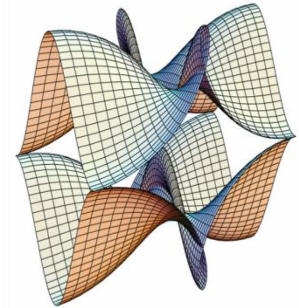
Sveučilište u Zagrebu
PMF – Matematički odsjek



SOFTVERSKO INŽENJERSTVO
Predavanja 2023/2024

Poglavlje 7: Novi trendovi u softverskom inženjerstvu

Sastavio: Robert Manger
24.01.2024

O čemu je riječ u Poglavlju 7

- U ovom poglavlju
  - Obrađujemo nekoliko tema koje su danas u središtu pažnje i smatraju se novim trendovima u softverskom inženjerstvu.
  - Te teme zahvaćaju razne dijelove procesa razvoja, korištenja i održavanja softvera.
  - Teško je predvidjeti na koje će se sve načine one dalje mijenjati te kolika će biti njihova stvarna važnost za softversko inženjerstvo.

Sadržaj Poglavlja 7

7.1. Softver u oblaku

7.2. Web servisi i mikro-servisi

7.3. DevOps

Što je oblak

- Oblak (cloud) razvio se zato što:
  - postoje snažna višejezgrena računala
  - postoje brze i propusne mreže za udaljeni pristup do tih računala.
- Oblak je veliki skup udaljenih poslužiteljskih računala (u nastavku: poslužitelja) koje vlasnici nude u najam korisnicima.
- Kompanije koji raspolažu svojim oblacima i nude ih korisnicima:
  - Amazon, Google, Microsoft, Oracle, SRCE, …

Korištenje oblaka

- Korisnik može unajmiti onoliko poslužitelja koliko mu treba te instalirati i pokretati softver na njima.
- Korisnik može pristupiti unajmljenom poslužitelju s vlastitog računala ili mobitela ili tableta.
- Korištenje oblaka mnogim korisnicima je jeftinije nego nabava vlastitog hardvera.

Fleksibilnost pri korištenju oblaka

- Vlasnik oblaka daje korisnicima na upotebu "cloud management software" koji im omogućuje:
  - uzimanje novih poslužitelja
  - otpuštanje neiskorištenih poslužitelja
  - konfiguriranje softvera na svakom poslužitelju.
- Na taj način, softver koji se izvršava na oblaku može zadovoljiti tri važna svojstva:
  - skalabilnost (scalability)
  - elastičnost (elasticity)
  - otpornost (resilience).

Skalabilnost

- Skalabilnost je održavanje zadovoljavajućih performansi softvera onda kad se opterećenje povećava.
- Dakle to je sposobnost softvera da izađe na kraj s povećanim brojem korisnika.
- Postiže se:
  - Dodavanjem novih poslužitelja koji će izvršavati kopije softvera (scaling out)
  - ili migracijom softvera na jači poslužitelj (scaling up).

Elastičnost

- Elastičnost je prilagođavanje poslužiteljske konfiguracije u skladu s promijenjenim zahtjevima.
- Uključuje “scaling down” uz “scaling up”.
- Prati se opterećenje na softver, pa se dinamički mijenja broj ili snaga poslužitelja kako se mijenja broj korisnika.
- Cilj je da korisnici plaćaju samo za poslužitelje koje zaista trebaju i kad ih trebaju.

Otpornost

- Otpornost je održavanje usluge koju softver pruža čak i u slučaju “rušenja” poslužitelja.
- Postiže se na sljedeći način:
  - Stvara se nekoliko kopija softvera koje su istovremeno dostupne.
  - Svaka kopija instalirana je na drugom poslužitelju na različitim fizičkim lokacijama.
  - Ako se jedan poslužitelj sruši, ostali preuzimaju posao.

Upotreba oblaka za razvoj softvera

- Osim što korisnici mogu koristiti oblak za rad sa softverom, proizvođači softvera mogu ga koristiti za razvoj softvera.
- Prednosti razvoja softvera pomoću oblaka:
  - Cijena … izbjegava se početni kapitalni trošak nabave hardvera.
  - Brz početak rada … ne mora se čekati isporuka hardvera.
  - Izbor poslužitelja … ako se iznajmljeni poslužitelj pokaže preslab, možemo ga zamijeniti jačim, ili možemo dodati nove poslužitelje.
  - Distribuirani razvoj … članovi razvojnog tima mogu raditi s različitih lokacija i dijeliti istu radnu okolinu.

Virtualizacija

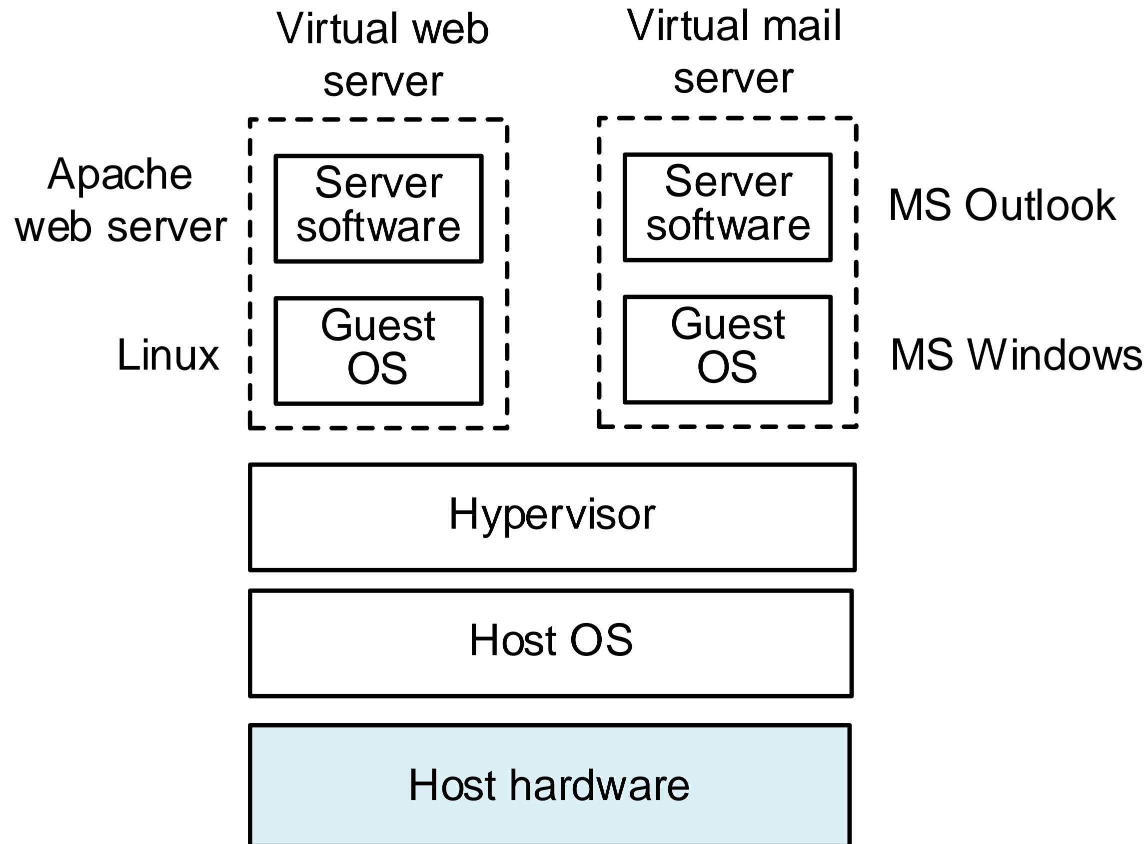
- Poslužitelji u oblaku u pravilu su virtualni.
- Više virtualnih poslužitelja može se izvršavati na jednom hardverskom čvoru.
- Danas su u upotrebi dvije tehnologije za virtualizaciju:
  - Takozvana VM tehnologija (virtual machine).
  - Kontejneri (containers).

VM tehnologija (1)

- Zasniva se na emulaciji virtualnih (guest) računala na stvarnom fizičkom (host) računalu.
- Emulaciju obavlja posebni softver, takozvani Hypervisor.
- Hypervisor upravlja hardverom i razdvaja fizičke resurse od virtualnih resursa.
- Kad softver koji se izvršava na virtualnom računalu treba neki resurs, hypervisor emulira taj resurs služeći se zajedničkom zalihom fizičkih resursa.

VM tehnologija (2)

- Primjer fizičkog računala koje emulira dva virtualna poslužitelja:



VM tehnologija (3)

- Najpoznatiji konkretni primjeri za Hypervisor:
  - VMWare Workstation
  - Oracle VM VirtualBox
  - Microsoft Hyper-V

- Prednost VM tehnologije je da različita virtualna računala na istom fizičkom računalu mogu koristiti različite guest OS. Npr. na slici:
  - Apache web server radi pod Linux-om
  - MS Outlook radi pod MS Windows.

VM tehnologija (4)

- Nedostatak VM tehnologije je da je prilično glomazna i spora.
  - Stvaranje novog guest računala zahtijeva učitavanje velikog i složenog guest OS.
  - Vrijeme potrebno za instalaciju guest OS i ostalog softvera može trajati između 2 i 5 minuta.
  - Ne možemo brzo reagirati kod povećanja opterećenja na korišteni softver tako da pokrećemo nova virtualna računala.

Kontejneri (1)

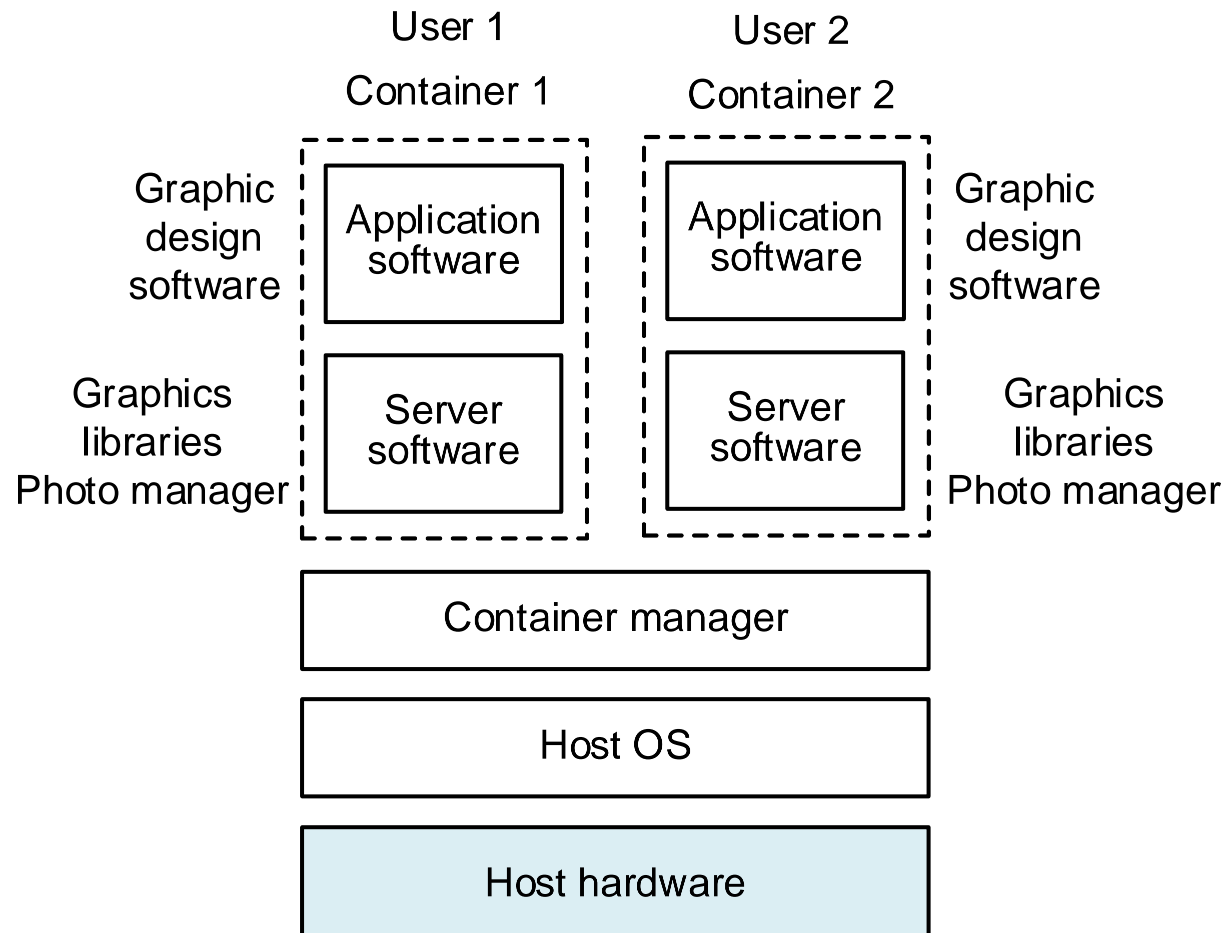
- Često nam nije potrebna općenitost VM tehnologije u pogledu korištenja različitih OS.
  - Ako u oblaku pokrećemo mnogo instanci istog softvera, tada sve te instance koriste isti OS.
- U takvim slučajevima treba koristiti kontejnere.
  - Oni su jednostavnija “lightweight” tehnologija za virtualizaciju.
  - Veličina im se mjeri u Mbyte, za razliku od virtualnih poslužitelja čija veličina se mjeri u Gbyte.
  - Drastično se ubrzava stvaranje virtualnih poslužitelja u oblaku (treba nekoliko sekundi).

Kontejneri (2)

- Kontejneri su tehnologija za virtualizaciju gdje nezavisni virtualni poslužitelji dijele isti OS.
  - Efikasni su kad želimo pružati izolirane usluge gdje svaki korisnik radi s vlastitom instancom softvera.
  - Posebno su efikasni za pokretanje malih dijelova softvera kao što su samostalni servisi.
- Svim kontejnerima na istom host računalu upravlja container manager.
  - On osigurava da je proces koji se izvršava u jednom kontejneru potpuno izoliran od procesa u drugom kontejneru.

Kontejneri (3)

- Primjer računala koje pokreće dvije instance aplikacije za grafičko oblikovanje. Svaki korisnik ima svoju instancu.



Kontejneri (4)

- Kontejneri nisu najbolji mehanizam za pokretanje velike baze podataka – tada je ipak bolja VM tehnologija.
- VM tehnologija i kontejneri mogu koegzistirati na istom fizičkom računalu.
  - Container manager i svi kontejneri rade na jednom virtualnom poslužitelju.
  - Zajednička baza podataka radi na drugom virtualnom poslužitelju.
  - Instance aplikacije koje se izvršavaju u kontejnerima mogu svaka zasebno pristupati bazi podataka.

