija ili zahtjeva bez utjecaja na korisničko iskustvo
  - Učinkovito (*efficient*) ili ekonomično (*cost-efficient, cost-effective*) ispunjavanje zahtjeva
- Kapacitet:
  - Sposobnosti programske podrške i sklopovlja namijenjene izvršenju funkcionalnosti sustava; raspoloživost resursa; broj i vrsta računala, broj i vrsta procesora, broj i kapacitet pohrane podataka tvrdih diskova, propusnost i brzina mreže, predmemorija tvrdih diskova, predmemorija baze podataka, …
- Neke definicije skalabilnosti iz literature:
  - *Capacity of the system to handle increasing amount work without affecting existing system.*
  - *Scalability is a characteristic of a system, model or function that describes its capability to cope and perform under an increased or expanding workload. A system that scales well will be able to maintain or even increase its level of performance or efficiency when tested by larger operational demands.*
  - *The measure of a system's ability to increase or decrease in performance and cost in response to changes in application and system processing demands.*
  - *Property of software and hardware systems that can improve functionality, performance, and speed by adding or removing more processors, memory, and other resources.1*

*https://www.igi-global.com/dictionary*

# Uvod (2) – performanse

- Skalabilnost je povezana s **performansama**
- Performanse određuju koliko je vremena potrebno za obradu zahtjeva ili za obavljanje određenog zadatka, dok skalabilnost mjeri koliko sustav može rasti (ili se smanjivati)

- Skalabilnost bi trebala biti što više jednostavna, jeftina (utrošak čovjek-sati) i moći se brzo provesti

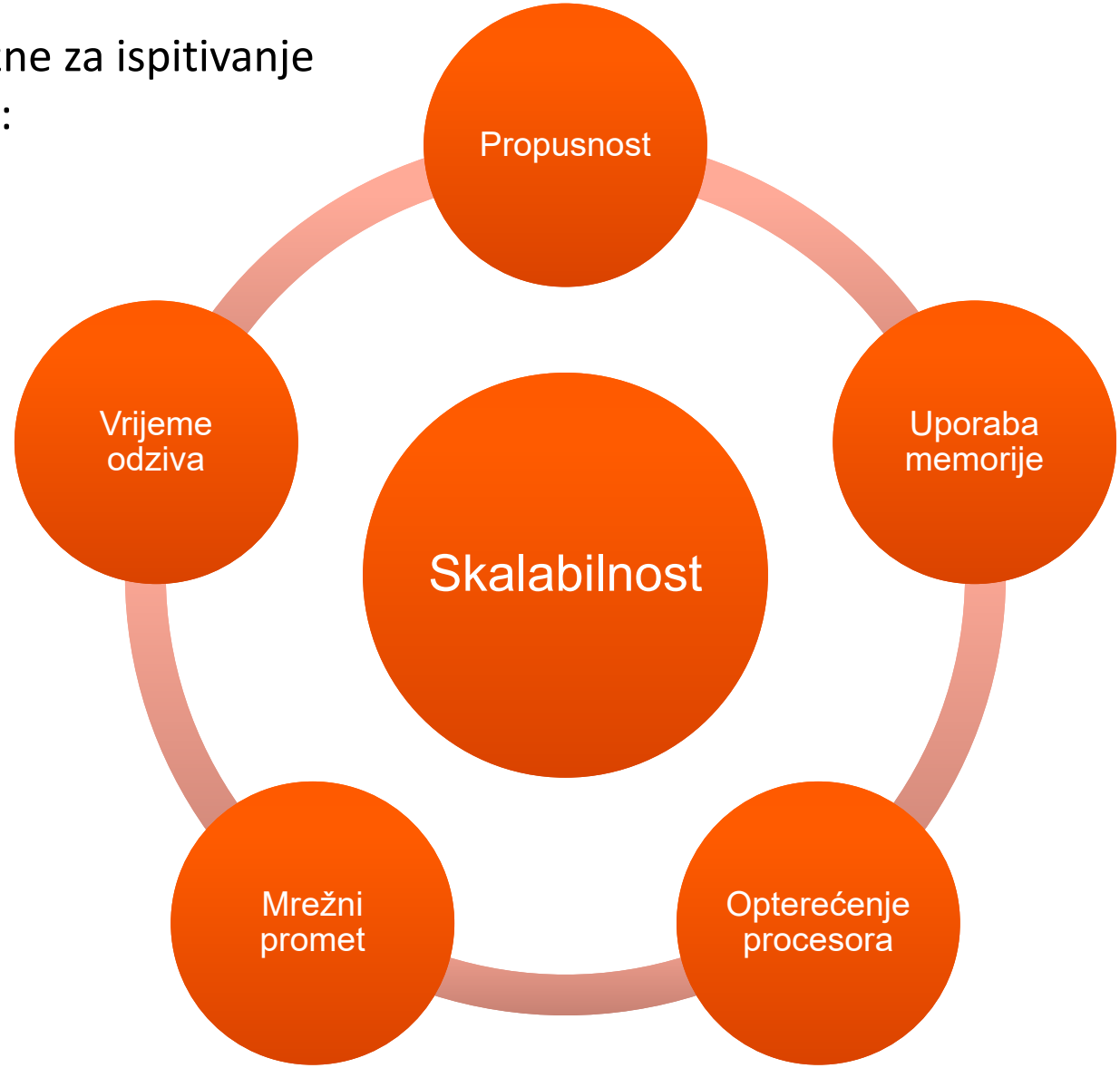
# Uvod (3)

- Problemi koje bi skalabilnost trebala rješavati su:
  - manipuliranje većim količinama podataka
  - posluživanje većeg broja korisnika istovremeno
  - omogućavanje većeg broja interakcija između sustava i klijenata

- Skalabilnost softverskog proizvoda može biti ograničena brojem inženjera koji mogu raditi na sustavu.
- Kako sustav raste, važno je omogućiti i organizacijsku skalabilnost kako bi se dovoljno brzo mogle napraviti promjene i prilagodbe. Ako je sustav vrlo čvrsto povezan, rad jednog inženjerskog tima sa više od npr. 8 do 15 ljudi postaje neučinkovit, s obzirom da svi rade na istom kodu, a opterećenje komunikacije raste eksponencijalno s veličinom tima.

# Uvod (4)

Značajke važne za ispitivanje skalabilnosti:



# Skalabilna arhitektura aplikacija

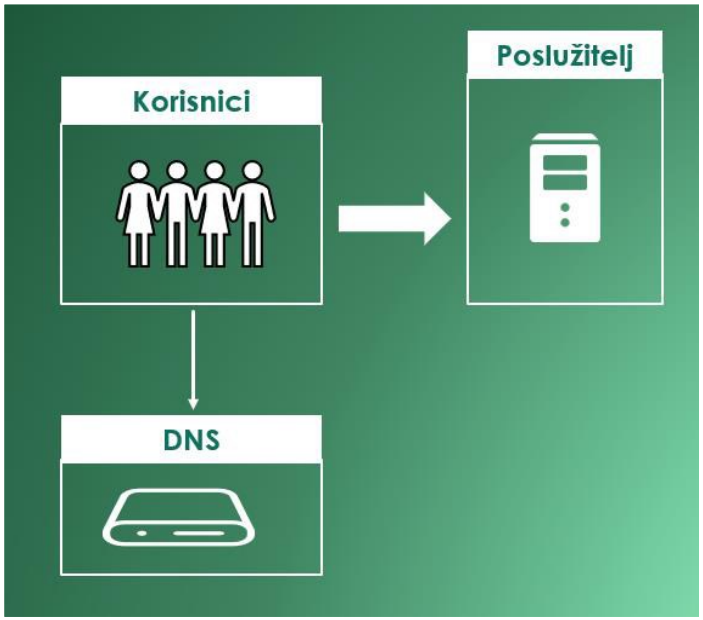
- Konfiguracija s jednim poslužiteljem
- Vertikalno skaliranje
- Horizontalno skaliranje
- Izdvajanje servisa
- CDN (*Content Delivery Network*)
- Globalna skalabilnost

# Konfiguracija s jednim poslužiteljem (1)

- Konfiguracija s jednim poslužiteljem je najjednostavnija konfiguracija
  - Često s ovom konfiguracijom započinju mali projekti
- Opis konfiguracije s jednim poslužiteljem:
  - Cijela aplikacija radi na jednom računalu
  - Sav promet za svaki korisnički zahtjev obrađuje isti poslužitelj za koji se obično ne koristi vlastiti poslužitelj nego DNS (*Domain Name System*) kao plaćena usluga koju pruža hosting tvrtka
  - U ovom scenariju korisnici se spajaju s DNS-om kako bi dobili adresu internetskog protokola (IP) poslužitelja na kojem se nalazi web-lokacija
  - Kada se dobije IP adresa, šalju se HTTP zahtjevi izravno na web-poslužitelj
  - Budući da se sve radi na jednom poslužitelju, potrebno je izvršiti sve dužnosti potrebne za pokretanje aplikacije kao što je upravljanje bazom podataka, posluživanje slika i dinamičkih sadržaja...
- **Konfiguracija s jednim poslužiteljem dovoljna je za web-stranice s niskim prometom**

# Konfiguracija s jednim poslužiteljem (2)

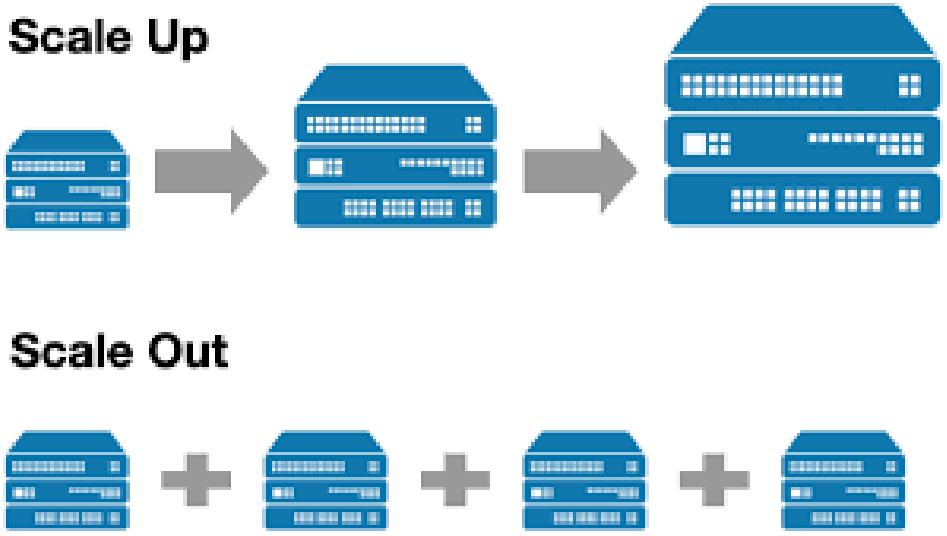
- Ova vrsta konfiguracije nije skalabilna i javljaju se problemi ako:
  - broj korisnika raste, čime se povećava promet, posluživanje svakog korisnika troši više resursa, uključujući memoriju, CPU vrijeme i I/O zahtjeve
  - baza podataka raste dodavanjem podataka, pa se izvođenje upita počinje usporavati
  - sustav se proširuje dodavanjem novih funkcionalnosti, zbog čega interakcije korisnika zahtijevaju više sistemskih resursa



# *Scale up / Scale out* (1)

- **Vertikalno skaliranje = *scale up / down***
  - povećanje performansi postojećih poslužitelja bilo u CPU-u, memoriji (primarnoj, RAM), brzini ili veličini diska (sekundarna memorija) poslužitelja, npr. dodavanje novih procesora na isti poslužitelj
  - Pogodno za male web aplikacije gdje se kapacitet može održavati povećanjem opterećenja samo povećanjem kapaciteta i veličine već korištenih resursa

- **Horizontalno skaliranje = scale out / in**
  - dodavanje više fizičkih strojeva ili resursa → smanjenje opterećenja na svakom stroju/resursu

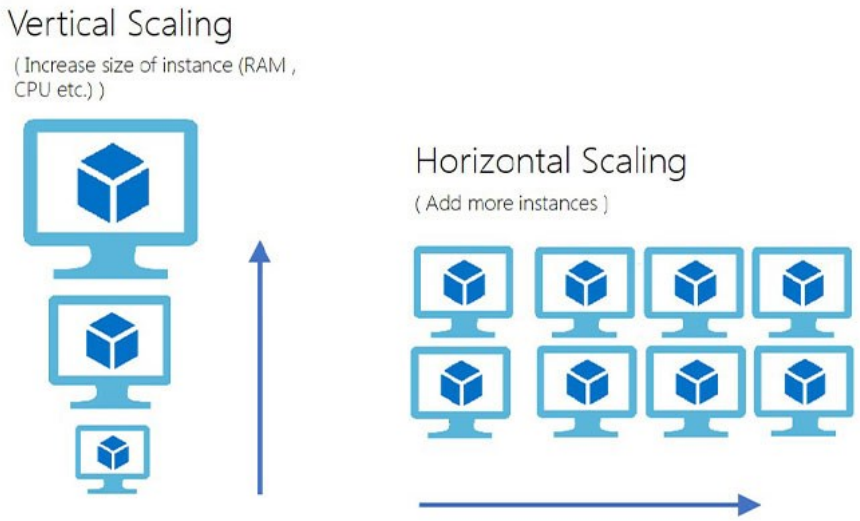



*https://blog.router-switch.com/2021/03/what-is-the-difference-between-scale-up-and-scale-out/*

# *Scale up / Scale out* (2)

- Sustav mora biti sposoban rasti kako bi adekvatno mogao biti korišten od više korisnika, obradio više podataka i upravljao s više transakcija ili zahtjeva klijenata bez degradacija u performansama (specifikacija sustava) i korisničkog iskustva.

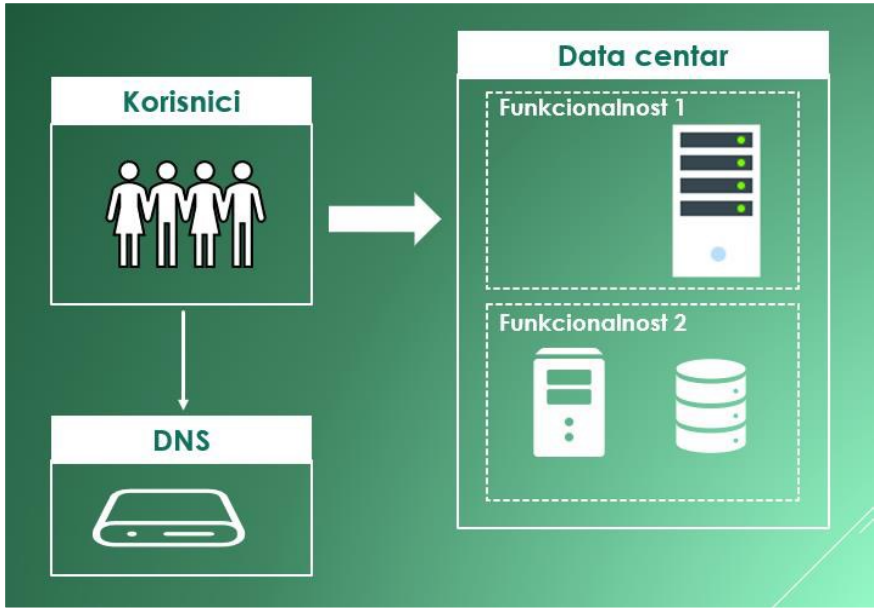
- Kvalitetan skalabilan sustav trebao bi omogućiti i smanjenje kapaciteta.
- Smanjenje je često manje važno od povećanja, ali potrebno je uštedjeti troškove i ne koristiti više od onoga što je potrebno.




*https://www.webairy.com/horizontal-and-vertical-scaling/*

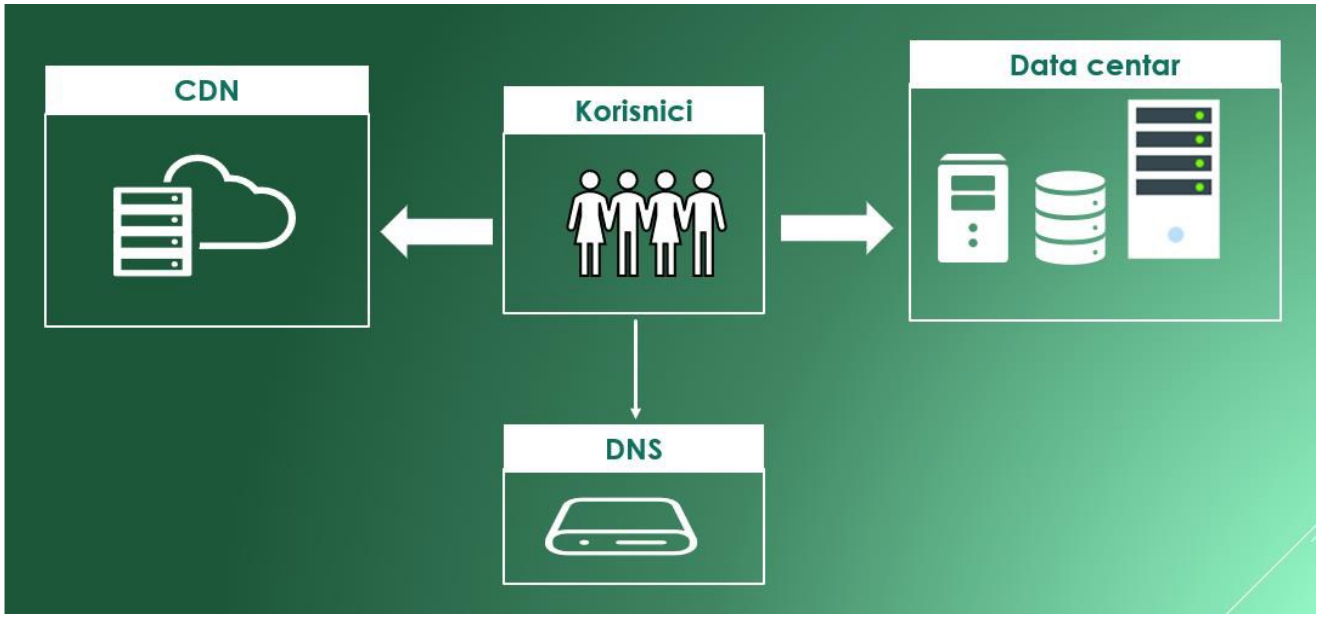
# Izdvajanje servisa

- Svodi se na izdvajanje različitih dijelova sustava na odvojene poslužitelje instaliranjem pojedine vrste usluge na zasebni fizički stroj
  - Usluga je bilo koji dio aplikacije poput web-poslužitelja, baze podataka, …
- Jednostavno, ali i ograničeno rješenje za ostvarivanje skalabilnosti
  - Kad se svaka vrsta usluge jednom razmjesti na zasebni poslužitelj više nema prostora za daljnji rast



# CDN

- Svodi se na prebacivanje dijela prometa na uslugu isporuke sadržaja treće strane
  - *Content Delivery Network*, CDN
- Prednosti:
  - Segmentacija publike na temelju korisničke analize
  - Ušteda troškova smanjenjem propusnosti
  - Omogućiti naprednu sigurnost web sjedišta



# Načela skalabilnosti

- Načela ili principi skalabilnosti su:
  1. Jednostavnost (*Simplicity*)
  2. Slaba povezanost (*Loose Coupling*)
  3. Izbjegavanje ponavljanja (*Don't Repeat Yourself*, DRY)
  4. Kodiranje temeljeno na ugovoru (*Coding to Contract*)
  5. Izrada dijagrama (modeliranje, *modelling*)
  6. Načelo jedinstvene odgovornosti (*Single-Responsibility Principle*)
  7. Otvoreno-zatvoreno načelo (*Open-Closed Principle*)
  8. Ubacivanje ovisnosti (*Dependency Injection*)
  9. Inverzija kontrole (*Inversion of Control,* IOC)
  10. Dodavanje klonova
  11. Funkcionalno razdvajanje (*Functional Partitioning*)
  12. Particioniranje podataka (*Data Partitioning*)
  13. Visoka dostupnost

# 1. Načelo jednostavnosti

- Temelji se na načelima objektno orijentiranog programiranja: skrivanje složenosti i izgradnja apstrakcija
- Postići lokalnu jednostavnost strukture kôda
  - Postići da se brzo može shvatiti koja je svrha kôda i kako radi, bez poznavanja svih detalja kako drugi udaljeni dijelovi sustava rade
  - Moći vidjeti samo aplikacije najviše razine sustava i prepoznati njihove zadaće
- Izbjeći „**overengineering**“
  - Ako se pokuša predvidjeti svaki mogući scenarij i svaki rubni uvjet, gubi se fokus na najčešće scenarije.
  - Rješavanjem svakog zamislivog problema doći će se do rješenja koje je mnogo složenije nego što je stvarno potrebno.
- Jednostavnost se može postići i **korištenjem TTD (test-driven development)** metodologije
  - Prvo definirati testove, a zatim implementirati funkcionalnosti
  - Izbjegava se pisanje nepotrebnog kôda

# 2. Načelo slabe povezanosti

- Veze između različitih dijelova sustava potrebno je održavati na što je moguće nižoj razini
  - Različite komponente znaju jedna o drugoj samo onoliko koliko je neophodno potrebno (***Loose Coupling***)
  - U idealnom slučaju komponente su potpuno odvojene, tj. rade potpuno neovisno jedna o drugoj
- Razdvajanje se ostvaruje na najvišoj razini
- Svaka pojedina aplikacija unutar sustava treba pokrivati neku užu funkcionalnost
- Prilikom pisanja koda treba dijeliti samo onoliko informacija i funkcionalnosti koliko je potrebno da bi se zadovoljili zahtjevi na sustav
- U objektno-orijentiranim jezici potrebno je:
  - izbjegavati javne (*public* metode)
  - izbjegavati povratne veze među klasama ili dijelovima sustava

# 3. Načelo izbjegavanja ponavljanja

- Nepotrebno je poduzimati iste aktivnosti više puta
  - *Don't Repeat Yourself*, DRY
  - Dupliciranje kôda, neučinkovit kôd, neučinkoviti procesi, neautomatizirani procesi, …
- Najčešći problem je **copy/paste programiranje**
  - Za slične dijelove programa kopira se veći komad kôda i samo malo izmijeni ono što je potrebno
  - Nastaje glomazni i nepregledan kôd bez modularnosti
  - Ako je potrebno provesti neku promjenu?