Tehnologije za rad s kontejnerima

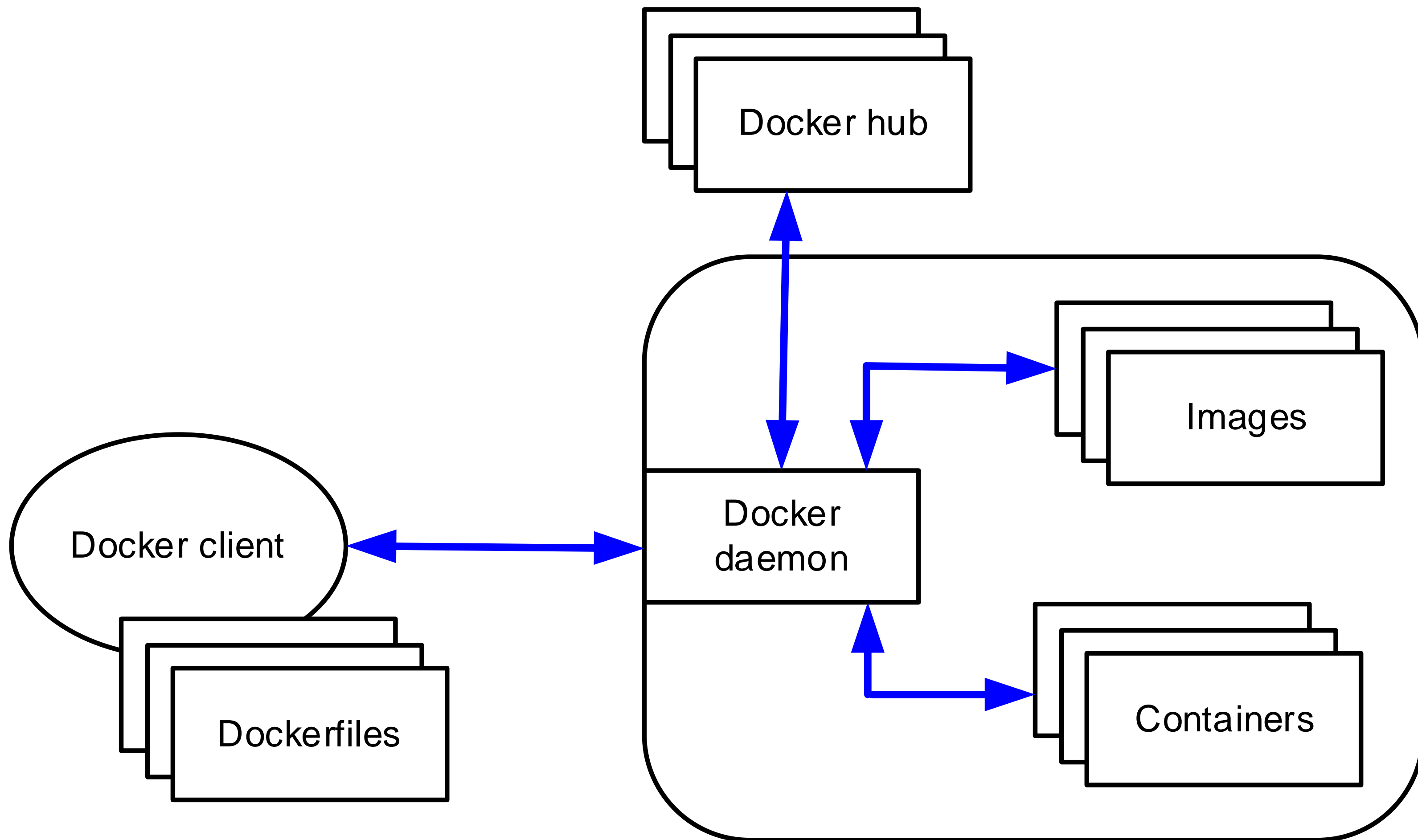
- Ideja o kontejnerima pojavila se u 2000-tim god.
  - Prve verzije nastale su se unutar Linux zajednice.
  - Google i Amazon razvijale su svoje verzije da bi upravljale svojim klasterima poslužitelja.
- No kontejneri su zaista postali mainstream tehnologija tek oko 2015. godine, zahvaljujući open-source projektu Docker.
  - Projekt je stvorio standardne alate za upravljanje kontejnerima (besplatni, brzi i lagani za korištenje).
  - To je danas najraširenija tehnologija za kontejnere.

Docker (1)

- Docker je sustav za upravljanje kontejnerima.
  - Omogućuje korisnicima da definiraju softver koji treba biti uključen u kontejneru. Takva definicija naziva se Docker image.
  - Uključen je i run-time sustav koji može stvoriti kontejnere i upravljati njima služeći se Docker image-ima.
  - Docker images mogu se arhivirati, dijeliti i pokretati na raznim Docker host-ovima.
  - Sve što je potrebno za rad softverskog sustava (binarne datoteke, biblioteke, sistemski alati) uključeno je u Docker image.

Docker (2)

Elementi Dockerovog sustava:



Docker (3)

- Funkcija svakog od elemenata.
  - Docker deamon … Proces koji radi na host računalu i služi za postavljanje, startanje, zaustavljanje i praćenje kontejnera, također za gradnju i upravljanje lokalnih Images
  - Docker client … Softver koji koriste razvojni inženjeri da bi definirali kontejnere i upravljali njima.
  - Dockerfile … Definira Image navođenjem naredbi koje specificiraju softver koji treba uključiti u kontejner. Svaki kontejner mora biti definiran preko pridruženog Dockerfile-a.
  - Image … Dockerfile se interpretira da bi se stvorio Image – to je skup direktorija s navedenim softverom i podacima instaliranim na pravim mjestima.

Docker (4)

- Funkcija svakog od elemenata (nastavak).
  - Docker hub … Registar svih Images koje su bile stvorene. Ti images mogu se ponovo upotrijebiti da bi se pokrenuli kontejneri, ili služe kao početak za definiranje novih Images.
  - Container … Container je Image koji se izvršava. Image se “load”-a u Container te se aplikacija definirana preko Image počinje izvršavati. Containeri se mogu prebacivati s poslužitelja na poslužitelj bez modifikacije te replicirati na mnogo poslužitelja. Nad Container-om mogu se napraviti promjene (npr. promjena datoteka), no tada se te promjene moraju “commit”-ati da bi se stvorio novi Image te restartao Container.

Docker (5)

- Docker image može se interpretirati kao samostalni file system za virtualni poslužitelj.
  - Pritom image uključuje samo datoteke koje se razlikuju od standardnih datoteka iz OS.
  - Zbog toga je image kompaktan i brzo se učitava.
- Docker ipak nije neovisan o OS host računala.
  - Taj OS trebao bi biti Linux.
  - Ipak, Docker kontejneri mogu se izvršavati na MS Windows ili MacOS računalima, tako da se na tim računalima uspostavi virtualni Linux poslužitelj preko odgovarajuće VM tehnologije.

Prednosti kontejnera za soft inž (1)

- Oni rješavaju problem softverskih ovisnosti.
  - Ne moramo brinuti o bibliotekama i drugom softveru na ciljanom poslužitelju koji se može razlikovati od onoga na razvojnom poslužitelju.
  - Umjesto da isporučimo samo naš softverski produkt, isporučujemo kontejner koji uključuje sav pomoćni softver koji naš produkt treba.
- Oni predstavljaju mehanizam za portabilnost softvera između različitih oblaka.
  - Npr. Docker kontejneri mogu se izvršavati na bilo kojem računalu gdje je prisutan Docker deamon.

Prednosti kontejnera za soft inž (2)

- Oni predstavljaju efikasan mehanizam za implementiranje softverskih servisa.
  - Bit će objašnjeno u potpoglavlju 7.2.
- Oni olakšavaju primjenu DevOps-a.
  - Riječ je o načinu održavanja softvera gdje je isti tim odgovoran i za razvoj softverskog produkta i za podršku pri korištenju tog produkta.
  - O tome ćemo govoriti u potpoglavlju 7.3.

“Everything as a Service” (1)

- Za računanje u oblaku od velike važnosti je ideja “unajmiti umjesto posjedovati”.
- Postoje tri stupnja te ideje koji su zajedno obuhvaćene krilaticom “Everything as a Service”.

  1. Infrastructure as a Service - IaaS

     Umjesto da posjeduje svoj poslužitelj, korisnik unajmljuje virtualni poslužitelj od nekog vlasnika oblaka. Unajmljeni virtualni poslužitelj ponaša se kao “goli hardver” na koji korisnik treba instalirati kupljeni sistemski softver (npr. OS, DBMS) i kupljenu aplikaciju.

“Everything as a Service” (2)

- Tri stupnja za ideju unajmljivanja umjesto posjedovanja (nastavak).

2. Platform as a Service - PaaS

   Korisnik od vlasnika oblaka unajmljuje već konfigurirani virtualni poslužitelj na kojem je već instaliran sistemski softver (npr. OS, DBMS). Na takav poslužitelj korisnik instalira samo kupljenu aplikaciju.

3. Software as a Service - SaaS

   Korisnik od proizvođača aplikacije ne kupuje tu aplikaciju nego unajmljuje pravo njezinog korištenja. Aplikacija radi na proizvođačevom poslužitelju (bilo posjedovanom bilo unajmljenom). Korisnik pristupa aplikaciji npr. preko web sučelja ili preko klijenta na mobitelu.

Software as a Service – SaaS (1)

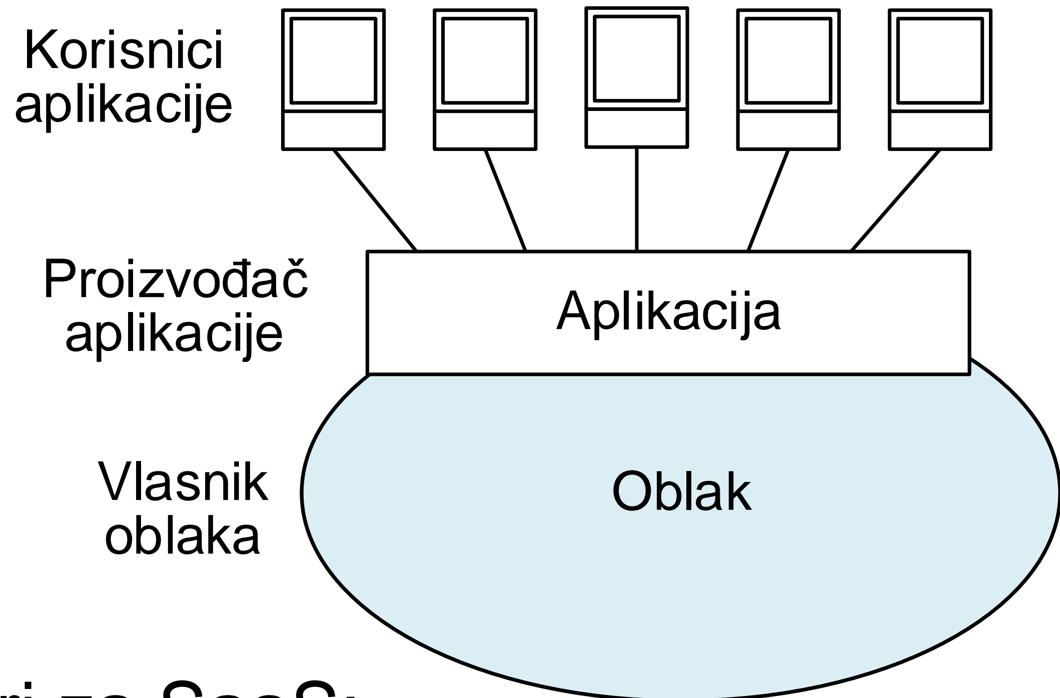
- Riječ je o vrlo naglašenom trendu u današnjem softverskom inženjerstvu.
- Proizvođači aplikacija sve manje prodaju svoje aplikacije, a sve više ih iznajmljuju korisnicima.
  - Aplikacija se ne instalira na korisnikovo računalo nego se izvršava na proizvođačevom poslužitelju.
  - Proizvođačev poslužitelj u pravilu je virtualan i unajmljen je od nekog vlasnika oblaka.

Software as a Service – SaaS (2)

- Primijetimo da u SaaS postoje tri subjekta.
  - Korisnik aplikacije (user): pristupa aplikaciji preko web sučelja ili preko klijenta na mobitelu.
  - Proizvođač aplikacije (application software provider): razvija, stavlja na raspolaganje i održava aplikaciju, koja radi na virtualnom poslužitelju, koji je unajmljen od vlasnika oblaka.
  - Vlasnik oblaka (cloud vendor): iznajmljuje virtualni poslužitelj sa ili bez instaliranog sistemskog softvera.
- Model plaćanja je obično pretplata:
  - Mjesečna ili godišnja pristojba koju korisnik plaća proizvođaču.

Software as a Service – SaaS (3)

- Ilustracija ideje SaaS:



- Poznati primjeri za SaaS:
  - Gmail, Google Disk, Google Calender … besplatni
  - MS Office 365 … plaća se pretplata
  - Wolfram Mathematica OnLine … besplatno

Software as a Service – SaaS (4)

- Mnogi proizvođači aplikacija ipak još uvijek kombiniraju model SaaS s klasičnim modelom prodaje. Dakle oni:
  - Iznajmljuju aplikaciju kao uslugu u oblaku.
  - Istovremeno dozvoljavaju korisnicima da kupe i instaliraju verziju aplikacije namijenjenu za rad na korisnikovom računalu.
- Klasični model je pogodniji onda kad korisnik nije spojen na mrežu ili kad je mreža prespora.
- Primjeri softvera koji su dostupni u oba modela:
  - Wolfram Mathematica
  - Adobe Lightroom Photo Management
  - MS Office.

Prednosti od SaaS za proizvođača

- Tok novca … Korisnici stalno pomalo plaćaju svoje pretplate. Nema naglih skokova u dotoku novca.
- Olakšano održavanje … Svi korisnici koriste isto izdanje aplikacije. Nema potrebe za održavanjem više izdanja.
- Kontinuirano stavljanje u upotrebu … Novo izdanje aplikacije može se staviti u upotrebu odmah nakon što je napravljeno i testirano. Brzo se ispravljaju bugovi.
- Fleksibilnost kod naplate … Moguće je uvesti više opcija za plaćanje pretplate, što će privući širi krug korisnika.
- Skupljanje podataka … Lako se skupljaju podatci o tome kako se aplikacija koristi, pa se tako identificiraju područja gdje aplikaciju treba poboljšati. Mogu se skupljati podatci o samim korisnicima te zatim reklamirati druge proizvode tim istim korisnicima.

Prednosti od SaaS za korisnike

- Nema kapitalnih troškova … Nije potrebno kupovati vlastite poslužitelje niti samu aplikaciju. Umjesto jednog velikog početnog troška imamo puno malih troškova pretplate raspoređenih kroz dulje vrijeme.
- Olakšani pristup … Moguće je pristupiti aplikaciji s bilo koje patforme u bilo koje vrijeme. Istu radnu okolinu vidimo s različitih računala ili mobitela.
- Automatsko ažuriranje … ne moramo se brinuti za ažuriranje aplikacije, tj. kada i kako ćemo instalirati novo izdanje.
- Manji troškovi upravljanja … Ne moramo zapošljavati lokalne administratore koji bi instalirali aplikaciju na lokalnim računalima.