# 4. Načelo kodiranja temeljenog na ugovoru

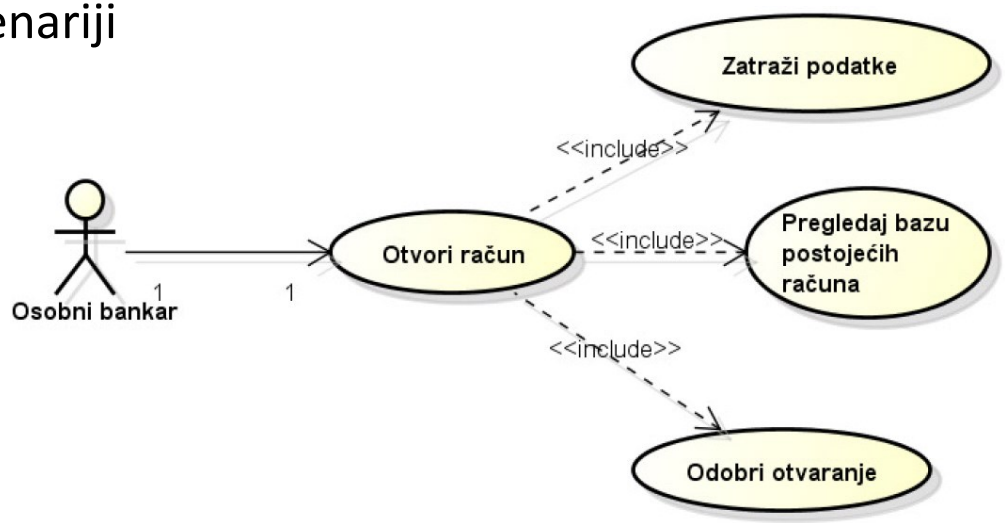
- Odnosi se na razdvajanje klijenata od pružatelja usluga
  - kodiranje temeljeno na sučelju, ***Coding to Contract***
- Ugovor je skup funkcionalnosti koje poslužitelj treba isporučiti
  - Klijentima je dostupna informacija kako se određeni dio softvera može koristiti i koje su funkcionalnosti dostupne
  - Ne moraju znati kako su te funkcionalnosti implementirane
- U razini objektno-orijentiranog kôda ugovor je identičan javnom sučelju klase (*public interface*)
- Na višim razinama ugovor je skup svih javno dostupnih klasa / metoda / sučelja

# 5. Načelo izrade dijagrama

- Namjena dijagrama je dokumentiranje sustava na jednoznačan način
  - Iznositi i razmjenjivati znanje (*mutual/shared understanding*) između razvojnih inženjera te između inženjera i korisnika sustava
  - Pomažu u shvaćanju vlastitog dizajna sustava
- Unified Modelling Language (UML)
- Tri vrste dijagrama su posebno korisni u ovom načelu:
  - Dijagrami obrazaca uporabe (*use-case diagrams*)
  - Dijagrami razreda (*class diagrams*)
  - Dijagrami komponenti (*component diagrams*)

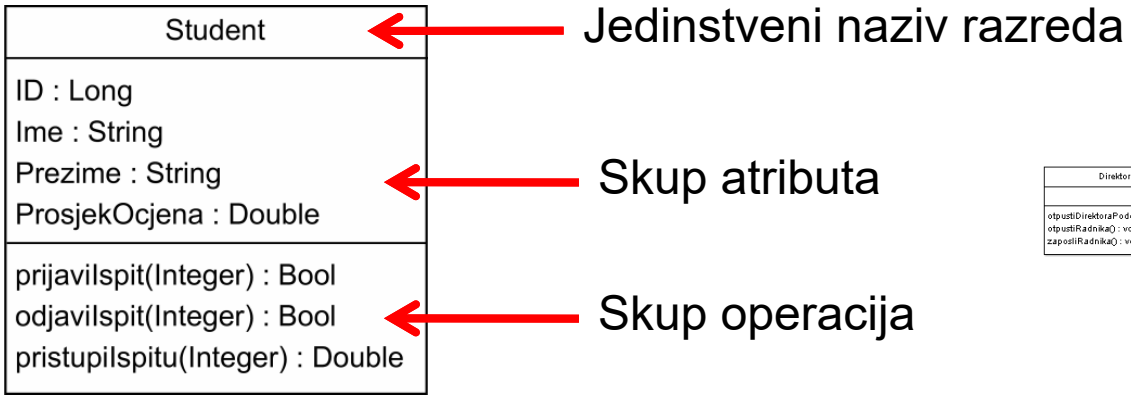
# 5. Načelo izrade dijagrama – Obrasci uporabe

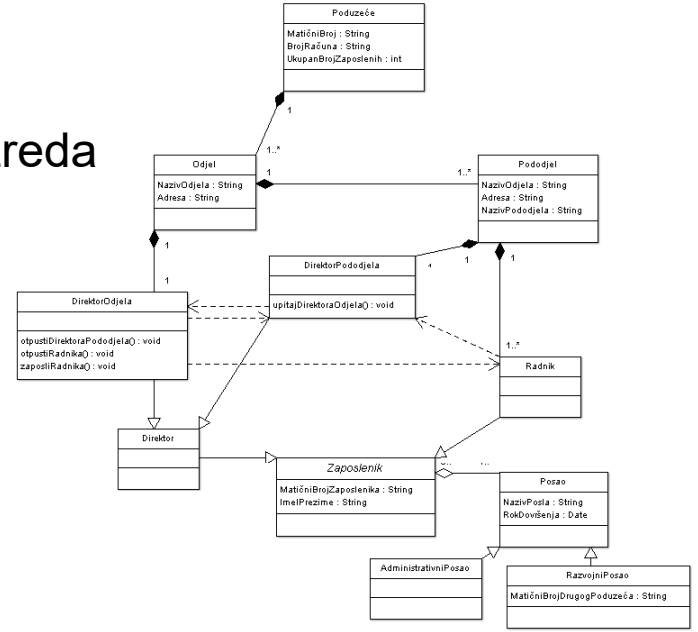
- UML obrasci uporabe koriste se za modeliranje zahtjeva korisnika i scenarija ispitivanja sustava
- Ne koristi se nijedan implementacijsko-specifičan jezik i uvijek se izrađuje na određenoj razini detalja, razine se ne miješaju na istom dijagramu
- Razina apstrakcije je najčešće visoka, konceptualna
- Služe za opis funkcionalnih zahtjeva projekta i to:
  - Svih aktora, bili oni aktivni aktori (inicijatori) ili pasivni aktori (sudionici)
  - Svih načina rada sustava - scenariji
- Prema potrebi, crta se više dijagrama da se obuhvate svi dijelovi sustava



# 5. Načelo izrade dijagrama – Dijagrami razreda

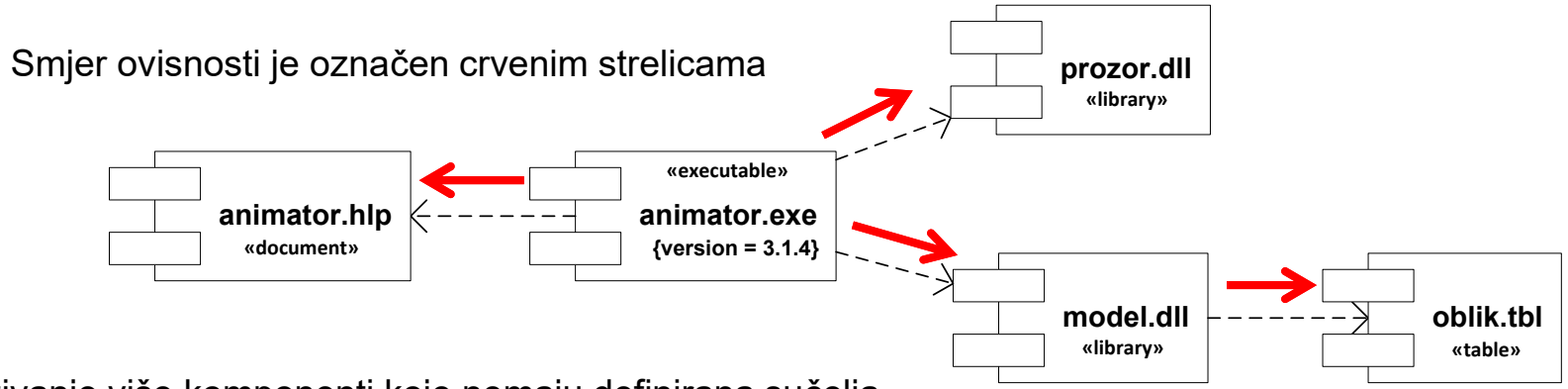
- Dijagram razreda opisuje vrste objekata unutar nekog sustava i njihove međusobne odnose.
  - *Class* dijagram opisuje razrede (klase) i njihove međusobne veze
- Strukturni UML dijagram, opisuje statične odnose.
- Prikazuju razrede, atribute i operacije razreda, njihova svojstva i ograničenja, sučelja, pridruživanja, vlastite tipove podataka, enumeracije, pakete i komentare



# 5. Načelo izrade dijagrama – Dijagrami komponenti

- Dijagram komponenti (*component diagram*) prikazuje komponente (strukturne cjeline) sustava i njihove međusobne odnose
  - Strukturni UML dijagram, opisuje statične odnose s fizičkog aspekta implementacije.
  - Komponenta je zasebna cjelina programske potpore s vlastitim sučeljem.
  - Komponentni dijagrami pomažu u modeliranju fizičkih cjelina sustava kao što su izvršne datoteke, programske biblioteke, tablice, datoteke i svi drugi dokumenti.
  - Zajedno s dijagramima razmještaja nazivaju se fizički dijagrami te se često prikazuju zajedno u jednom UML dijagramu komponenti i razmještaja.




Pridruživanje više komponenti koje nemaju definirana sučelja

# 6. Načelo jedinstvene odgovornosti

- Klasa bi trebala imati samo jednu odgovornost
  - *Single-Responsibility Principle*
  - Način za smanjenje složenosti koda

- Načini pridržavanja načela jedinstvene odgovornosti:
  - Duljina klase je maksimalno četiri zaslona koda
  - Klasa ne ovisi o više od pet drugih sučelja / klasa
  - Klasa ima specifičan cilj / svrhu
  - Odgovornost klase je moguće sažeti u jednoj rečenici, u komentaru iznad klase

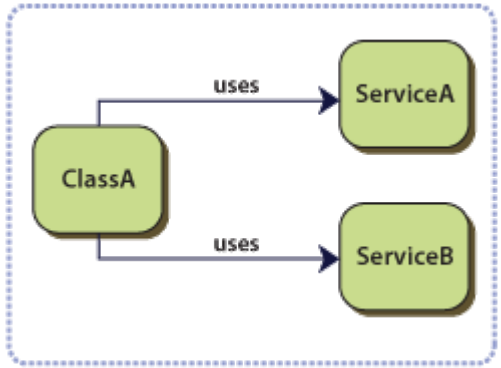
# 7. Otvoreno-zatvoreno načelo

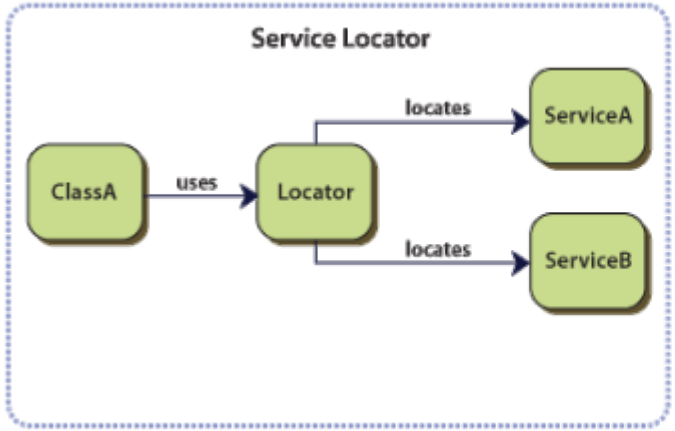
- Odnosi se na stvaranje koda koji se ne mora mijenjati kada se promijene zahtjevi ili kada se pojave novi obrasci uporabe
  - *Open-Closed Principle*
  - Kôd je otvoren za proširenje i zatvoren za izmjenu
  - Kreira se s namjerom da ga se u budućnosti proširi, ali bez potrebe za njegovom promjenom

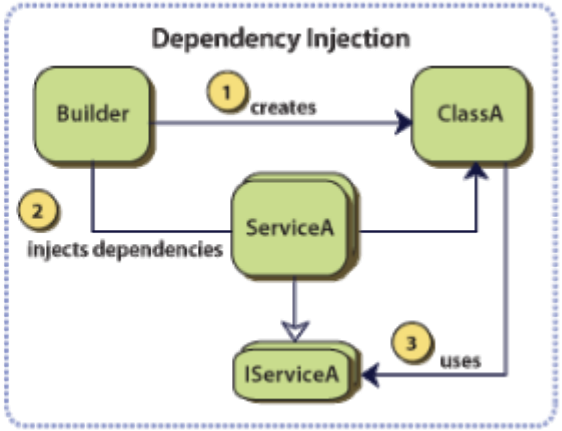
- Glavni cilj ovog načela je povećati fleksibilnost softvera i učiniti buduće promjene jeftinijima.

# 8. Načelo ubacivanja ovisnosti (1)

- Načelo ubacivanja ovisnosti je jednostavna tehnika koja smanjuje povezivanje klasa, modula i komponenti sustava
  - *Dependency Injection*
  - Promiče otvoreno-zatvoreno načelo
- U objektno-orijentiranim jezicima daje reference na objekte o kojima klasa ovisi, umjesto da dopusti samoj klasi da prikupi ovisnosti
  - Omogućuje klasama da ne znaju kako su njihove zavisnosti složene i odakle dolaze.




*https://www.c-sharpcorner.com/UploadFile/dacca2/service-locator-design-pattern/*

Oblikovni obrasci *Dependency Injection* i *Service Locator*

# 8. Načelo ubacivanja ovisnosti (2)

- Tri pristupa ubacivanja ovisnosti:
  - Ubacivanje konstruktora
    - *Constructor Injection*
    - Stvoriti instancu ovisnosti i proslijediti je kao argument konstruktoru zavisne klase
  - Ubacivanje metode
    - *Method Injection*
    - kreirati instancu ovisnosti i proslijediti je specifičnoj metodi zavisne klase
  - *Ubacivanje svojstva*
    - *Property Injection*
    - dodijeliti instancu ovisnosti određenom svojstvu zavisne klase

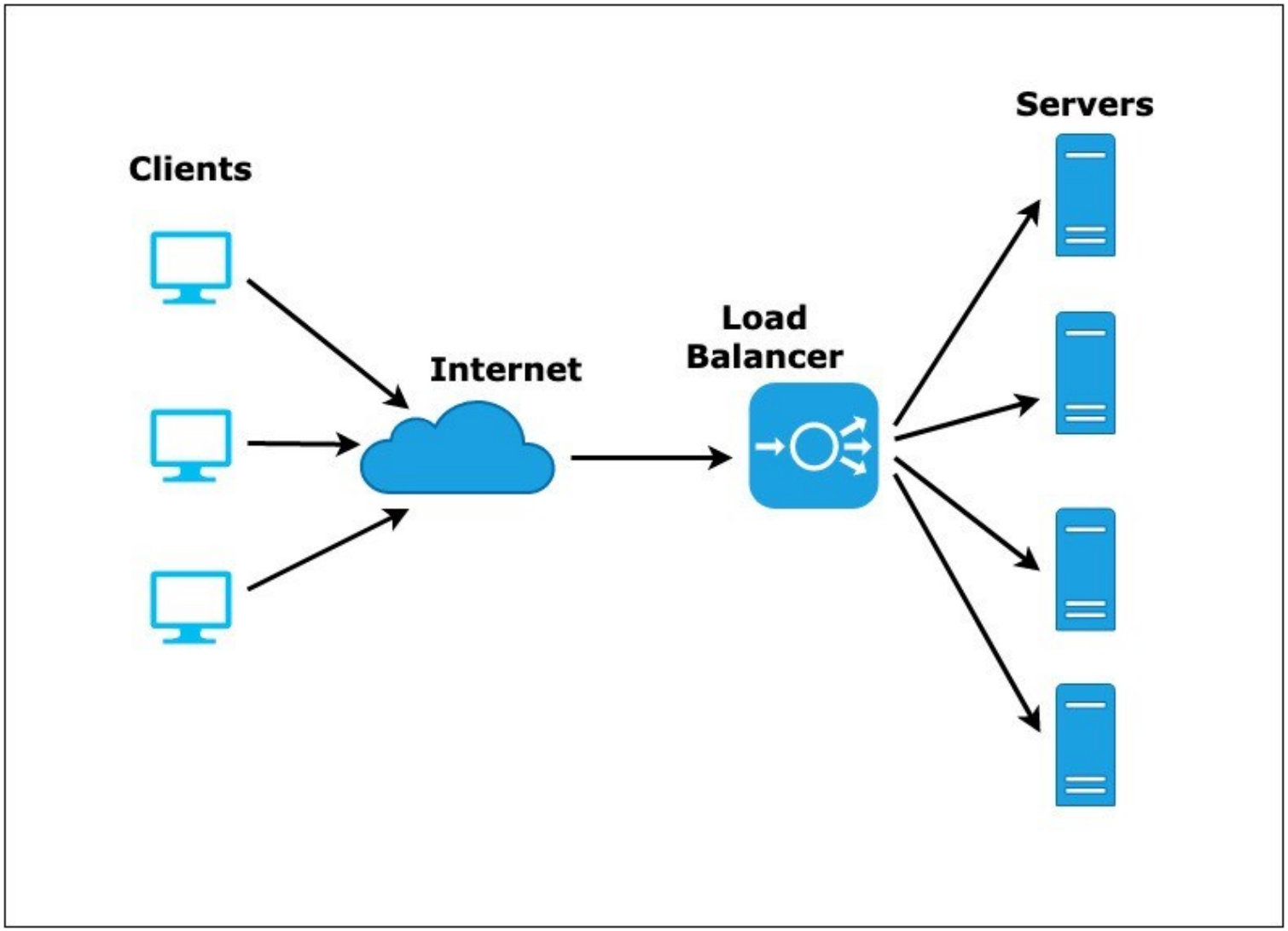
# 9. Načelo inverzije kontrole

- Načelo uklanjanja odgovornosti iz klase kako bi bila jednostavnija i manje povezana s ostatkom sustava
  - *Inversion of Control*, IOC
  - U suprotnosti od načela ubacivanja ovisnosti koje je ograničeno na stvaranje objekata i sastavljanje njihovih ovisnosti
- Korištenjem ovog načela nije potrebno znati tko će stvarati i koristiti objekte iz klase, kako ili kada
- Nije potrebno imati kontrolu nad stvaranjem instanca objekata i pozivanjem metoda
  - Stvaraju se samo dodaci i proširenja radnog okvira
- U objektno-orijentiranom programiranju oblikovni obrasci *Dependency Injection* i *Service Locator* su implementacije ovog načela

# 10. Načelo dodavanja klonova (1)

- Načelo dodavanja klonova je najlakša i najčešća strategija skaliranja
- Klonovi su kopije komponente ili poslužitelja
  - Moraju biti međusobno zamjenjivi i jednako kvalificirani za izvršavanje zahtjeva
  - Zahtjev se može poslati bilo kojem slučajno odabranom klonu
- Potrebno je koristiti alat za balansiranje opterećenja (***load balancer***)
  - Sadržava listu svih poslužitelja i raspodijeljuje zahtjeve na poslužitelje
  - Najčešći algoritmi: 1) slučajnim odabirom, 2) slijedno (*round robin*), 3) ovisno o opterećenju (*least connections*) i 4) hash (poslužitelj se odabire temeljem IP adrese zahtjeva – isti poslužitelj obrađuje sve zahtjeve jednog korisnika)
  - Skaliranje dodavanjem klonova najbolje funkcionira za usluge bez stanja, jer ne treba brinuti o sinkronizaciji
  - Pogodan način skaliranja za baze podataka kroz primjenu replikacije (zasebna i složena tema)

# 10. Načelo dodavanja klonova (2)




*https://codeburst.io/load-balancers-an-analogy-cc64d9430db0*

# 11. Načelo funkcionalnog razdvajanja

- Glavna zamisao je tražiti dijelove sustava usmjerene na određenu funkcionalnost i stvoriti od njih neovisne podsustave.
  - *Functional Partitioning*
- U kontekstu infrastrukture svodi se na izoliranje različitih uloga poslužitelja: web-poslužitelji, spremišta podataka, *caching* poslužitelji, *load balanceri*
  - Svaka od tih komponenti može biti ugrađena u glavnu aplikaciju, ali je bolje rješenje izolirati različite funkcije u nezavisne podsustave.
- U općem kontekstu svodi se na podjelu sustava na samostalne aplikacije
  - Najčešće se primjenjuje u sloju web-usluga, i to je jedna od ključnih praksi uslužno orijentirane arhitekture.