Nedostatci od SaaS za korisnike

- Neadekvatno čuvanje osobnih podataka … Zemlje EU imaju zakone o osobnim podatcima koji se možda ne primjenjuju zemlji gdje je fizički smještena usluga.
- Gubitak kontrole nad ažuriranjem … Ne možemo spriječiti nadogradnju aplikacije koja ne odgovara našem načinu rada.
- Ograničenja izazvana korištenjem mreže … Ako aplikacija zahtijeva mnogo prijenosa podataka, vrijeme odziva može postati nezadovoljavajuće.
- Sigurnosna pitanja … Ako imamo povjerljive poslovne podatke, postoji rizik da proizvođač aplikacije neće dovoljno dobro zaštititi te podatke od zloupotrebe.
- Problemi s razmjenom podataka … Ako moramo razmjenjivati podatke između usluge i lokalnih aplikacija, to može biti komplicirano.

Arhitekture softvera u oblaku

- Jedna aplikacija obično služi većem broju korisnika. Svaki korisnik unutar aplikacije pohranjuje neke podatke.
- Ako je aplikacija realizirana kao usluga u oblaku (SaaS), tada se odgovarajući softver zajedno s podacima nalazi u oblaku.
- Postoje dvije arhitekture za takvu aplikaciju koje se razlikuju po načinu pohranjivanja podataka.
  - Sustav s više stanara (multi-tenant system).
  - Sustav s više instanci (multi-instance system).

Sustav s više stanara (1)

- Svi korisnici dijele jednu (višekorisničku) instancu sustava i služe se istom logičkom shemom za bazu podataka.
- Postoji samo jedna baza u koju svi upisuju podatke. Postoji jedinstvena sigurnosna kontrola.
  - Kažemo da u korisnici “stanari” u istoj multi-tenant bazi, svaki korisnik ima svoj “tenant-identifier”.
  - Podatci u bazi označeni su identifikatorom stanara koji ih je upisao.
  - DBMS se služi tim oznakama da bi ostvario “logičku izolaciju” stanara.
  - Dakle svaki stanar vidi samo svoje podatke i čini mu se da radi sa svojom vlastitom bazom.

Sustav s više stanara (2)

- Slijedi primjer za multi-tenant bazu.
  - U prvom stupcu nalazi se tenant identifier.
  - Stanar T516 vidi samo 1., 4. i 5. redak, bez identifikatora.
  - DBMS mora spriječiti “curenje podataka” (data leakage) od jednog korisnika prema drugom.

Tenant	Key	Item	Stock	Supplier	Ordered
T516	100	Widg 1	27	S13	2023/2/12
T632	100	Obj 1	5	S13	2023/1/11
T973	100	Thing 1	241	S13	2023/2/7
T516	110	Widg 2	14	S13	2023/2/2
T516	120	Widg 3	17	S13	2023/1/24
T973	100	Thing 2	132	S26	2023/2/12

Sustav s više stanara (3)

- Prednosti multi-tenant sustava.
  - Korištenje resursa … Proizvođač aplikacije ima kontrolu nad resursima i on može optimizirati softver tako da se resursi efikasno koriste.
  - Sigurnost … Mjere sigurnosti se pojednostavnjuju budući da imamo samo jednu instancu baze koju treba “patch”-ati ako se otkrije ranjivost.
  - Upravljanje promjenama … Lakše je ažurirati jednu instancu softvera nego mnogo instanci. Promjene stižu svim korisnicima u isto vrijeme. Svi korisnici uvijek rade s najnovijom verzijom softvera.

Sustav s više stanara (4)

- Nedostatci multi-tenant sustava.
  - Nefleksibilnost … Svi korisnici moraju koristiti istu logičku shemu baze podataka, uz eventualni ograničeni raspon za prilagodbu sheme.
  - Sigurnost … Postoji teorijska mogućnost da će podatci "curiti" od jednog korisnika prema drugome. No to se rijetko dešava. Češće dolazi do proboja sigurnosti koji pogađa sve korisnike zajedno.
  - Složenost … Sustavi s više stanara su složeniji od sustava s više instanci s obzirom da moramo upravljati s više korisnika odjednom. Zato postoji veća vjerojatnost za bugove u softveru koji upravlja bazom podataka.

Sustav s više instanci (1)

- Svaki korisnik ima svoju instancu sustava, dakle:
  - vlastitu kopiju softvera
  - vlastitu bazu podataka koja može imati specifičnu logičku shemu i specifični režim sigurnosti.
- Postoje dva načina realizacije:
  - VM-based multi-instance system … Kopija softvera i korisnikova baza podataka rade na posebnom virtualnom poslužitelju . Ima smisla onda kad je "korisnik" kompanija u čije ime radi više osoba.
  - Container-based multi-instance system … Svaki korisnik ima svoju izoliranu kopiju softvera i svoju bazu u posebnom kontejneru. Ima smisla onda kad je "korisnik" zapravo pojedinac i kad takvi pojedinci rade nezavisno te ne dijele nikakve podatke.

Sustav s više instanci (2)

- VM tehnologija i kontejneri mogu se i kombinirati. Dakle kompanija može:
  - Imati svoj “VM-based” sustav
  - Pokretati kontejnere nad tim VM sustavom za individualne korisnike (zaposlenike).
- Nedostatci multi-instance sustava.
  - Cijena … takvi sustavi su skupi jer zahtijevaju unajmljivanje mnogo virtualnih poslužitelja u oblaku i upravljanje s njima. Budući da se virtualni poslužitelji sporo pokreću, potrebno ih je držati stalno u aktivnom stanju čak i kad se malo koriste.
  - Upravljanje promjenama … održavanje je složeno budući da se ažuriranja moraju obavljati nad više instanci softvera koje možda nisu identične.

Sustav s više instanci (3)

- Prednosti multi-instance sustava.
  - Fleksibilnost … Svaka instanca softvera može se prilagoditi korisnikovim potrebama, Korisnici mogu imati različite logičke sheme za bazu podataka.
  - Sigurnost … Svaki korisnik ima svoju bazu podataka tako da ne postoji mogućnost “curenja” podataka od jednog korisnika do drugog.
  - Skalabilnost … Instance sustava mogu se skalirati u skladu s potrebama svojih korisnika. Npr. nekim korisnicima trebat će unajmiti jači poslužitelj nego drugima.
  - Otpornost …Ako dođe do greške u radu softvera, to će vjerojatno pogoditi samo jednog korisnika. Ostali mogu nastaviti raditi.

Odabir pogodne arhitekture (1)

- Odabir arhitekture za softver u oblaku (SaaS) najviše ovisi o zahtjevima na bazu podataka. Potrebno je odgovoriti na sljedeća pitanja:
  - Ciljani korisnici … Zahtijevaju li korisnici drukčije sheme za bazu podataka? Brinu li se korisnici za sigurnost podataka u slučaju korištenja zajedničke baze? Ako da, treba koristiti sustav s više instanci.
  - Transakcije … Je li nužno da sustav podržava ACID transakcije gdje se garantira konzistentnst podataka u svakom trenutku? Ako jeste, tada treba korisiti sustav s više stanara.
  - Veličina baze … Koliko je velika tipična korisnikova baza? Za velike baze obično je bolji sustav s više stanara gdje je lakše optimizirati performanse.

Odabir pogodne arhitekture (2)

- Potrebno je odgovoriti na sljedeća pitanja (nastavak):
  - Interoperabilnost baze … Žele li korisnici prebacivati podatke iz postojećih baza? Ako te postojeće baze imaju različite logičke sheme, treba koristiti sustav s više instanci.
  - Struktura sustava … Može li se sustav rastaviti na više izoliranih servisa? Može li se korisnikova baza razbiti u skup manjih baza gdje svaka služi jednom servisu? Ako je tako, tada treba koristiti kontejnersku realizaciju sustava s više instanci.

Utjecaj arhitekture na skalabilnost

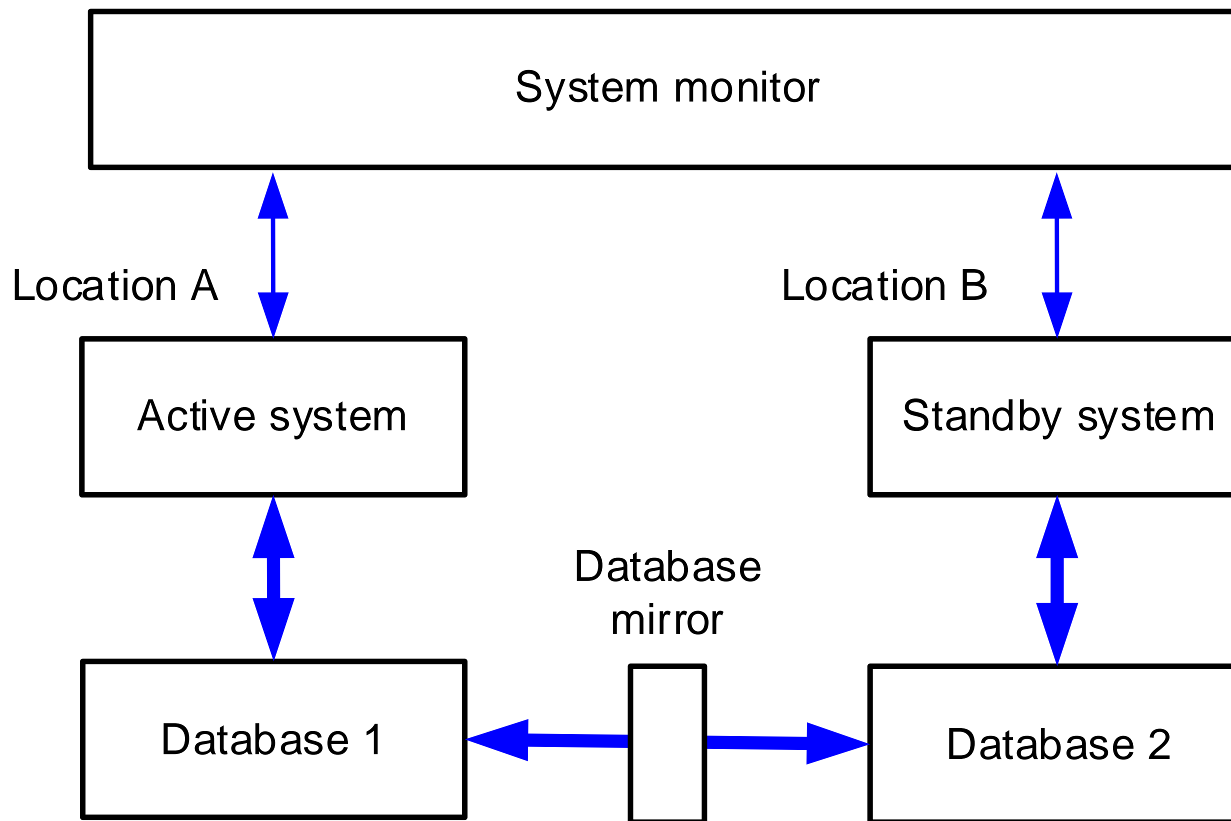
- Skalabilnost se općenito može postići dodavanjem novih virtualnih poslužitelja (scaling out) ili povećavanjem snage postojećih poslužitelja (scaling up).
- Kod softvera u oblaku, skalabilnost se lakše postiže ako smo odabrali arhitekturu s više instanci.
  - Dolaskom novih korisnika stvaraju se nove instance sustava, dakle automatski imamo scaling out.
  - Stvaranje novih instanci vrlo je efikasno ako su te instance implementirane kao kontejneri.

Utjecaj arhitekture na otpornost (1)

- Kod softvera u oblaku, otpornost se lakše postiže ako smo odabrali arhitekturu s više stanara.
  - Dakle trebala bi postojati samo jedna (aktivna) instanca sustava.
  - No zbog otpornosti dodajemo drugu (standby) instancu koja radi na drugoj fizičkoj lokaciji.
  - Sve promjene podataka “zrcale” se iz jedne baze u drugu.
  - Postoji “system monitor” koji stalno provjerava status aktivne instance te prebacuje uslugu na standby instancu ako se aktivna instanca sruši.

Utjecaj arhitekture na otpornost (2)

- Slika prikazuje organizaciju otpornog sustava s više stanara.



Autentikacija korisnika

- Ista aplikacija u oblaku daje prilagođene usluge raznim korisnicima. Zato aplikacija obavezno mora od korisnika tražiti da se predstave i dokažu svoj identitet. To se zove autentikacija.
- Tri mogućnosti kako da se obavi autentikacija:
  - Korištenjem vlastitog podsustava unutar aplikacije.
  - Oslanjanje na autentikaciju iz nekog drugog popularnog sustava (npr. Google, Facebook, LinkedIn, …).
  - Oslanjanjem na autentikacijski sustav organizacije u kojoj korisnici rade (npr. MS Office 365 na MO oslanja se na AAI).

Sadržaj Poglavlja 7

7.1. Softver u oblaku

7.2. Web servisi i mikro-servisi

7.3. DevOps

Pojam web servisa (1)

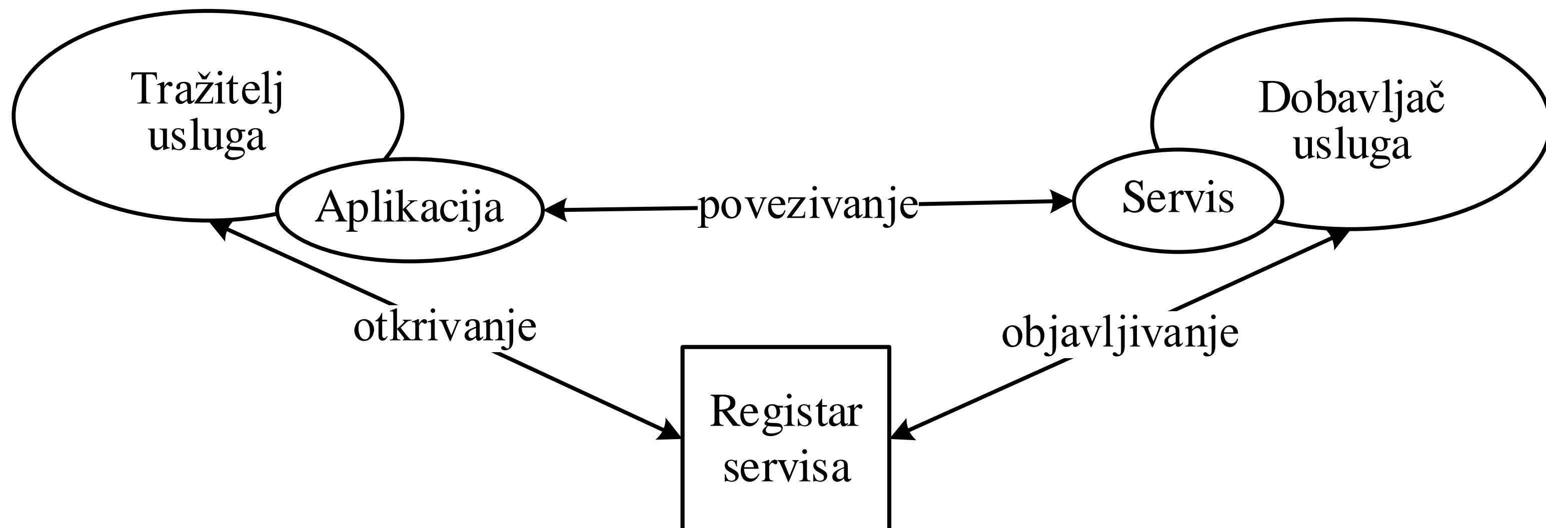
- Web servisi bili su zamišljeni kao nadogradnja klasične web tehnologije. Njome sadržaji s weba postaju dostupni i drugim programima, a ne samo web preglednicima.
- Web servis je resurs na jednom računalu kojem se može pristupiti s drugog računala preko Interneta.
  - To može biti podatkovni resurs, npr. katalog proizvoda.
  - Ili računalni resurs, npr. pretvarač slika iz jednog grafičkog formata u drugi,
  - Ili kombinacija jednog i drugog.