# 12. Načelo particioniranja podataka

- Umjesto da se cijeli skup podataka klonira na svaki stroj, na svakom stroju se smješta podskup podataka
  - Svaki poslužitelj ima svoj podskup podataka, koji ne dijeli s ostalim poslužiteljima i može ga samostalno kontrolirati pa nema sinkronizacije podataka, ni potrebe za zaključavanjem podataka a greške se lako mogu izolirati
  - Složena tehnika za implementaciju
    - Potrebno je locirati particiju na kojoj se nalaze podaci prije slanja upita poslužiteljima
    - Upiti koji obuhvaćaju višestruke particije vrlo su neučinkoviti i složeni
- Particioniranje podataka i dodavanje klonova omogućuju „učinkovitu skalabilnost”
  - Ako se podaci ispravno razdijele među čvorovima, uvijek se može dodati više korisnika, obraditi više paralelnih veza, prikupiti više podataka i implementirati sustav na više poslužitelja

*http://dev.bizo.com/2013/04/efficiency-scalability.html*

# 13. Načelo visoke dostupnosti

- Sustav se smatra dostupnim sve dok obavlja svoje funkcije onako kako se očekuje iz perspektive klijenata
  - Nije važno da li postoje neke greške unutar sustava koje klijentima nisu vidljive
  - Što je sustav veći to je veća mogućnost pojavljivanja grešaka
- Za poboljšanje otpornosti sustava potrebno je kontinuirano testirati različite scenarije kvarova
  - **Uklanjanje jedinstvenih točaka kvara** (*single point of failure*)
- Uvođenje redundancije za kritične komponente
- Plan oporavka za sve kritične dijelove infrastrukture
  - Na dostupnost sustava ne utječe samo učestalost pojave kvarova nego i vrijeme oporavka
- Da bi sustav bio stalno dostupan i potpuno tolerantan na pogreške
  - Mogućnost samooporavka (*self-healing*): automatsko otkrivanje i rješavanje problema bez ljudske intervencije

# Performanse web aplikacije (1)

- Kod **određivanja performansi web sjedišta** (npr. prije puštanja u rad web trgovine) potrebno je utvrditi sljedeće:
  - Brzina mrežnog prometa
    - Maksimalna i prosječna brzina prijenosa podataka, brzina odziva i latencija mreže
  - Test opterećenja (*load test*)
    - Koliko istodobnih korisnika web aplikacija podržava. Određuje se maksimalni radni kapacitet weba, identificiraju se kritične komponente (*bottlenecks*) i degradacija komponenti.
  - Test krajnjeg opterećenja (*stress test*)
    - Test opterećenja pod maksimalnim opterećenjem i opterećenjem većem od dizajniranog
    - Ispituje se nagli porast korisnika i način povrata nominalnom opterećenju, druge kritične komponente

# Performanse web aplikacije (2)

- Kod određivanja performansi web sjedišta (npr. web trgovine) potrebno je još utvrditi :
  - Test skalabilnosti
    - Nagli porast broja korisnika može negativno utjecati na performanse (npr. vrijeme potrebno za učitavanje stranica)
  - Ispitivanje izdržljivosti (*endurance testing*)
    - Iznadprosječno opterećenje sustava tijekom dužeg vremenskog razdoblja kako bi se provjerilo ponašanje sustava pri dugotrajnoj uporabi
    - U praksi sustav se ne mora ponašati deterministički ili linearno u ovisnosti o vremenu
  - Ispitivanje kapaciteta baze podataka (*volume testing*)
    - Analiza rada sustava na različiti broj zapisa koji se pohranjuju u bazi podataka
    - *Flood testing*

# Strategije dizajna skalabilnih web-aplikacija

- Metode dizajna skalabilnih web-aplikacija mogu se primijeniti na sljedeće elemente njihove arhitekture:
  - *Front-end*
  - *Caching*
  - Web servisi
  - Podatkovni sloj

# *Front-end*

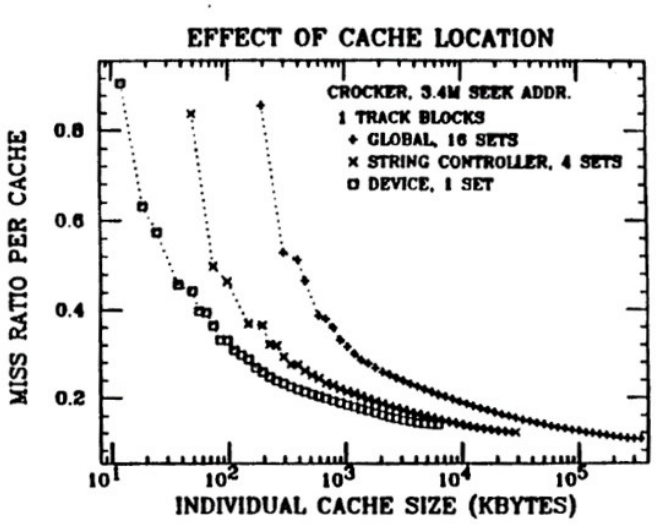
- Korisnik može komunicirati sa aplikacijom putem web stranica, mobilnih aplikacija ili pozivima web servisa
  - *Front-end* se treba koristiti samo kako bi se predstavila funkcionalnost sustava korisnicima
  - Jedina zadaća *front-enda* je korisničko sučelje
  - Logiku treba držati na razini sloja web servisa
  - Izbjeći miješanje funkcionalnosti sa korisničkim sučeljem
  - Korisnička sučelja ne bi trebala biti svjesna podatkovnog sloja ili usluge trećih strana
- Prednosti:
  - Poslužitelji korisničkim sučelja mogu se nezavisno skalirati, jer ne trebaju izvoditi složene operacije
  - Mogu se koristiti programske tehnologije specijalizirane za razvoj sučelja, i druge tehnologije koje su optimizirane za srednji sloj

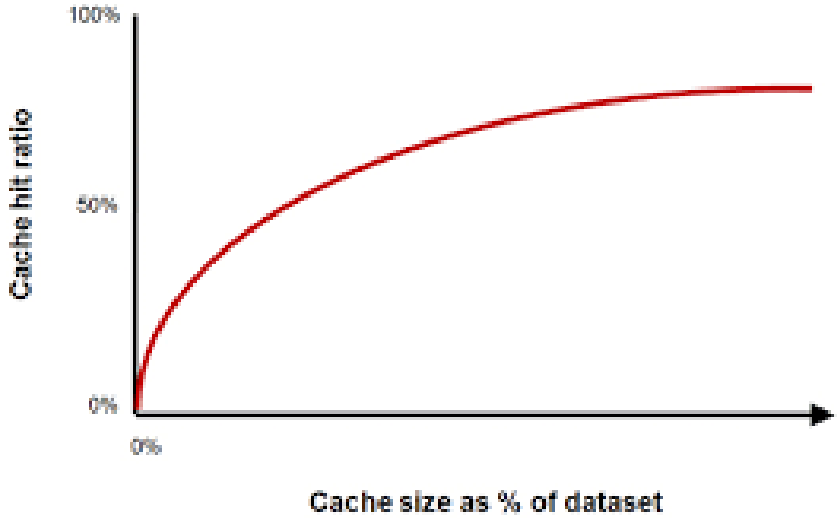
# *Caching* (1)

- Svodi se na spremanje podataka u predmemoriju (*cache*) čime se omogućuje posluživanje zahtjeva za te podatke bez izračunavanja odgovora
  - Vrlo jednostavna i često korištena metoda (CPU, predmemorija tvrdih diskova, predmemorije upita baze podataka, HTTP predmemorija preglednika...)
- Koristi se kako bi se smanjilo vrijeme i resursi potrebni za generiranje rezultata
  - Umjesto dohvaćanja podataka iz izvora ili generiranja odgovora svaki put kada se to zatraži, *caching* gradi rezultat jednom i pohranjuje ga u predmemoriju
  - Naknadni zahtjevi zadovoljavaju se vraćanjem predmemoriranog rezultata sve dok ne istekne ili je izbrisan
  - Budući da se svi predmemorirani predmeti mogu ponovno dohvatiti iz izvora, oni se mogu izgubiti ili izbrisati u bilo kojem trenutku bez posljedica

# *Caching* (2)

- Efektivnost predmemorije: koliko puta možemo ponovno upotrijebiti isti spremljeni odgovor
  - Mjeri se omjerom pogodaka predmemorije (*cache hit ratio*)
- *Caching* je moguće dodati i u kasnijoj fazi, u slučaju ako on nije planiran od samog početka razvoja web aplikacije




*http://www.dataram.com/blog/?p=112*

# Web servisi (1)

- U ovoj strategiji skalabilnosti web servise potrebno je koristiti za izvođenje većinu obrade i gdje se nalazi većina poslovne logike
- Razlikujemo:
  - **Uslužno orijentirana arhitektura (*Service-Oriented Architecture,* SOA):** usmjerena je na slabo povezane i visoko autonomne usluge usmjerene na rješavanje poslovnih potreba. Poželjno je da sve usluge koriste iste komunikacijske protokole.
  - **Višeslojna arhitektura (*Layered Architecture*):** način da se funkcionalnost podijeli u skup (točnije: hijerarhijski organiziran niz) slojeva.
    - Komponente u nižim slojevima izlažu aplikacijsko programsko sučelje (*application programming interface,* API) koje mogu koristiti klijenti koji se nalaze u višim slojevima, ali se nikada ne smije dopustiti da niži slojevi ovise o funkcionalnosti koju pružaju viši slojevi.
    - Primjer je TCP / IP programski stog, gdje svaki sloj dodaje funkcionalnost i ovisi o sloju ispod.

# Web servisi (2)

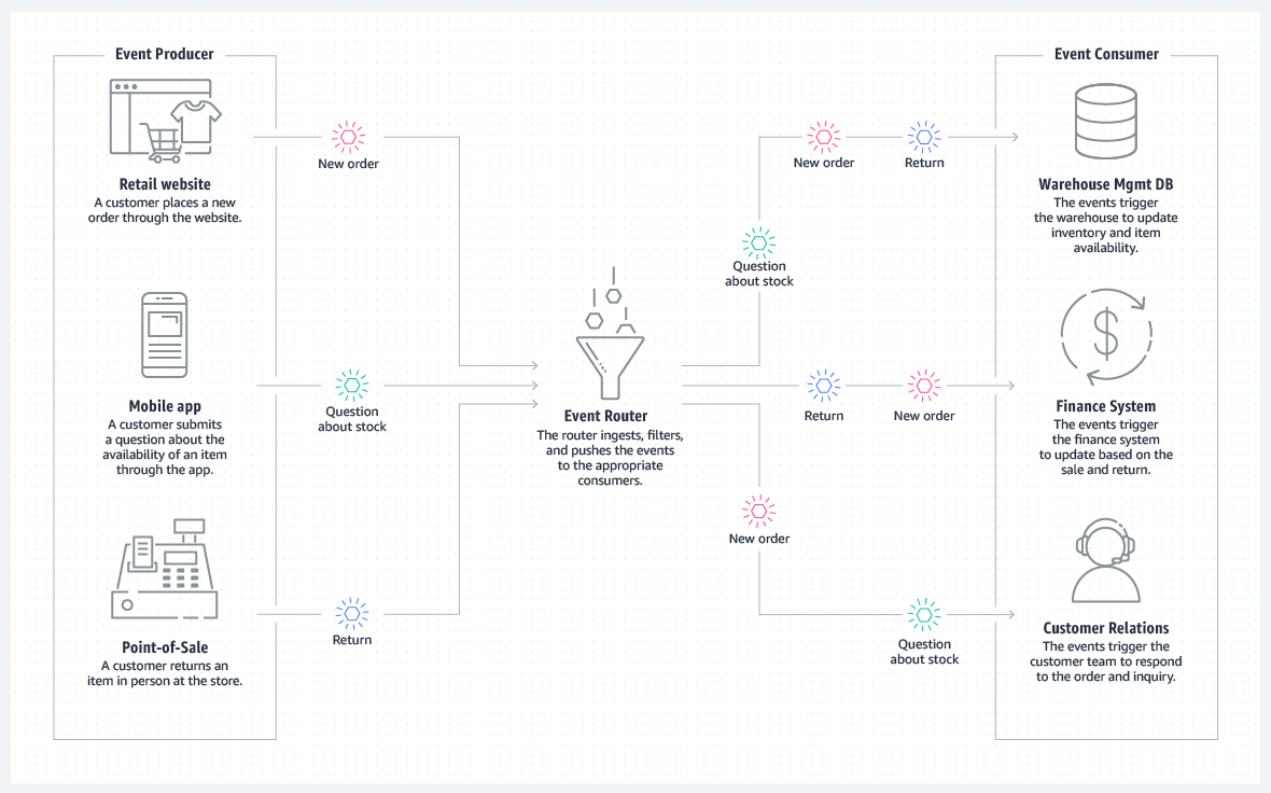
- Razlikujemo (nastavak):
  - **Šesterokutna arhitektura (*Hexagonal Architecture*)**: pretpostavlja da je poslovna logika u središtu arhitekture i da su sve interakcije s spremištima podataka, klijentima i drugim sustavima jednake. Korisnici koji komuniciraju s aplikacijom se u načelu ne razlikuju od sustava baze podataka s kojim je aplikacija u interakciji. Oboje se nalaze izvan poslovne logike aplikacije i oboje trebaju strog ugovor. Definiranjem tih granica može se osobu zamijeniti automatiziranim testnim programom ili bazu podataka s drugim mehanizmom za pohranu bez utjecaja na jezgru sustava.

    *https://medium.com/idealo-tech-blog/hexagonal-ports-adapters-architecture-e3617bcf00a0*
  - **Arhitektura vođena događajima (*Event-Driven Architecture,* EDA)**: temelji se na reagiranju na događaje koji su se već dogodili za razliku od tradicionalnih arhitektura koje se temelje na zahtjevima odnosno odgovaranju na zahtjeve. U tradicionalnom programskom modelu klijent šalje zahtjev i čeka potvrdu da je taj zahtjev izvršen nakon čega nastavlja s obradom. U modelu upravljanom događajima, nema takvog čekanja nego se komunikacija s drugim komponentama svodi na objavljivanje događaja za koje znamo da su se već dogodili nakon čega se nastavlja s obradom.

# Arhitektura vođena događajima – primjer

- Događaji pokreću komunikaciju između odvojenih usluga, koristi se u modernim aplikacijama izgrađenim na mikrouslugama.
- Tri ključne komponente arhitekture: **proizvođači** događaja (*producer*), **usmjerivači** događaja (*router*) i **potrošači** događaja (*consumer*).

- Proizvođač objavljuje događaj na usmjerivaču, koji filtrira i gura događaje potrošačima. Usluge proizvođača i potrošačke usluge su odvojene, što im omogućuje da se samostalno skaliraju, ažuriraju i implementiraju.




*https://aws.amazon.com/event-driven-architecture/*

# Podatkovni sloj

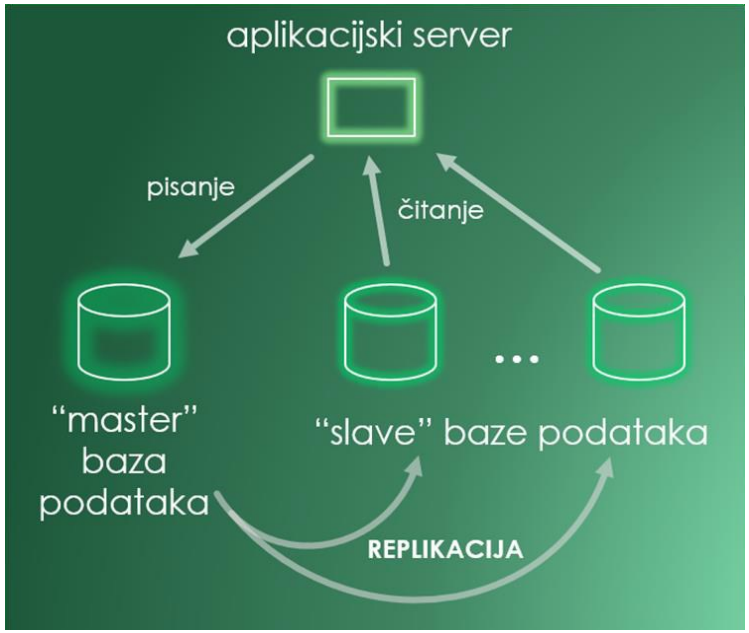
- Strategija skalabilnosti usmjerena na podatkovni sloj, tj. bazu podataka web aplikacije
- Dva glavna razloga za skaliranje baze podataka:
  - 1) prevelik broj pristupa bazi
  - 2) prevelika količina podataka koje treba pohraniti
  - Prvi se problem može riješiti replikacijom, a drugi particioniranjem podataka.

- **Replikacija:** radi se u slučaju da ima puno zahtjeva za čitanje, a malo za pisanje – baza podataka se malo mijenja, a velik broj upita.

# Podatkovni sloj – replikacija

- Kod **replikacije** podatke iz glavne baze podataka (*master*) dupliciramo u pomoćne baze podataka (*slave data base*). Prilikom svakog unosa podataka u master, ažuriraju se i slave baze podataka. Upiti na baze se sada mogu raspodijeliti na više baza, što smanjuje opterećenje glavne baze. To ne povećava brzinu pisanja, ali glavna baza mora podržati zahtijevanu količinu pisanja.

- Možemo replicirati i replike. Problem koji se ovdje pojavljuje je kašnjenje replikacije (***replication lag***) u slučaju da se čita iz replicirane baze prije nego je podatak proslijeđen iz master baze.

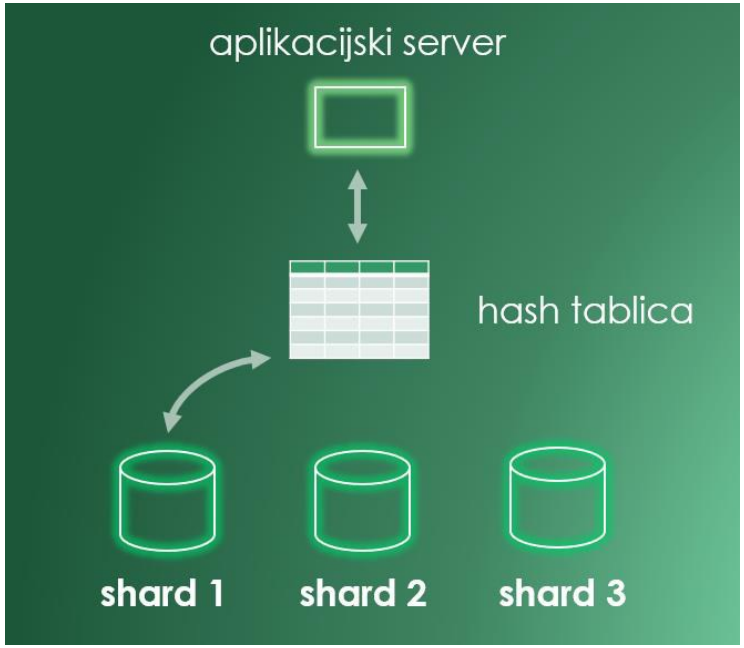


# Podatkovni sloj – *sharding*

- **Postupak particioniranja (*sharding*):** koristi se u slučaju da količina podataka za pohranu nadmašuje kapacitet sekundarne memorije („podaci ne stanu na disk”)
- Kod particioniranja umjesto jednog *master* poslužitelja, imamo više *mastera* jednake veličine.

- U svaki *master* sprema se po dio podataka, raspodijeljenih po ključu zapisanom u tablici raspršenih vrijednosti (*hash* tablica).

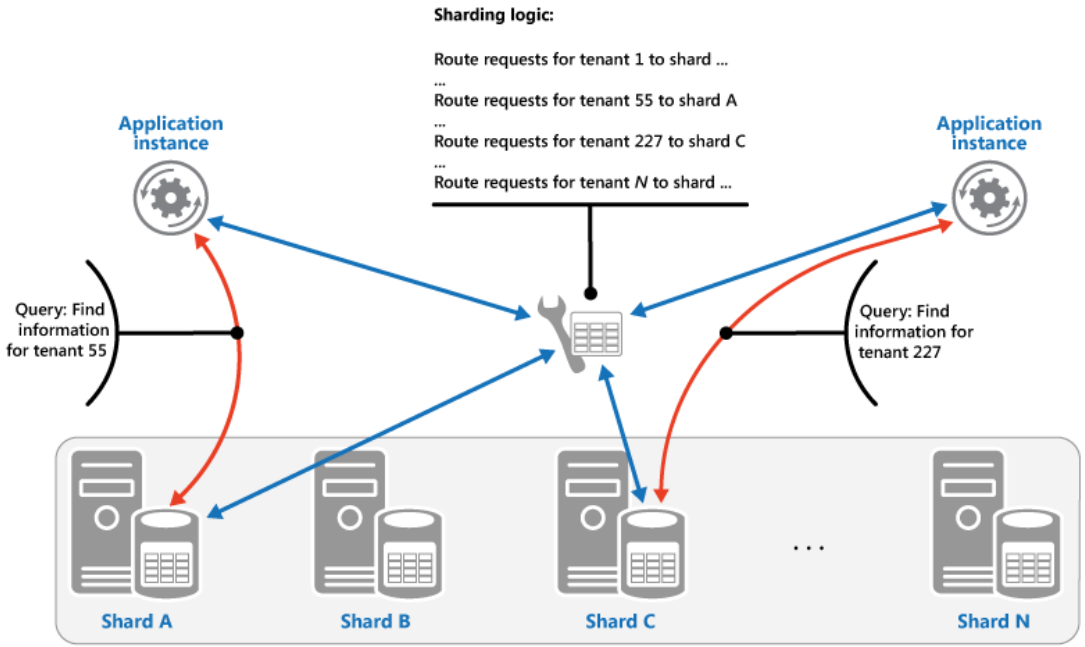
- Postupak particioniranja može se dodatno kombinirati s replikacijom i svaki od tih *mastera* replicirati.



# Podatkovni sloj – *sharding* – nedostaci (1)

- Nedostaci postupka particioniranja:
  - Problem kod particioniranja podataka su **kompleksni upiti**, npr. ako se traže podaci sortirani po nekom drugom ključu, morat će se pristupiti svim bazama podataka.
  - Drugi problem su **povezani upiti**. Korištenjem tehnike *shardinga* postaje teško izvesti povezane upite koji koriste više tablica (*joinove)*, jer su tablice razbacane na više baza ili računala.

- U slučaju korištenja *shardinga* podatkovni model je potrebno dizajnirati tako da ne zahtijeva kompleksne upite i *joinove*.