Pojam web servisa (2)

- Za zadani ulaz, servis proizvodi odgovarajući izlaz, bez ikakvih side-efekata.
- Servis je „stateless", tj. on ne pamti nikakvo lokalno stanje, kad ga pozovemo s istim ulazom, on daje isti izlaz.
- Servisu se pristupa preko njegovog javnog sučelja. Skriveni su detalji vezani uz njegovu implementaciju.
- Web servisi omogućuju specifičnu arhitekturu distribuiranih sustava: Service Oriented Architecture – SOA.

Service Oriented Architecture (1)

- Prema SOA, aplikacija se gradi povezivanjem samostalnih servisa dostupnih na Internetu.
- Postoje dvije vrste subjekata:
  – Tražitelj usluge (service requester)
  – Dobavljač usluge (service provider)



Service Oriented Architecture (2)

- Dobavljači usluga oblikuju i implementiraju servise te definiraju sučelja za pristup tim servisima.
- Također, dobavljači objavljuju informacije o svojim servisima u nekom registru.
- Tražitelji usluga otkrivaju u registru servise koji ih zanimaju i lociraju dobavljače usluga.
- Nakon toga tražitelji usluga mogu povezati svoju aplikaciju s odabranim servisom i komunicirati s njime preko njegovog sučelja, razmjenom poruka u skladu s predviđenim protokolom.

Svojstva tradicionalnih web servisa (1)

- Ideja web servisa pojavila se krajem 1990-ih godina. Uvidjelo se da je za efikasno korištenje servisa potrebno definirati
  - standardni protokol za komunikaciju
  - standardni format za opis sučelja.
- Nakon duljeg eksperimentiranja, početkom 2000-tih godina razvio se standard zasnovan na:
  - protokolu za komunikaciju SOAP (Service Oriented Architecture Protocol)
  - jeziku za opis sučelja WSDL (Web Service Definition Language).

Svojstva tradicionalnih web servisa (2)

- SOAP i WSDL temelje se na markup jeziku XML i na internetskom protokolu HTTP.
- Rad s ovakvim servisima zahtijevao je razmjenu i analiziranje velikih i složenih tekstova u XML-u.
- To je usporavalo rad aplikacija.
- Tražila se jednostavnija i efikasnija realizacija ideje web servisa.

Svojstva mikro-servisa (1)

- Mikro-servisi nastali su kao „perolaka” (lightweight) alternativa relativno glomaznim i sporim tradicionalnim web servisima.
- Osmislile su ih kompanije Amazon i Google.
- U manjoj mjeri služe za ponovnu upotrebu softvera. U većoj mjeri ih kompanije razvijaju interno, za potrebe svojih aplikacija.
- Umjesto niza srodnih usluga, mikro-servis obavlja samo jednu usko definiranu poslovnu funkciju (tzv. single responsibility).

Svojstva mikro-servisa (2)

- Umjesto oslanjanja na zajedničku bazu podataka, svaki mikro-servis koristi vlastite podatke.
- Umjesto zahtjevnih tvorevina kao što su SOAP i WSDL, mikro-servisi koriste:
  - „goli” HTTP protokol za komunikaciju
  - hijerarhijski građene URI za identifikaciju resursa (usluge).
- Za prikaz i razmjenu informacija, mikro-servisi preferiraju JSON format (JavaScript Object Notation), makar je i XML u upotrebi.

Svojstva mikro-servisa (3)

- Način interakcije između mikro-servisa nije do kraja standardiziran, no danas se uglavnom nastoje slijediti tzv. RESTful principi.
- Zbog svoje jednostavnosti i nezavisnosti u pogledu korištenja podataka, mikro-servisi su pogodni za implementaciju u kontejneru.
- Kad je mikro-servis implementiran u kontejneru, lagano ga se može replicirati, multiplicirati i premještati s poslužitelja na poslužitelj.
- Za aplikaciju koja koristi mikro-servise kaže se da ona ima „microservices architecture”.

Primjer za servise - autentikacija (1)

- Promatramo modul za autentikaciju koji podržava sljedeće značajke (features):
  - registracija korisnika, gdje korisnici daju podatke o svojem identitetu, broju mobitela, email adresi, …
  - autentikacija pomoću korisničkog imena i lozinke
  - dvo-faktorna autentikacija pomoću koda poslanog na mobitel
  - resetiranje lozinke.
- Svaka od ovih značajki mogla bi se realizirati kao jedan tradicionalni web servis.
  - Svi ti servisi koristili bi zajedničku bazu podataka.

Primjer za servise - autentikacija (2)

- No za realizaciju pomoću mikro-servisa, ove značajke su prevelike.
  - Da bismo identificirali mikro-servise, značajke se trebaju se razbiti u detaljnije funkcije.
  - Sljedeća slika prikazuje takve detaljnije funkcije za prve dvije značajke.

Registracija korisnika
Zadaj novo korisničko ime
Zadaj novu lozinku
Unesi podatke za oporavak lozinke
Potvrdi registraciju

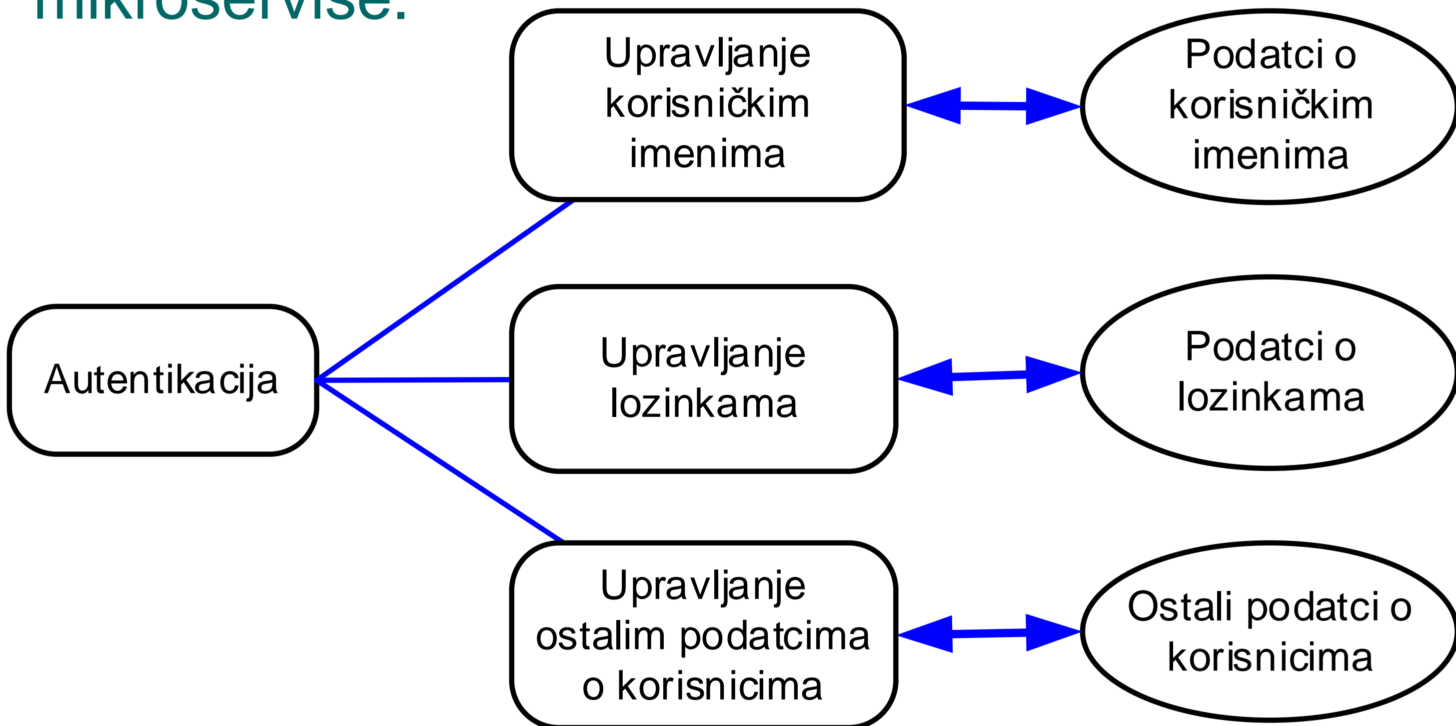
Autentikacija pomoću korisničkog imena i lozinke
Unesi korisničko ime
Unesi lozinku
Provjeri usklađenost imena i lozinke
Potvrdi autentikaciju

Primjer za servise - autentikacija (3)

- Jedna mogućnost je da se svaka funkcija sa slike implementira kao zasebni mikro-servis:
  - No tada bi svaki mikro-servis morao imati svoje podatke.
  - Dobiveni mikro-servisi bili bi previše specifični te bi se jedni te isti podatci morali replicirati.
  - Postoje načini da se podatci u mikro-servisima održavaju u konzistentnom stanju, no to usporava sustav.
- Druga mogućnost je da gledamo podatke koji se koriste za autentikaciju te odredimo po jedan mikro-servis za svaku vrstu podataka.

Primjer za servise - autentikacija (4)

- Sljedeća slika prikazuje tako dobivene servise, ali samo za autentikaciju.
- Registracija korisnika zahtijevala bi dodatne mikroservise.



Primjer za servise – ispis fotografija (1)

- Razvijamo sustav za ispis fotografija s mobilnih uređaja.
  - Korisnici mogu „uploadati” fotografije sa svojeg mobitela na naš poslužitelj.
  - Ispis može biti u različitim veličinama i na različite medije (npr. šalica, majica).
  - Predmet s ispisanom fotografijom šalje se korisniku na kućnu adresu.
  - Korisnik plaća preko servisa za plaćanje poput Google Pay ili Apple Pay ili preko kreditne kartice.

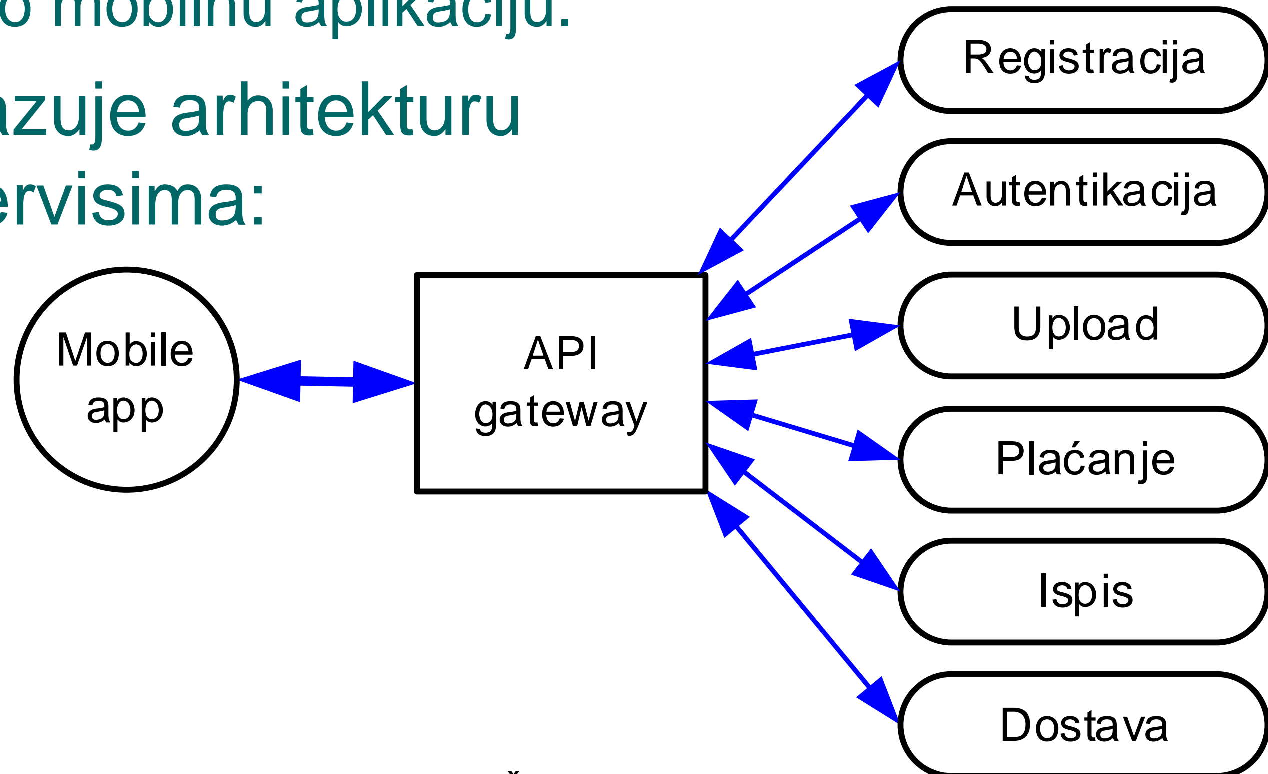
Primjer za servise – ispis fotografija (2)

- Zamišljeni sustav može se implementirati preko monolitne troredne arhitekture klijent-poslužitelj:
  - Mobilna aplikacija bila bi klijent.
  - Poslovna logika bila bi u srednjem sloju.
  - Treći sloj predstavljala bi zajednička baza podataka.
- Druga mogućnost je implementacija s mikro-servisima:
  - Postojao bi posebni mikro-servis za svaki dio funkcionalnosti.
  - Postojao bi „API gateway” koji izolira mobilnu aplikaciju od mikro-servisa i prevodi zahtjeve za uslugama u pozive prema mikro-servisima.

Primjer za servise – ispis fotografija (3)

- Mobilna aplikacija ne mora znati koji protokol se koristi za komunikaciju sa servisima.
- Moguće je promijeniti dekompoziciju u servise bez da mijenjamo mobilnu aplikaciju.

- Slika prikazuje arhitekturu s mikro-servisima:



Oblikovanje mikro-servisa

- Zadatak oblikovanja mikro-servisa sastoji se od sljedećih pod-zadataka:
  - dekomponirati zamišljeni složeni sustav u skup jednostavnih mikro-servisa
  - ostvariti komunikaciju između mikro-servisa
  - ostvariti usklađivanje vrijednosti podataka unutar različitih mikro-servisa
  - osigurati koordinaciju rada mikro-servisa
  - upravljati greškama u radu mikro-servisa.