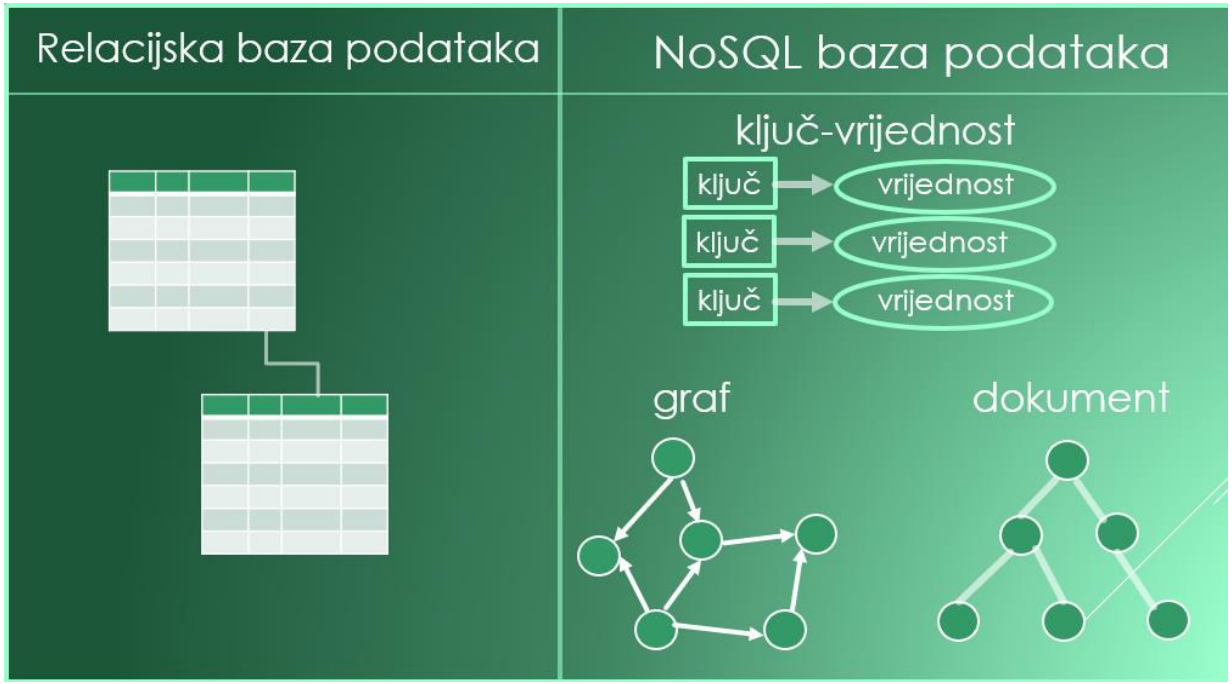

*https://docs.microsoft.com/en-us/azure/architecture/patterns/sharding*

# Podatkovni sloj – *sharding* – nedostaci (2)

- Kod relacijskih baza podataka svaki je podatak unutar tablice povezan sa ostalim podacima u toj tablici. Između tablica često postoje veze poznate kao strani ključevi. Većina aplikacija ovisi o bazi podataka koja podržava te veze zbog tzv. **ACID svojstava**:
  - **Atomicity** – atomnost, sve operacije u transakciji će se završiti, ili neće niti jedna
  - **Consistency** – dosljednost, baza podataka će biti u konzistentnom stanju kada transakcija počinje i završava
  - **Isolation** – izolacija, transakcija će se ponašati kao da je jedina operacija koja se izvodi na bazi podataka, jamči da se transakcije mogu izvoditi paralelno bez međusobnog utjecaja
  - **Durability** – trajnost, garantira da se podaci zadržavaju prije vraćanja klijentu, tako da se jednom dovršena transakcija nikada ne može izgubiti, čak ni zbog kvara poslužitelja
- Zahtijevanje da baza podataka provodi te odnose čini particioniranje baze podataka složenim.

# Podatkovni sloj – *sharding* – NoSQL

- NoSQL je široki pojam koji se koristi za označavanje raznih tipova pohrane podataka koji se razlikuju od tradicionalnog modela relacijskih baze
- NoSQL baze su više specijalizirane i pogodnije za skaliranje od tradicionalnih SQL baza. Pogodne su za pohranjivanje složenijih podatkovnih struktura, ali nisu fleksibilne. Nije moguće postaviti bilo kakav upit na podatke i u slučaju dodavanja funkcionalnosti često je potreban potpuni redizajn NoSQL baze.



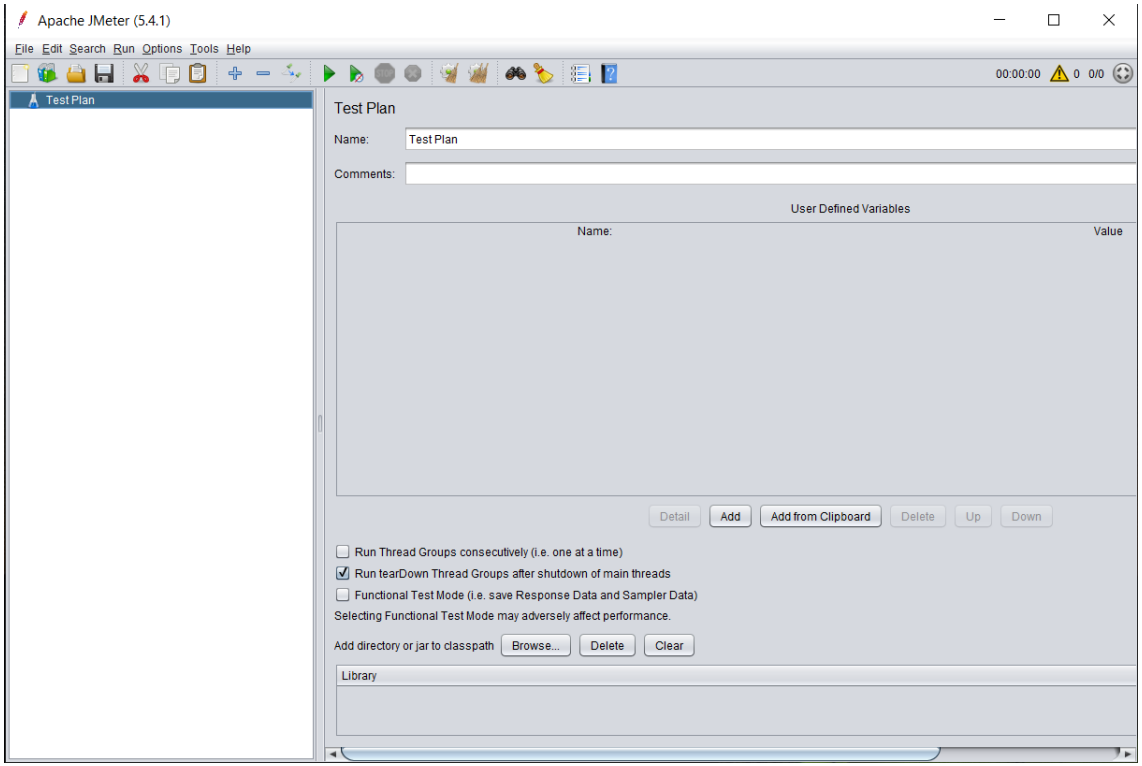
# Apache JMeter

# JMeter

- Apache JMeter testni alat omogućuje snimanje HTTP zahtjeva prema poslužiteljima i testiranje njihovih performansi
  - Desktop aplikacija otvorenog koda, besplatna, napisana u programskom jeziku Java
  - Provodi testove opterećenja (*load test, stress test*) koji su dio testa prihvaćanja (*acceptance test*) i potrebni prije puštanja aplikacije u produkciju
  - Performanse statičkih resursa (JavaScript i HTML) i dinamičkih (JSP, servleti, AJAX komponente)
  - Pomaže odrediti maksimalni broj istodobnih korisnika web aplikacije
  - Podržava izvještavanje, grafovi i izvješća o učinku (*performance reports*)
  - Preuzimanje: https://jmeter.apache.org/download_jmeter.cgi

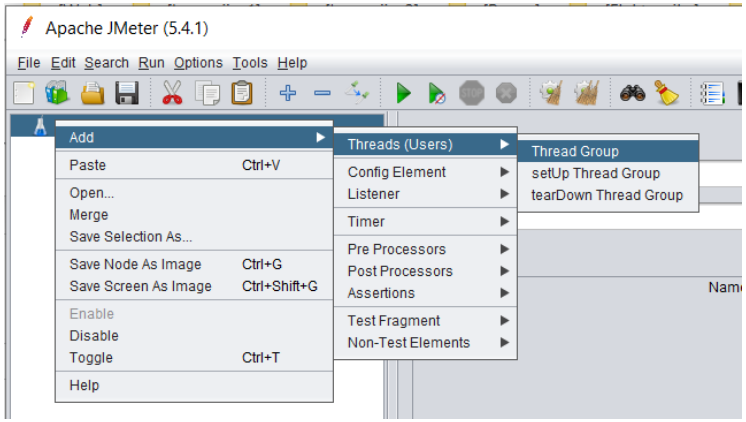
# Izrada *Performance Test Plan* (1)

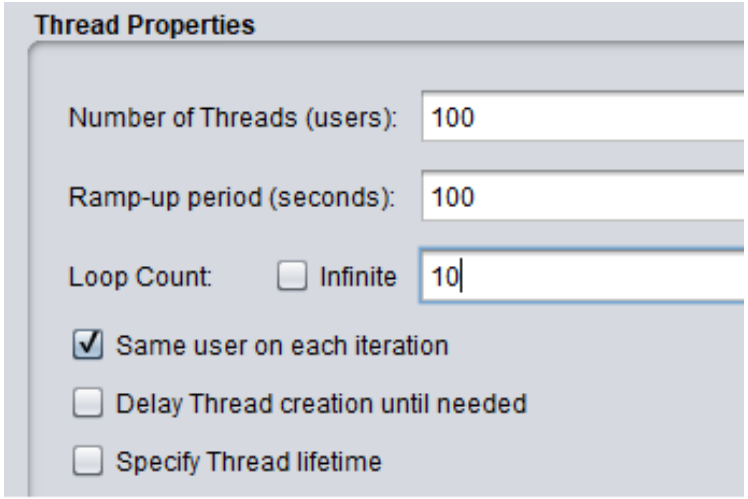
- Postupak izrade *Performance Test Plan*
- Pokrenuti aplikaciju:
  - jmeter.bat
  - ApacheJMeter.jar



# Izrada *Performance Test Plan* (2)

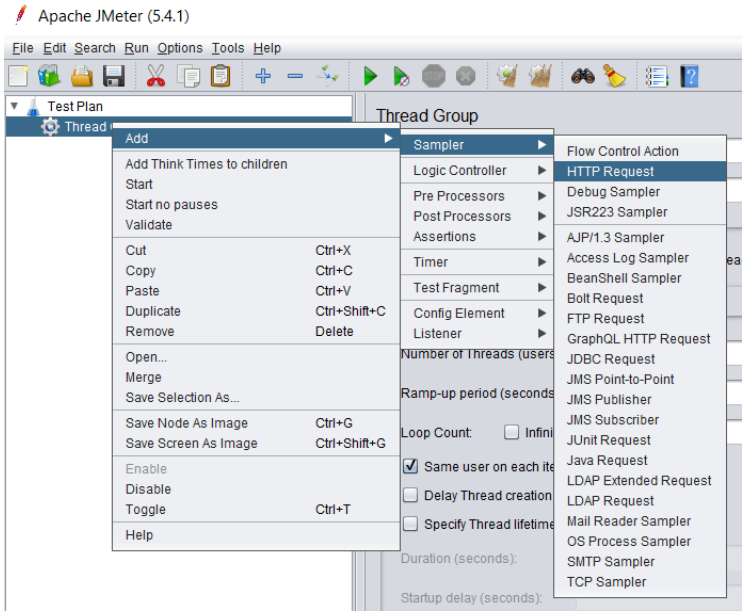
- Odabrati:
  - Add Thread Group
    - Number of Threads: 100
      - Broj simuliranih korisnika koji će povezati s web stranicama
    - Ramp-Up Period: 100
      - Koliko dugo će aplikacija čekati u sekundama prije pokretanja novog simuliranog korisnika. Primjerice, ako imamo 100 korisnika i Ramp-Up period je 100 s, onda će kašnjenje između pokretanja dva uzastopna korisnika biti 1 s (100 s /100 korisnika)
    - Loop Count: 10
      - Koliko puta će se test pokrenuti