Dekomponiranje u mikro-servise (1)

- Ne postoji egzaktna metoda. No postoje sljedeći naputci koji mogu biti korisni:
  1. Kao i kod oblikovanja klasa u OO sustavima, treba postići da svaki servis ima jaku unutarnju koheziju (high cohesion) i slabe veze prema van (low coupling).
  2. Treba uravnotežiti usko definiranu funkcionalnost pojedinih servisa i performanse cijelog sustava.
     - Sa stanovišta održavanja, dobro je da svaki servis ostvaruje samo jednu funkciju.
     - No razbijanjem sustava na veći broj vrlo specifičnih servisa nastat će potreba za većom komunikacijom između tih servisa. To degradira performanse.

Dekomponiranje u mikro-servise (2)

3. Elementi sustava za koje je vjerojatno da će se istovremeno mijenjati trebaju biti smješteni unutar istog servisa. Većina promjena zahtjeva pogodit će tada po jedan servis.
4. Servise treba staviti u vezu s „poslovnim sposobnostima”. Poslovna sposobnost je dio poslovne funkcionalnosti koji je u nadležnosti jedne osobe ili grupe.
   – Npr. u sustavu za ispis fotografija imamo grupu odgovornu za dostavu fotografija (dispatch capability) odnosno osobu za financije (payment capability).
5. Treba oblikovati servise tako da svaki od njih ima pristup samo onim podatcima koji su mu nužni.
   – Kad god imamo preklapanje između podataka u raznim servisima, morat ćemo uvesti i mehanizme za međusobno usklađivanje vrijednosti

Dekomponiranje u mikro-servise (3)

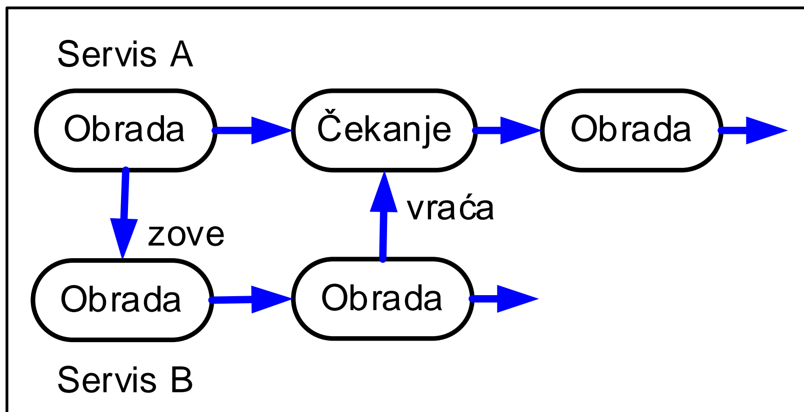
- Dobar početak za identificiranje mikro-servisa je analiza podataka kojima servisi trebaju upravljati. Obično ima smisla razvijati servise oko logički koherentnih podataka.
  - To smo radili u primjeru sustava za autentikaciju.
- Iskusni projektanti tvrde da je najbolje krenuti od monolitnog rješenja zasnovanog na više-rednoj arhitekturi klijenta i poslužitelja, te zatim rastaviti takvo rješenje u skup mikro-servisa.
  - Nakon što steknemo iskustvo u radu s monolitnom implementacijom i vidimo kako se koriste podatci, lakše je identificirati funkcionalnost koja se može zatvoriti u pojedini servis.

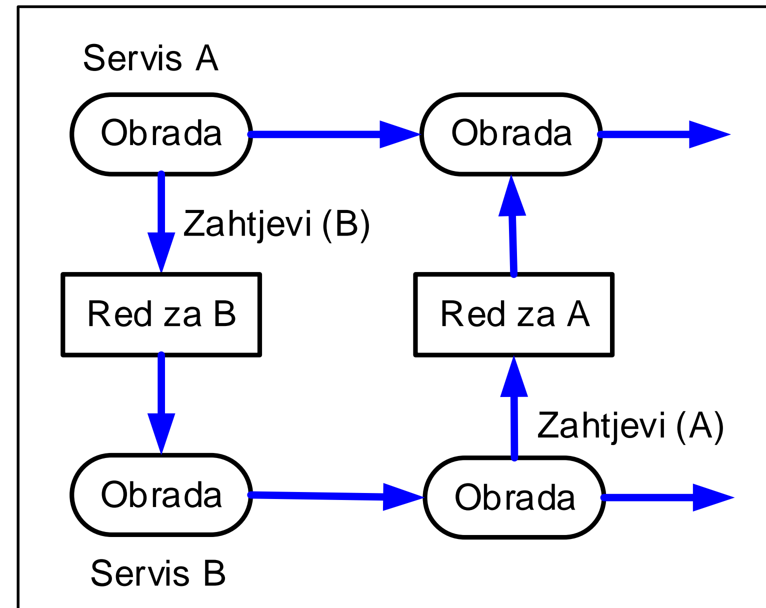
Komunikacija između mikro-servisa (1)

- Mikro-servisi komuniciraju tako da razmjenjuju poruke.
  - Jedna poruka predstavlja zahtjev za uslugom ili odgovor na prethodni zahtjev.
  - Poruka sadrži informaciju o pošiljatelju te ulazne podatke za traženu uslugu odnosno rezultate za izvršenu uslugu.
- Kad oblikujemo mikro-servise, moramo odabrati standard za komunikaciju koji će svi mikro-servisi slijediti. Glavne odluke koje trebamo donijeti su:
  - Treba li interakcija servisa biti sinkrona ili asinkrona?
  - Trebaju li servisi komunicirati izravno ili preko odgovarajućeg među-softvera (message broker-a)?
  - Koji protokol će se koristiti za razmjenu poruka?

Komunikacija između mikro-servisa (2)

- Sljedeće slike objašnjavaju razliku između sinkrone i asinkrone interakcije.
  – Sinkrona interakcija je jednostavnija te manje podložna bug-ovima.
  – Asinkrona interakcija je efikasnija jer servisi nisu besposleni dok čekaju odgovor.

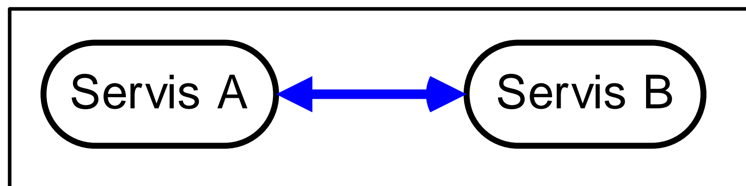




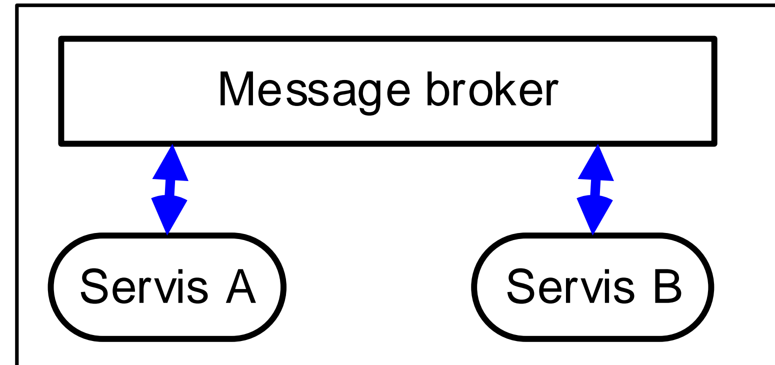
Komunikacija između mikro-servisa (3)

- Sljedeće slike objašnjavaju razliku između izravne i neizravne komunikacije (preko message broker-a).
  - Izravna komunikacija je obično brža, no ona zahtijeva da servisi znaju URI-je drugih servisa.
  - Neizravna komunikacija zahtijeva dodatni softver, message broker. Zahtjevi se šalju preko imena servisa umjesto URI-ja, a message broker pronalazi odgovarajući URI. To je korisno onda kad se URI mijenja ili kad isti servis postoji u više instanci.

Izravna komunikacija



Neizravna komunikacija



Komunikacija između mikro-servisa (4)

- Ako koristimo izravnu komunikaciju, tada sami možemo oblikovati protokol za razmjenu poruka. Uobičajeni pristup je da se slijede RESTful principi:
  - Poruke se prenašaju protokolom HTTP i sadrže ključne riječi GET, PUT, POST, DELETE.
  - Podatci unutar poruke prikazani su u formatu JSON.
  - Servis je predstavljen kao resurs koji ima vlastiti URI.
- Ako koristimo neizravnu komunikaciju, koristimo protokol definiran korištenim message broker-om.
  - Postoje već gotovi message broker-i koje možemo ugraditi u naš sustav, npr. RabbitMQ.
  - RabbitMQ podržava protokol AMQP (Advanced Message Queuing Protocol).

Usklađivanje vrijednosti podataka unutar mikro-servisa (1)

- Idealno bi bilo kad bi svaki mikro-servis upravljao svojim vlastitim podatcima.
- No u stvarnosti uvijek postoje preklapanja između podataka u raznim servisima. To znači sljedeće:
  - Treba nastojati da preklapanja budu što manja.
  - Treba nastojati da većina kopija istog podatka bude read-only, s minimalnim brojem servisa koji su odgovorni za ažuriranje.
  - Čim postoje preklapanja, nužno je uključiti mehanizam koji održava konzistenciju raznih kopija istog podatka.

Usklađivanje vrijednosti podataka…(2)

- Kod tradicionalnih sustava s višerednom arhitek-turom klijent-poslužitelj podatci su pohranjeni u zajedničkoj bazi podataka.
  - Tom bazom upravlja DBMS i on jamči da su podatci uvijek konzistentni.
  - Sve promjene baze odvijaju se kroz tzv. ACID transakcije koje DBMS izvršava na serijalizabilni način.
- Kod sustava s mikro-servisima mora se tolerirati određeni stupanj nekonzistentnosti podataka.
  - Podatci unutar različitih servisa ili replika istog servisa ne mogu biti sasvim konzistentni u svakom trenutku.
  - Ipak, moraju postojati mehanizmi koji osiguravaju da će se konzistentnost postići u “dogledno vrijeme”.

Usklađivanje vrijednosti podataka…(3)

- Postoje dvije vrste nekonzistentnosti koje treba držati pod kontrolom:
  1. Nekonzistentnost ovisnog podatka. Akcija ili neuspjeh jednog servisa može uzrokovati da podatak kojim upravlja drugi servis postane nekonzistentan
  2. Nekonzistentnost replika. Moguće je da se više replika istog servisa izvršava istovremeno. Svaka od njih mijenja svoju kopiju podataka. Mora postojati mehanizam koji će sve te kopije učiniti “eventualno konzistentnima”, tako da sve replike u “dogledno vrijeme” opet rade nad istim podatcima.

Usklađivanje vrijednosti podataka…(4)

- Kao ilustraciju za nekonzistentnost ovisnog podatka, promatramo primjer gdje korisnik naručuje knjigu. To pokreće nekoliko servisa:
  - servis koji upravlja skladištem i koji smanjuje broj primjeraka knjige na skladištu za 1
  - servis koji upravlja narudžbom i koji stavlja narudžbu u red onih koje treba ispuniti.
- Ova dva servisa su ovisni budući da neuspjeh drugog znači da će broj primjeraka knjige na skladištu biti netočan. Da bi upravljali situacijom:
  - trebamo otkriti neuspjeh drugog servisa
  - pokrećemo “kompenzirajuću transakciju” u prvom servisu koja povećava broj primjeraka knjige na skladištu za 1.

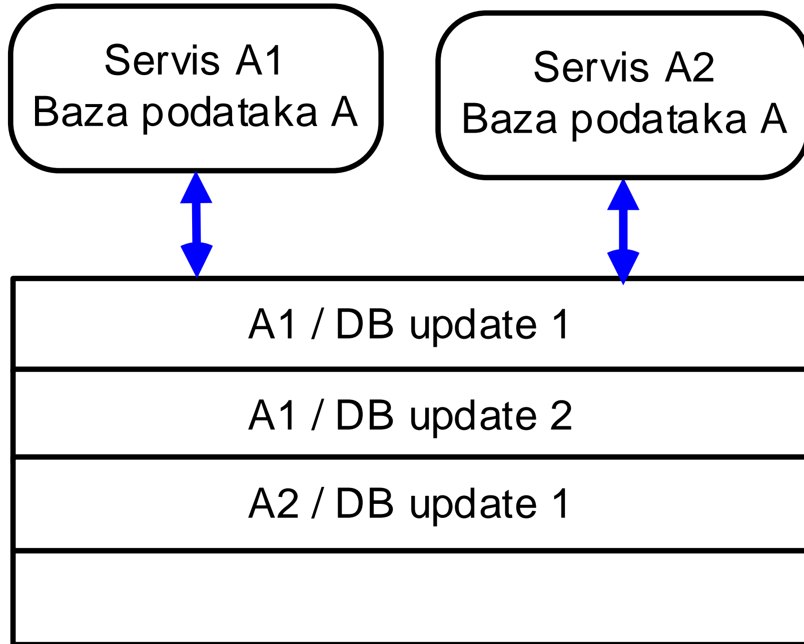
Usklađivanje vrijednosti podataka…(5)

- Doduše, kompenirajuće transakcije još uvijek ne daju potpunu garanciju da problema neće biti.
  - Npr. Ako je stanje na skladištu privremeno netočno i jednako 0, tada će neka nova narudžba koja se pojavila baš u tom trenutku biti odbijena makar je mogla biti ispunjena.
- Kao ilustraciju za nekonzistentnost replika, promatramo situaciju gdje dvije instance servisa za upravljanje skladištem (A i B) rade u isto vrijeme. Zamislimo sljedeći scenarij:
  - Servis A ažurira broj primjeraka knjige X na skladištu.
  - Servis B ažurira broj primjeraka knjige Y na skladištu.

Usklađivanje vrijednosti podataka…(6)

- Nakon toga, podatci unutar dviju instanci postaju nekonzistentni:
  - Servis A nema točno stanje za knjigu Y
  - Servis B nema točno stanje za knjigu X.
- Da bi ipak postigli "eventualnu konzistentnost", uvodimo žurnal transakcija (transactions log).
  - Kad jedna instanca servisa napravi promjenu podataka, ona to zapiše u žurnal.
  - Druge instance gledaju žurnal, ažuriraju svoje podatke i označavaju u žurnalu da su napravile promjenu.
  - Nakon što su sve instance obavile ažuriranje, transakcija se miče iz žurnala.

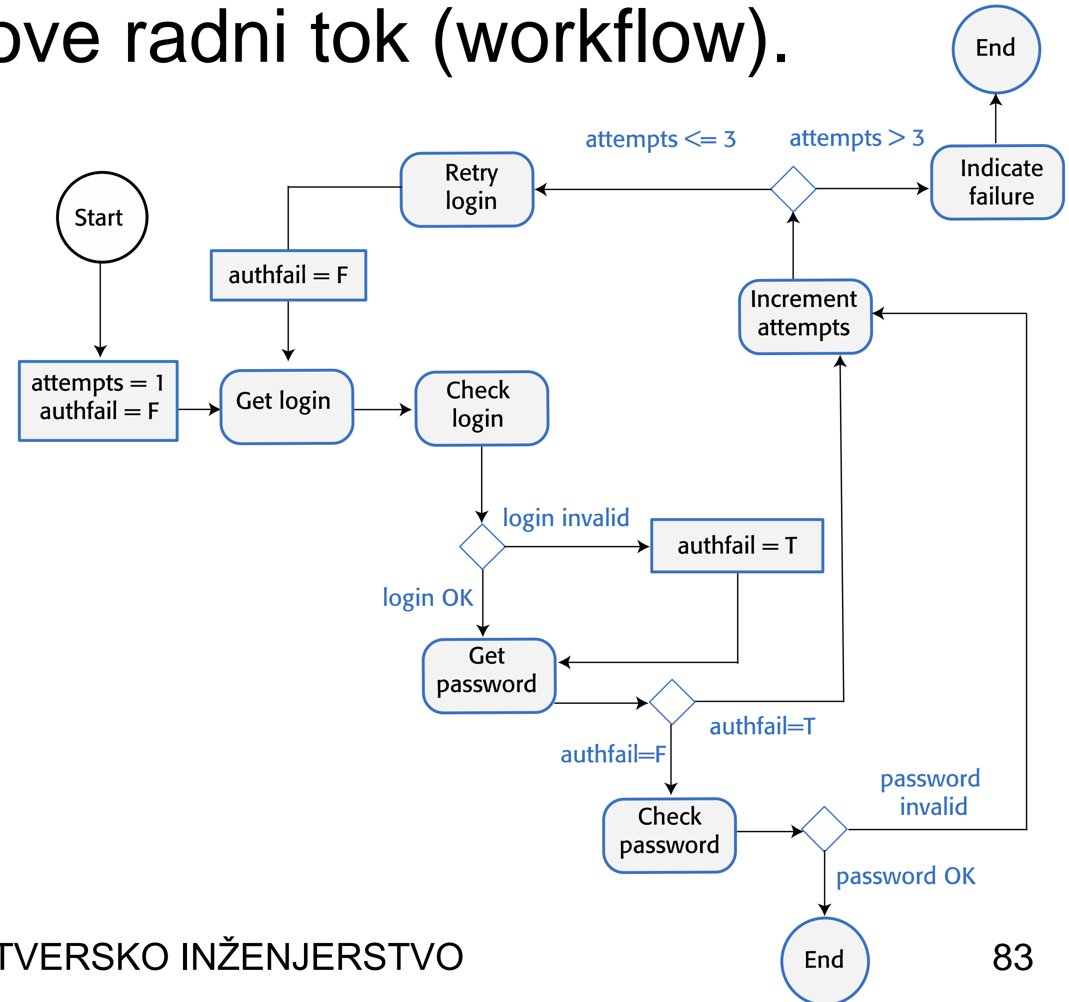
Usklađivanje vrijednosti podataka…(7)



- Instanca servisa gleda u žurnal svaki put kad počinje izvršavati novi zahtjev, da bi vidjela jesu li odgovarajući podatci bili ažurirani ili nisu.
- Ako jesu bili ažurirani, tada instanca najprije ažurira svoju kopiju podatka u skladu s žurnalom pa tek onda počinje sa svojom radnjom.
- Inače, instanca servisa može usklađivati svoje podatke s žurnalom onda kad nije opterećena zahtjevima.

Koordinacija rada mikro-servisa (1)

- Rad korisnika sa sustavom obično se sastoji od niza interakcija koje se moraju obaviti u nekom redolijedu. To se zove radni tok (workflow).
- Slika prikazuje radni tok za autentikaciju. Korisnik treba upisati korisničko ime i lozinku te ne smije pogriješiti više od 3 puta.

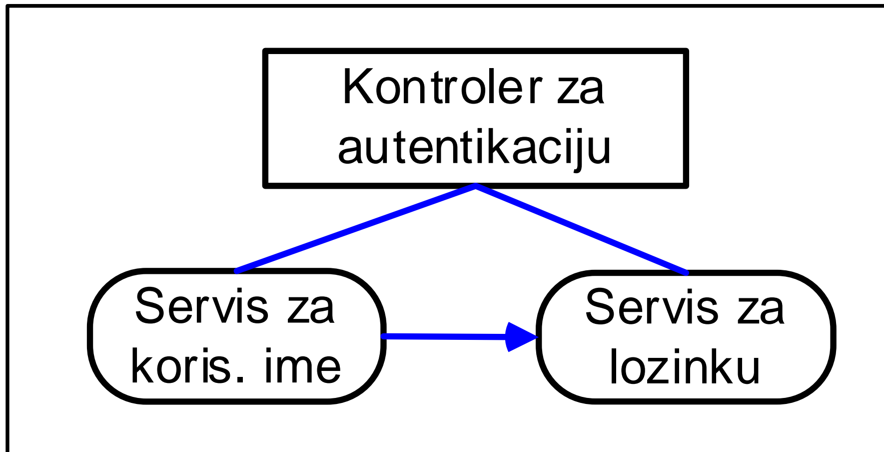


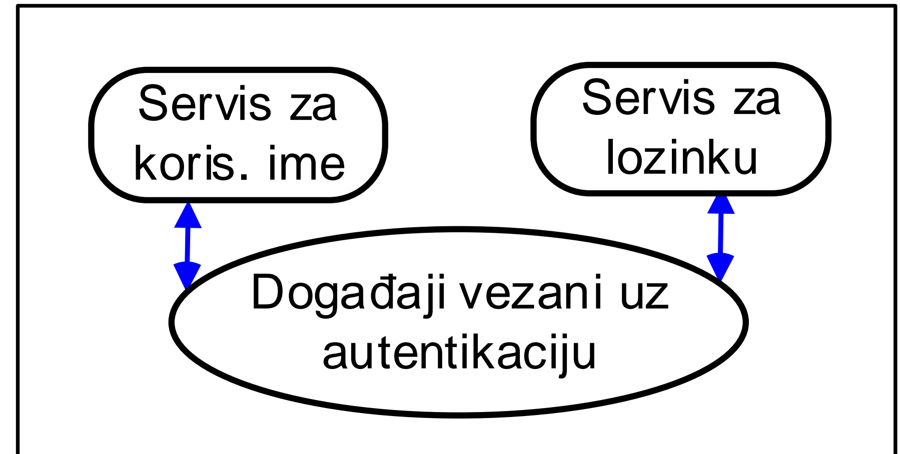
Koordinacija rada mikro-servisa (2)

- Svaka interakcija na prethodnoj slici može se implementirati kao posebni mikroservis. No kako koordinirati njihov rad? Postoje dva rješenja:
  - Orkestracija (analogija s orkestrom). Dodaje se još jedan (glavni) servis, tzv. service controller, koji implementira radni tok i poziva ostale servise u predviđenom redoslijedu.
  - Koreografija (analogija s baletom). Svaki servis publicira "događaj" kao znak da je obavio svoju radnju. Ostali servisi prate događaje na koje su pretplaćeni te reagiraju na njih. Ne postoji eksplicitni service controller, Umjesto toga, potreban je message broker koji podržava mehanizam pretplaćivanja i publiciranja.

Koordinacija rada mikro-servisa (3)

- Slika prikazuje razliku između orkestracije i koreografije.





Koordinacija rada mikro-servisa (4)

- Prednost orkestracije je da je lakša za implementirati i debugirati. U slučaju neuspjeha u radu servisa, service controller zna koji servis nije uspio.
- Prednost koreografije je da su dijelovi sustava labavije povezani (low coupling) te ih je lakše mijenjati.
- Nedostatak koreografije je da je u slučaju neuspjeha teže odrediti koji mikroservis nije uspio obaviti svoj zadatak. Potrebno je dodati service monitoring system.

Upravljanje greškama u radu mikro-servisa (1)

- Veliki sustavi pokreću tisuće instanci servisa u oblaku. Za očekivati je da će se pojaviti greške (neuspjesi, propusti) u radu nekih instanci.
- Zato takve sustave treba oblikovati tako da izlaze na kraj s greškama. Postoje tri tipa grešaka:
  - Interna greška servisa … otkriva je sam servis i dojavljuje je tražitelju usluge preko poruke o greški.
  - Eksterna greška servisa … Servis ne odgovara na zahtjeve i mora ga se ponovo startati.
  - Loše performanse … performanse servisa degradiraju se na neprihatljivu razinu, obično zbog prevelikog opterećenja. Potreban je scaling-out ili scaling-up.

Upravljanje greškama ... (2)

- U danas raširenim RESTful servisima, interne greške mogu se javljati preko HTTP status kodova.
  - Npr. ako se od servisa traži da pristupi nepostojećem URI, odgovarajući GET zahtjev treba vratiti status 404.
  - Štoviše, naši servisi mogu vraćati kodove koje sam HTTP ne koristi. Npr. kod 422 može značiti da je servis dobio podatak neispravnog tipa.
  - Kad oblikujemo naš sustav, važno je postići da svi servisi vraćaju isti status za istu vrstu greške.

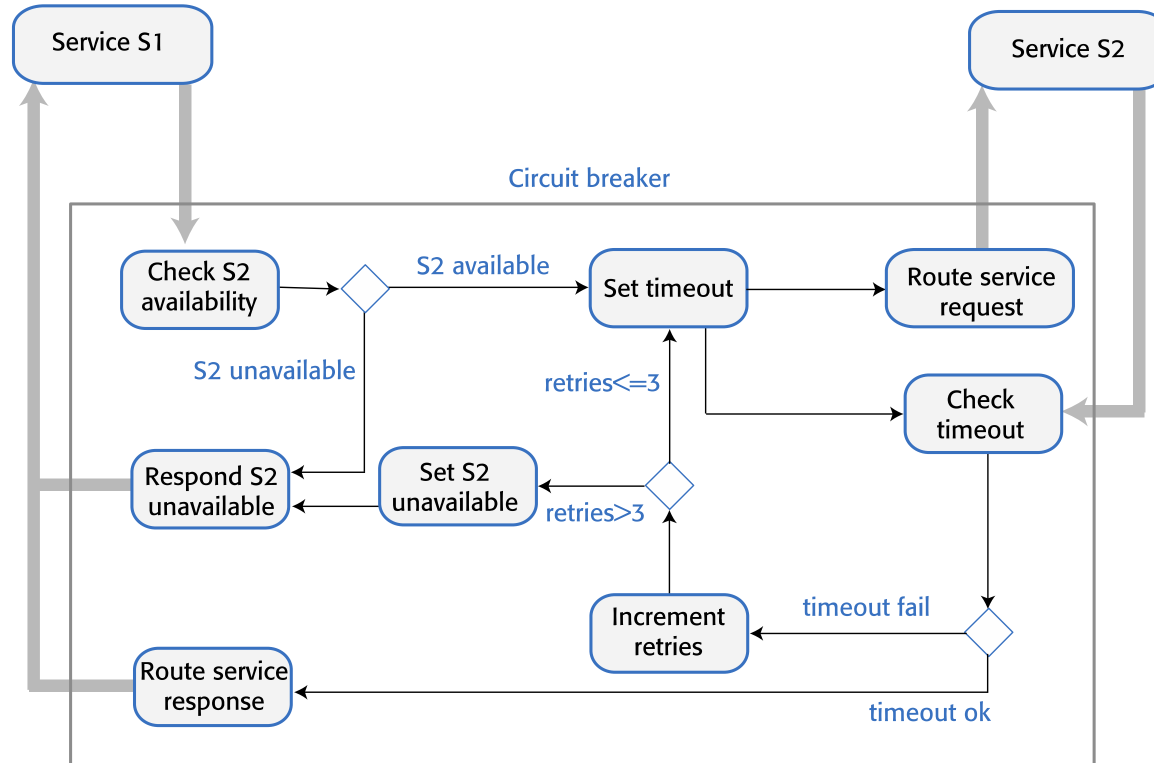
Upravljanje greškama … (3)

- Da bismo otkrili da neki mikro-servis ne radi ili radi presporo, možemo kod svakog zahtjeva za uslugu postaviti timeout.
  - Timeout je npr. brojač sekundi koji teče nakon što smo uputili zahtjev. Ako taj brojač dosegne neku unaprijed zadanu vrijednost, npr. 10 sekundi, pozivajući servis smatrat će da pozvani servis ne radi.
  - Nedostatak timeouta je da u slučaju nedostupnog servisa svaki pozivatelj mora čekati cijeli timeout, što usporava rad cijelog sustava.

Upravljanje greškama … (4)

- Spomenutom nedostatku timeout-a može se doskočiti uvođenjem „strujnog osigurača” (circuit breaker). Riječ je o posredniku između pozivatelja i pozvanog servisa koji radi ovako:
  - Inicijalno, osigurač kod svakog poziva primjenjuje timeout.
  - Kad kod jednog poziva timeout zaista istekne, osigurač „pregara”.
  - Daljnji pozivi odmah dobivaju signal da je servis nedostupan.
  - Pregoreni osigurač povremeno provjerava je li servis proradio. Ako je proradio, osigurač se resetira u inicijalno stanje.

Upravljanje greškama … (5)



- Strujni osigurač je primjer sustava za nadgledanje servisa (service monitoring system). Postoje i druge vrste monitora, npr. nadgledanje vremena odziva da bi se otkrila uska grla u sustavu.

Svojstva RESTful servisa (1)

- Mikro-servisi mogu komunicirati na sinkroni ili asinkroni način te koristiti razne formate i protokole za organizaciju i razmjenu poruka.
- No od tih različitih oblika interakcije servisa, danas se najviše koristi tzv. RESTful.
  - RESTful nije protokol u pravom smislu riječi, nego skup principa koji se mogu slijediti u većoj ili manjoj mjeri.
  - Kratica REST (Representional State Transfer) označava stil za arhitekturu, a za servise koji su usklađeni s tim stilom kaže se da su RESTful.
  - Osnovne postavke REST-a su:
    - Komunikacija preko (sinkronog) HTTP protokola
    - Identificiranje resursa pomoću URI (poopćenje URL)
    - Analogija s klasičnim world-wide-web sustavom.

Svojstva RESTful servisa (2)

- Detalnije, RESTful servisi trebali bi poštovati sljedeće principe:
  - Korištenje HTTP-ovih ključnih riječi … Da bi pristupili operacijama koje su dostupne u servisu, koristimo ključne riječi GET, PUT, POST, DELETE.
  - Servisi su "stateless"… Servis nikada ne smije pamtiti neko interno stanje. Za isti zahtjev servis uvijek daje isti odgovor.
  - Adresiranje preko URI … Svaki resurs ima svoj URI. Koristi se hijerarhijska struktura URI-ja da bi se pristupilo pod-resursima.
  - Korištenje JSON ili XML … Resursi se u pravilu prikazuju u JSON ili XML formatu ili oboje. Ako je potrebno, može se koristiti i audio ili video.