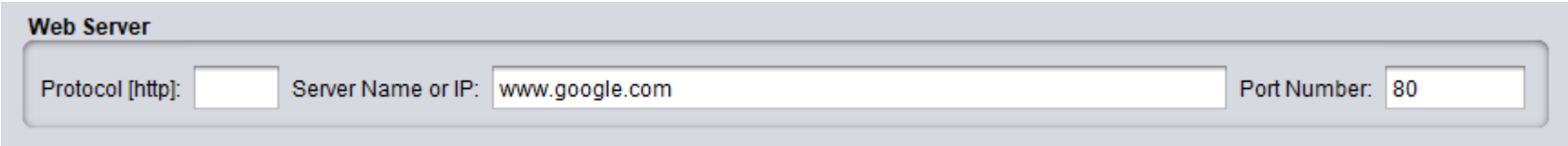


# Izrada *Performance Test Plan* (3)

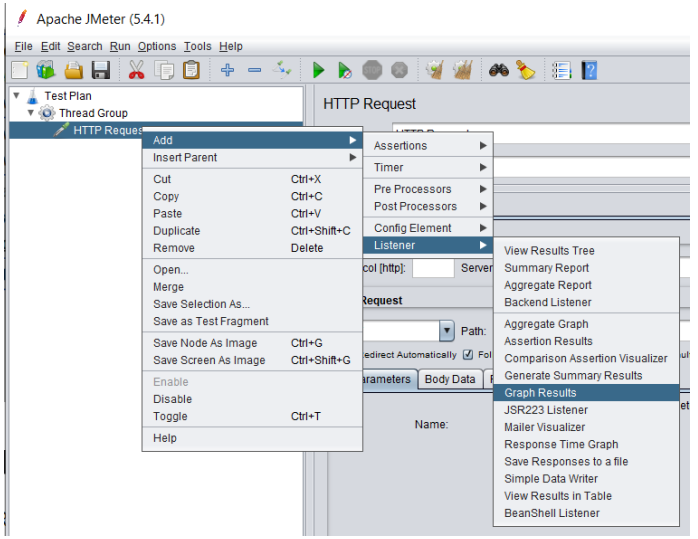
- Dodati JMeter elemente:
  - HTTP Request
    - U polje „Server Name or IP” unesite URL ili IP adresu poslužitelja koji testirate, npr. www.google.com
    - Unesite port na koji se šalju zahtjevi (80)
    - U polje „Path” unesite dodatne elemente zahtjeva koji šaljete na poslužitelja, npr. ako ovdje upišete „calendar” JMeter će izraditi zahtjev „http://www.google.com/calendar” i poslati ga na Google poslužitelj

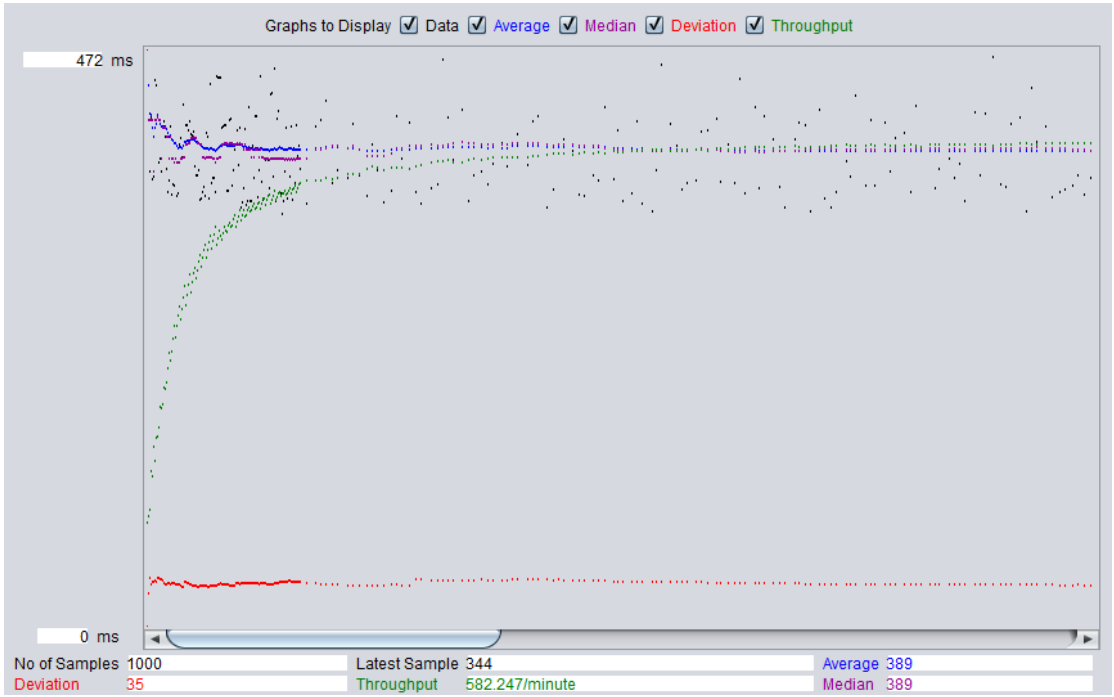


# Izrada *Performance Test Plan* (4)

- Dodati izvještaje - vizualizacija rezultata testiranja
  - Odabrati „Graph Results” u meniju „Listener”
  - Podržano je više vrsta izvještavanja
  - Pokrenuti test („Run Test”)  (ili CTRL+R)
  - Rezultati se prikazuju u stvarnom vremenu




Graf prikazuje rezultat plana testiranja u kojemu je simulirano 100 korisnika koji pristupaju web stranicama www.google.com

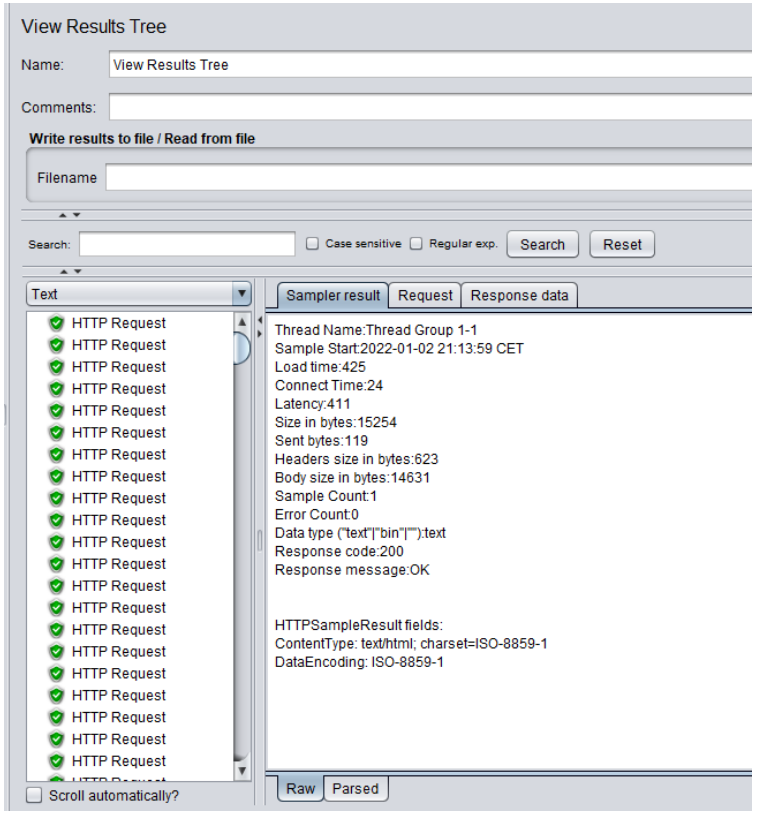
# Izrada *Performance Test Plan* (5)

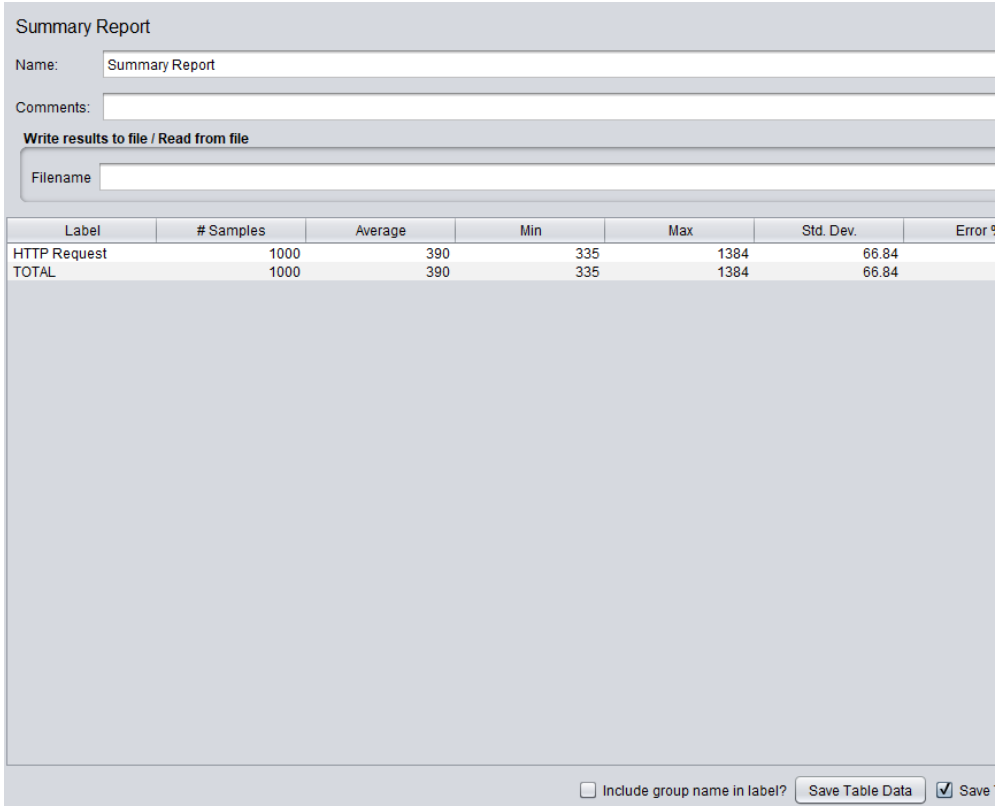
- Analiza rezultata testa
  - Rezultati su predstavljeni u sljedećim bojama:
    - Crna: Ukupan broj poslanih zahtjeva
    - Plava: Trenutni prosjek svih poslanih zahtjeva
    - Crvena: Trenutna standardna devijacija
    - Zelena: Stopa propusnosti (*throughput rate*) koja predstavlja broj svih zahtjeva u minuti koje je poslužitelj obradio
  - Za analizu test opterećenja osobito su važne dvije vrijednosti:
    - Propusnost (*throughput*)
    - Devijacija (*deviation*)
  - Propusnost predstavlja sposobnost poslužitelja da podnosi visoku razinu opterećenja zahtjevima. Viša vrijednost je bolja.
  - U ovom testu propusnost Google poslužitelja bio je 582.247/minute, tj. toliko zahtjeva je poslužitelj mogao obraditi u 1 minuti.
  - Devijacija je u crvenoj boji (35). Manja vrijednost je bolja.

- Isti test može se pokrenuti više puta, rezultati će se agregirati

# *View Results Tree* i *Summary Report*

- Mogućnost View Results Tree
  - Vidljivi su detalji pojedinih HTTP zahtjeva
- Summary Report
  - Završni tablični izvještaj



# Ostale aplikacije

- Selenium WebDriver – aplikacija za automatizaciju web preglednika (automatsko upravljanje), W3C preporuka, https://www.selenium.dev/documentation/webdriver/
- Postman – izrada i ispitivanje performansi aplikacijskih programskih sučelja, https://www.postman.com/
- Pact – ispitivanje integriteta HTTP poruka (*contract testing*), npr. za mikroservisne arhitekture, https://docs.pact.io/