Svojstva RESTful servisa (3)

- Resursi su u pravilu podatkovne cjeline:
  - npr. podatci o nečijoj kreditnoj kartici, nečiji zdravstveni karton, katalog neke knjižnice, časopis, knjiga itd.
- Nad resursima se mogu obavljati četiri operacije koje se pridružuju standardnim HTTP riječima:
  - Create … implementira se pomoću HTTP POST, stvara resurs sa zadanim URI. Ako je resurs već stvoren, vraća se greška.
  - Read … implementira se pomoću HTTP GET, čita resurs i vraća njegovu vrijednost. Ne ažurira resurs, tako da uzastopne GET operacije bez umetnutih PUT operacija uvijek vraćaju isti rezultat.
  - Update … implementira se pomoću HTTP PUT, mijenja postojeći resurs. Ne smije se koristiti za stvaranje resursa.
  - Delete … implementira se pomoću HTTP DELETE, resurs sa zadanim URI postaje nedostupan. Resurs može ali i ne mora biti fizički uklonjen.

Primjer - prometni incidenti (1)

- Zamišljamo sustav koji održava informacije o incidentima na državnim cestama: gužve u prometu, radovi na cesti, prometne nezgode, …
- Sustav se nalazi na URL https://trafficinfo.net/incidents/
- Korisnici mogu postavljati upite sustavu da bi otkrili incidente na cestama po kojima planiraju putovati.
- Sustav je realiziran kao RESTful web servis. Resursi odgovaraju incidentima i organizirani su hijerarhijski.
- Incident se bilježi pomoću identifikatora ceste (npr. A90), lokacije (npr. Stonehaven), smjera vožnje (npr. Sjever) i rednog broja incidenta (npr. 1).
- Pojedinom incident pristupa se preko njegovog URI, npr: https://trafficinfo.net/incidents/A90/stonehaven/north/1

Primjer - prometni incidenti (2)

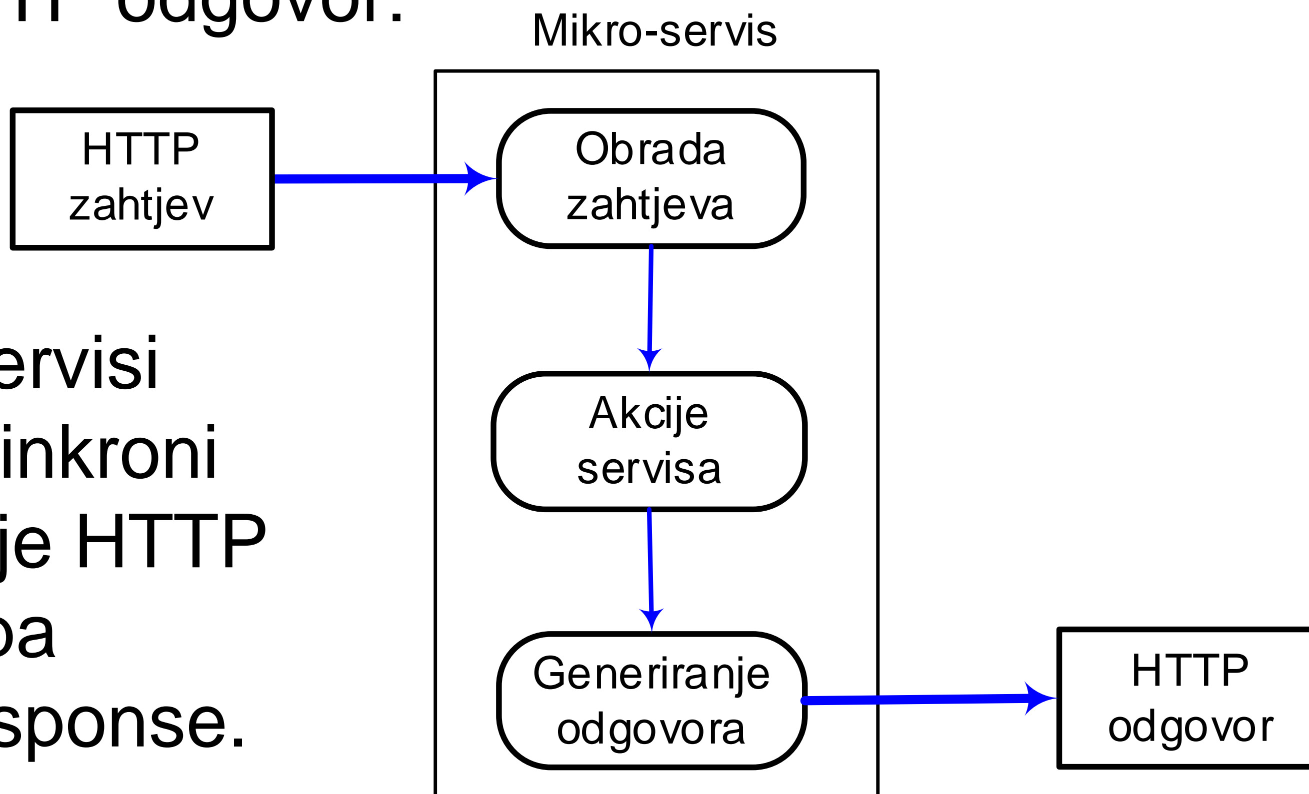
- Sam incident opisan je npr. Ovako:

> Incident ID: A90N17061714391
>
> Date: 17 June 2017
>
> Time reported: 1439
>
> Severity: Significant
>
> Description: Broken-down bus on north carriageway. One lane closed. Expect delays of up to 30 minutes.

- Naš servis podržava četiri operacije:
  1. Retrieve … vraća opis incidenta, implementira se pomoću GET.
  2. Add … dodaje informacije o novom incident, implementira se pomoću POST.
  3. Update … mijenja informacije o postojećem incident, implementira se pomoću PUT
  4. Delete … briše incident onda kad su njegove posljedice otklonjene, implementira se pomoću DELETE.

Način rada RESTful servisa (1)

- Općenito, RESTful mikro-servisi primaju HTTP zahtjeve u RESTful stilu, obrađuju te zahtjeve i vraćaju HTTP odgovor:



- RESTful servisi uvijek su sinkroni budući da je HTTP protokol tipa request/response.

Način rada RESTful servisa (2)

- Zahtjevi za RESTful servis i odgovori iz tog servisa strukturirani su kao HTTP poruke na sljedeći način:

REQUEST

<table>
<tr><td>[HTTP verb]</td><td>[URI]</td><td>[HTTP version]</td></tr>
<tr><td colspan="3">[Request header]</td></tr>
<tr><td colspan="3">[Request body]</td></tr>
</table>

RESPONSE

<table>
<tr><td>[HTTP version]</td><td>[Response code]</td></tr>
<tr><td colspan="2">[Response header]</td></tr>
<tr><td colspan="2">[Response body]</td></tr>
</table>

- Komponenta “header” u zahtjevu ili odgovoru sadrži meta-podatke o tijelu poruke te informaciju o poslužitelju gdje se servis nalazi.
- Komponenta “body” sadrži parametre zahtjeva odnosno odgovor. Obično je prikazana u JSON ili XML formatu.

Način rada RESTful servisa (3)

- Za mikro-servise posebno su važni sljedeći meta-podatci:
  - Accept … navodi tipove sadržaja koji se mogu obraditi zahtijevanim servisom te su prihvatljivi kao servisov odgovor. Najčešće korišteni tipovi su text/plain i text/json. Ovime se specificira da prihvatljivi odgovor može biti ili goli tekst ili JSON.
  - Content-Type … određuje tip sadržaja u tijelu zahtjeva ili tijelu odgovora. Npr. text/json znači da tijelo sadrži strukturirani JSON tekst.
  - Content-Length … određuje duljinu teksta u tijelu zahtjeva ili tijelu odgovora. Ako je 0, to znači da nema teksta u tijelu.

Primjer - prometni incidenti - (3)

- Ponovo gledamo sustav za evidenciju prometnih incidenata. Slijedi konkretan primjer GET zahtjeva i pripadnog odgovora:

REQUEST

GET	incidents/A90/stonehaven/	HTTP/1.1

```
Host: trafficinfo.net
...
Accept: text/json, text/xml, text/plain
Content-Length: 0
```

RESPONSE

HTTP/1.1	200

```
...
Content-Length: 461
Content-Type: text/json
```


{
  "number": "A90N17061714391",
  "date": "20170617",
  "time": "1437",
  "road_id": "A90",
  "place": "Stonehaven",
  "direction": "north",
  "severity": "significant",
  "description": "Broken-down bus on north carriageway. One lane closed. Expect delays of up to 30 minutes."
}
{
  "number": "A90S17061713001",
  "date": "20170617",
  "time": "1300",
  "road_id": "A90",
  "place": "Stonehaven",
  "direction": "south",
  "severity": "minor",
  "description": "Grass cutting on verge. Minor delays"
}

Primjer - prometni incidenti - (4)

- GET zahtjev nema tijelo poruke pa je odgovarajući Content-Length jednak 0. Općenito, GET zahtjev može imati tijelo ako želimo zadati neki selektor (filter) za sadržaj koji treba vratiti.
- URI naveden u GET zahtjevu nema ime poslužitelja. To ime je navedeno posebno. Ime poslužitelja obavezno se mora navesti kao parameter u zaglavlju zahtjeva.
- Tijelo odgovora sadrži opise za dva pronađena prometna incidenta. Ti incidenti opisani su u JSON formatu.
- Odgovor sadrži kod 200, što znači da je zahtjev uspješno bio obrađen. U slučaju kad ne bi bilo ni jednog prometnog incidenta (tj. kad bi odgovor bio prazan) vratio bi se kod 204.

Sadržaj Poglavlja 7

7.1. Softver u oblaku

7.2. Web servisi i mikro-servisi

7.3. DevOps

Raspoređivanje softvera (deployment)

- Nakon što je softver razvijen, on se mora izdati (release) i rasporediti u radnu okolinu (deploy). Tradicionalni način kako se to radi:
  - Softver je dostupan na proizvođačevim web stranicama ili u odgovarajućem "dućanu" poput Google Play, Microsoft Store ili App Store.
  - Korisnik mora izvršiti "download" softvera te ga instalirati na svoje računalo ili mobitel.
- No sve je više zastupljen i ovakav način:
  - Softver je dostupan korisnicima u izvršivom obliku, kao usluge u oblaku.
  - Korisnici odmah mogu koristiti softver, bez ikakvog download-a ili instalacije.

Podrška korisnicima (support)

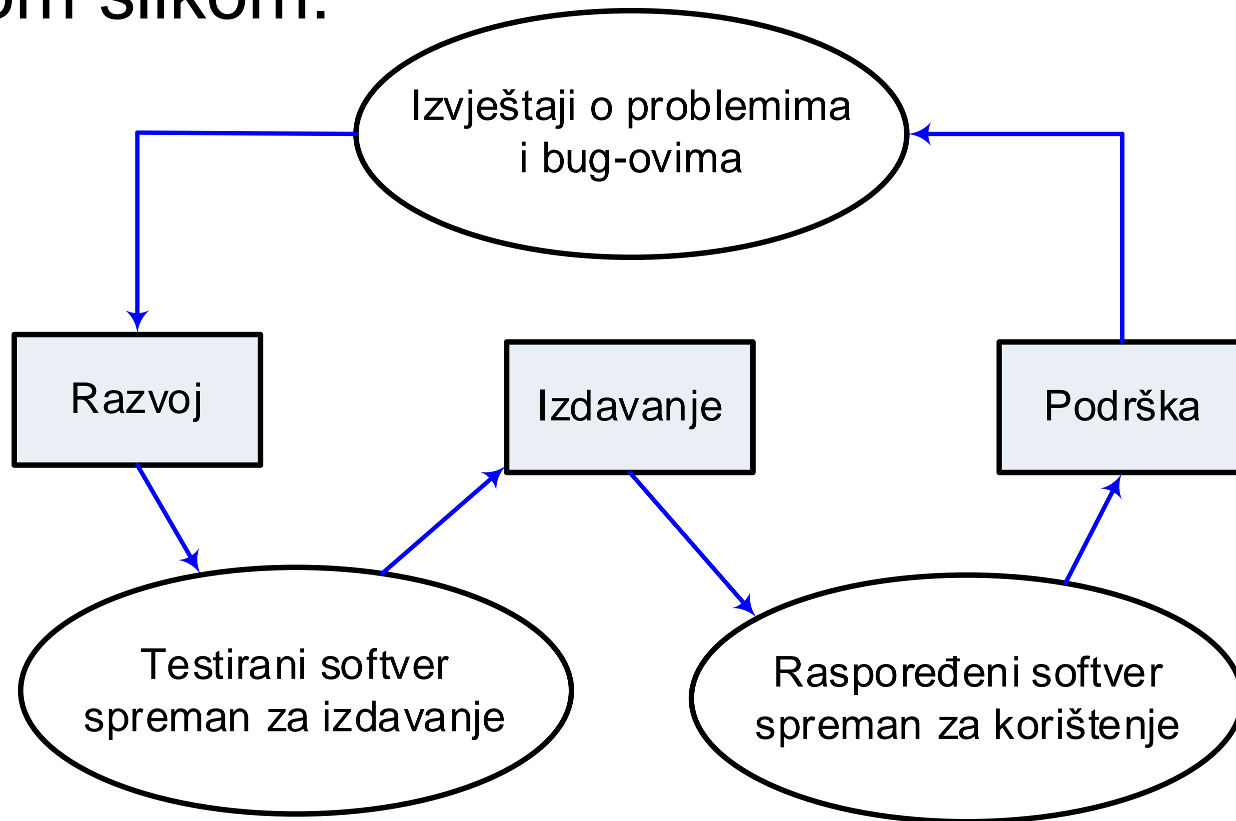
- Nakon što je softver izdan i raspoređen, proizvođač treba korisnicima pružiti podršku.
  - To može biti nešto sasvim skromno kao lista FAQ na web stranici.
  - Ili nešto mnogo intenzivnije, kao npr posebni “help desk” gdje se korisnici mogu obratiti u svakom trenutku.
- Kroz davanje podrške, proizvođač softvera također može skupljati i izvještaje korisnika o problemima u radu.
  - Takvi izvještaji omogućuju proizvođaču da odluči o promjenama koje će napraviti u novim izdanjima softvera.

Odnos između razvoja, izdavanja i podrške (1)

- Prema tradicionalnog podjeli posla, posebni timovi bili su zaduženi za razvoj (development), izdavanje (release) odnosno podršku (support).
  - Razvojni tim proslijedio je “konačnu” verziju softvera timu za izdavanje.
  - Tim za izdavanje sagradio je novo izdanje, testirao ga, pripremio prateću dokumentaciju te sve zajedno učinio dostupno korisnicima.
  - Treći tim pružao je podršku korisnicima.
  - Razvojni tim koji put je bio odgovoran i za naknadno održavanje softvera.
  - Ili je postojao četvrti tim zadužen samo za održavanje.

Odnos između razvoja, izdavanja i podrške (2)

- Tradicionalna podjela posla ilustrirana je sljedećom slikom:



Nedostatci tradicionalne podjele posla pri razvoju, izdavanju i podršci

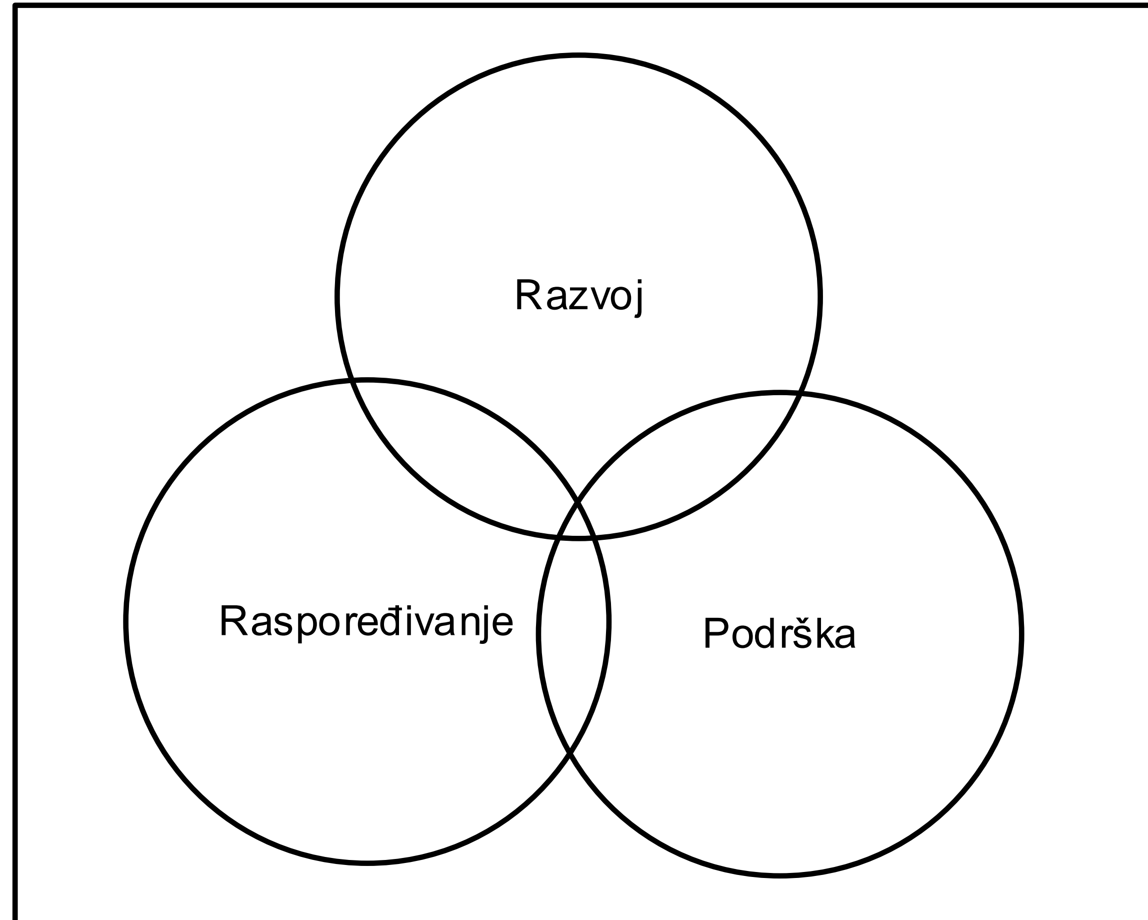
- Dolazi do komunikacijskih zastoja između timova.
- Razvojni tim koristi drukčiju radnu okolinu i alate nego ostali timovi.
- Ispravak kritičnih bug-ova ili sigurnosnih propusta traje danima prije nego što dođe do korisnika.

Pojam DevOps (1)

- Kratica znači “Development plus Operations”.
- Riječ je o novom (alternativnom) pristupu razvoju, izdavanju / raspoređivanju i podršci.
- Nastoje se otkloniti nedostatci tradicionalnog pristupa.
- Razne kompanije interpretiraju pojam DevOps na donekle različite načine, ovisno o svojim navikama te o vrsti softvera koji razvijaju.
- Ipak, osnovna ideja je:
  - da se sve spomenute aktivnosti (razvoj, izdavanje odnosno raspoređivanje, podrška) integriraju zajedno
  - da postoji jedinstveni tim koji je odgovoran za njih.

Pojam DevOps (2)

- Osnovna ideja DevOps-a ilustrirana je sljedećom slikom:



Faktori koji su doveli do nastanka i širokog prihvaćanja DevOps

- Agilne metode softverskog inženjerstva skratile su vrijeme razvoja softvera, no tradicionalni postupak izdavanja stvorio je usko grlo između razvoja i raspoređivanja. Pobornici agilnih metoda tražili su načine da se zaobiđe to usko grlo.
- Amazon je reorganizirao svoj softver tako da ga je pretvorio u servise, pa je onda uveo pravilo da za svaki pojedini servis treba postojati jedinstveni tim za razvoj i podršku. Amazon tvrdi da je to dovelo do značajnih poboljšanja. Tvrdnja je naišla na veliki odjek u struci.
- Postalo je moguće izdavanje softvera kao usluge u oblaku. Softverski proizvodi ne moraju se više izdavati na tradicionalan način.

Principi na kojima se zasniva DevOps

- Makar nema precizne definicije za DevOps, postoje tri osnovna principa:
  - Svatko je odgovoran za sve
    - Svi članovi tima imaju zajedničku odgovornost za razvoj softvera, njegovo izdavanje i raspoređivanje te podršku korisnicima.
  - Sve što se može automatizirati treba se automatizirati
    - Sve aktivnosti vezane uz testiranje, raspoređivanje odnosno podršku trebaju se automatizirati koliko god je moguće. Treba biti što manje manualnog rada i ljudskog angažmana.
  - Prvo mjeri, nakon toga mijenjaj
    - Process DevOps treba biti vođen mjerenjima, gdje se skupljaju podatci o softveru i njegovom radu. Skupljeni podatci služe kao podrška za odluke o promjenama procesa i alata u njemu.

Prednosti DevOps-a

- Kompanije koje su prihvatile DevOps tvrde da taj pristup donosi sljedeće prednosti:
  - Brže raspoređivanje … Softver može brže biti raspoređen zato što su smanjeni komunikacijski zastoji između ljudi koji su uključeni u proces.
  - Smanjeni rizik … Dodatak funkcionalnosti u svakom novom izdanju je sasvim mali. Time su smanjene mogućnosti za neželjene interakcije novih značajki sa starima.
  - Brži ispravci … Cijeli DevOps tim radi zajedno da bi ispravio problem u radu softvera te što prije vratio softver u upotrebu. Nije potrebno otkrivati koji tim je odgovoran za problem niti je potrebno čekati da taj tim riješi problem.
  - Veća produktivnost … DevOps timovi su sretniji i produktivniji od onih koji se bave razdvojenim aktivnostima. Sretni članovi tima manje su skloni napuštanju posla tj. promjeni radnog mjesta.

Poteškoće pri uvođenju DevOps-a

- Neke kompanije susrele su se sa sljedećim poteškoćama pri uvođenju DevOps-a:
  - Stvoriti DevOps tim znači skupiti zajedno ljude različitih profila, koji se bave razvojem softvera, računalnom infrastrukturom, sigurnošću, interakcijom s korisnicima. Nažalost, neki od tih ljudi smatraju svoj posao izazovnijim i važnijim od poslova drugih ljudi. To stvara napetosti u timu.
  - Uspješni DevOps tim mora razviti kulturu uzajamnog poštovanja i zajedništva. Neki članovi tima nisu skloni druženju s ostalima, dijeljenju svojih iskustava s njima, ili učenju novih vještina od njih.
  - Svi članovi DevOps tima moraju se osjećati odgovorni za rad softvera. No događa se npr. da softver prestane raditi preko vikenda. Većina razvojnih inženjera smatrat će da to nije njihova krivnja i neće prekidati svoj odmor. Radije će prebacivati krivnju na nekog drugog.

Automatizacija u DevOps – primjer mikro-servisa (1)

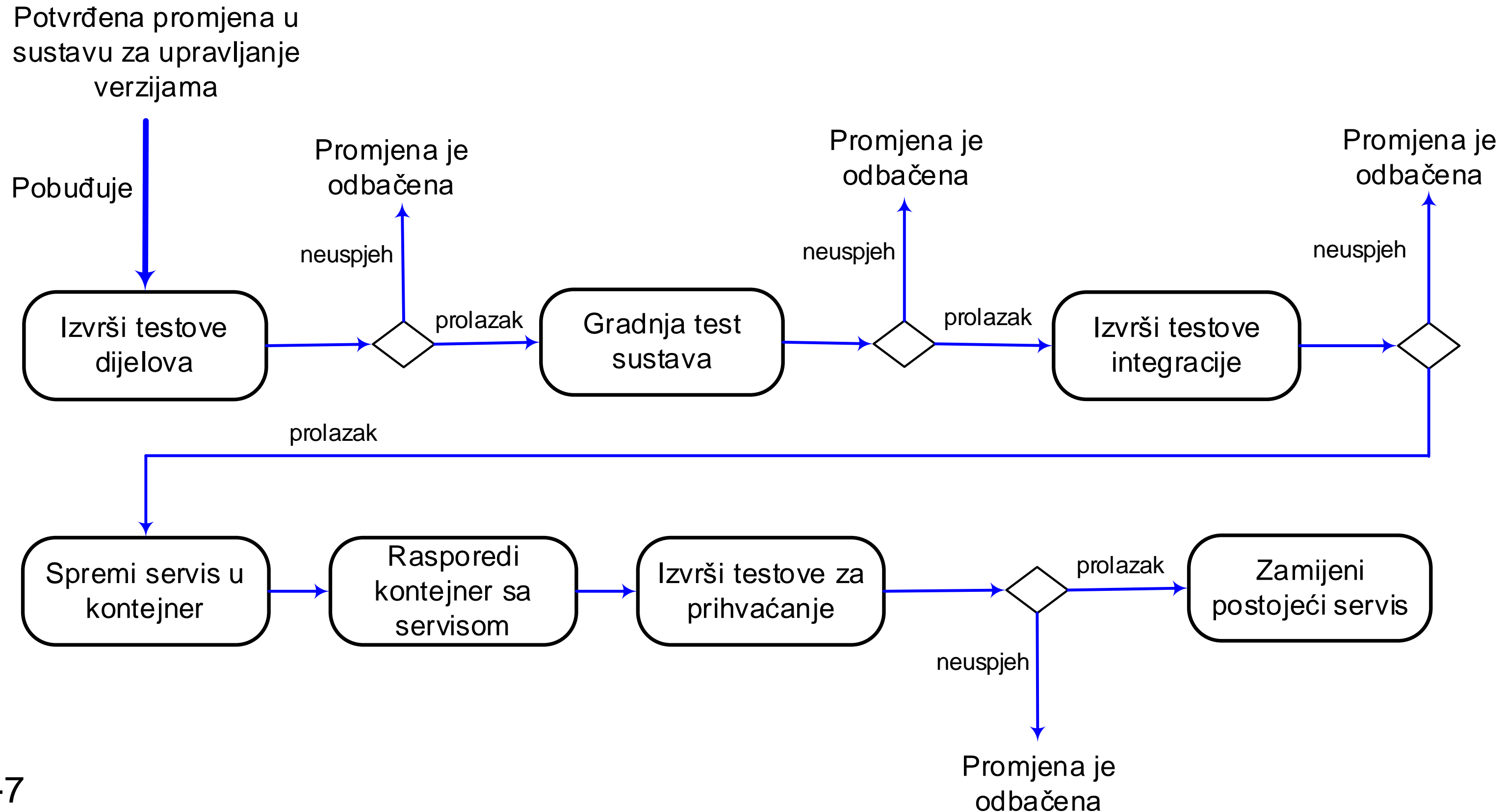
- DevOps je naročito pogodan za sustave građene od servisa koji su ponuđeni kao usluga u oblaku.
- Ako se koriste mikro-servisi, tada je danas uobičajena praksa da je tim koji razvija servis ujedno odgovoran i za raspoređivanje servisa.
- Isti tim odlučuje kad treba rasporediti novu verziju.
- Kao dobra praksa, pokazala se „politika kontinu-iranog raspoređivanja” (continuous deployment).
  - Čim je napravljena i testirana neka promjena na servisu, promijenjeni servis se odmah raspoređuje

Automatizacija u DevOps – primjer mikro-servisa (2)

- Da bi politika kontinuiranog raspoređivanja zaista bila moguća, nužna je automatizacija.
  - Čim je napravljena promjena u softveru (commit u repozitoriju koda), pokreće se niz automatskih testiranja.
  - Ako softver prođe testove, on ulazi u idući niz automatskih aktivnosti koje pakiraju softver u kontejner i stavljaju taj kontejner u upotrebu.

Automatizacija u DevOps – primjer mikro-servisa (3)

- Kontinuirano raspoređivanje, uz odgovarajuću automatizaciju, prikazano je na sljedećoj slici.



Automatizacija u DevOps – primjer mikro-servisa (4)

- Prednosti kontinuiranog raspoređivanja:
  - Smanjeni troškovi … Nakon početne investicije, troškovi su manji u odnosu na ručno raspoređivanje.
  - Brže rješavanje problema … Svako novo raspoređivanje donosi vrlo malu promjenu, pa je u slučaju pojave problema očito gdje se nalazi izvor problema.
  - Brža povratna informacija od korisnika … Čim je nova značajka implementirana, odmah je vidljiva korisnicima.
  - A/B testiranje … ako postoji više replika servisa, tada se nekim korisnicima može dati nova a nekima stara verzija servisa, pa se može mjeriti učinak promjene.

Mjerenje u DevOps (1)

- Proces DevOps treba stalno poboljšavati u cilju još bržeg raspoređivanja još kvalitetnijeg softvera.
- Poboljšanja se trebaju oslanjati na mjerenja koja spadaju u četiri kategorije.
  1. Mjerenje procesa … skupljamo i analiziramo podatke o razvoju softvera, testiranju i raspoređivanju.
  2. Mjerenje kvalitete usluge … skupljamo i analiziramo podatke o performansama softvera, pouzdanosti njegovog rada i prihvaćenosti kod korisnika.
  3. Mjerenje načina korištenja … skupljamo i analiziramo podatke o tome kako korisnici koriste naš softver.
  4. Mjerenje poslovne uspješnosti … skupljamo i analiziramo podatke o tome kako naš softver pridonosi ukupnom uspjehu softverske kuće u kojoj radimo.

Mjerenje u DevOps (2)

- Prve dvije kategorije (mjerenje procesa, mjerenje kvalitete usluge) su najvažnije za DevOps.
- Treća kategorija (mjerenje načina korištenja) pomaže da se identificiraju problemi u samom softveru.
- Četvrta kategorija (mjerenje poslovne uspješnosti) teško je mjerljiva i obično je nepouzdana.
- Da bismo obavljali mjerenja, potrebno je odabrati pogodne metrike koje se mogu lako mjeriti i analizirati.

Metrike u DevOps (1)

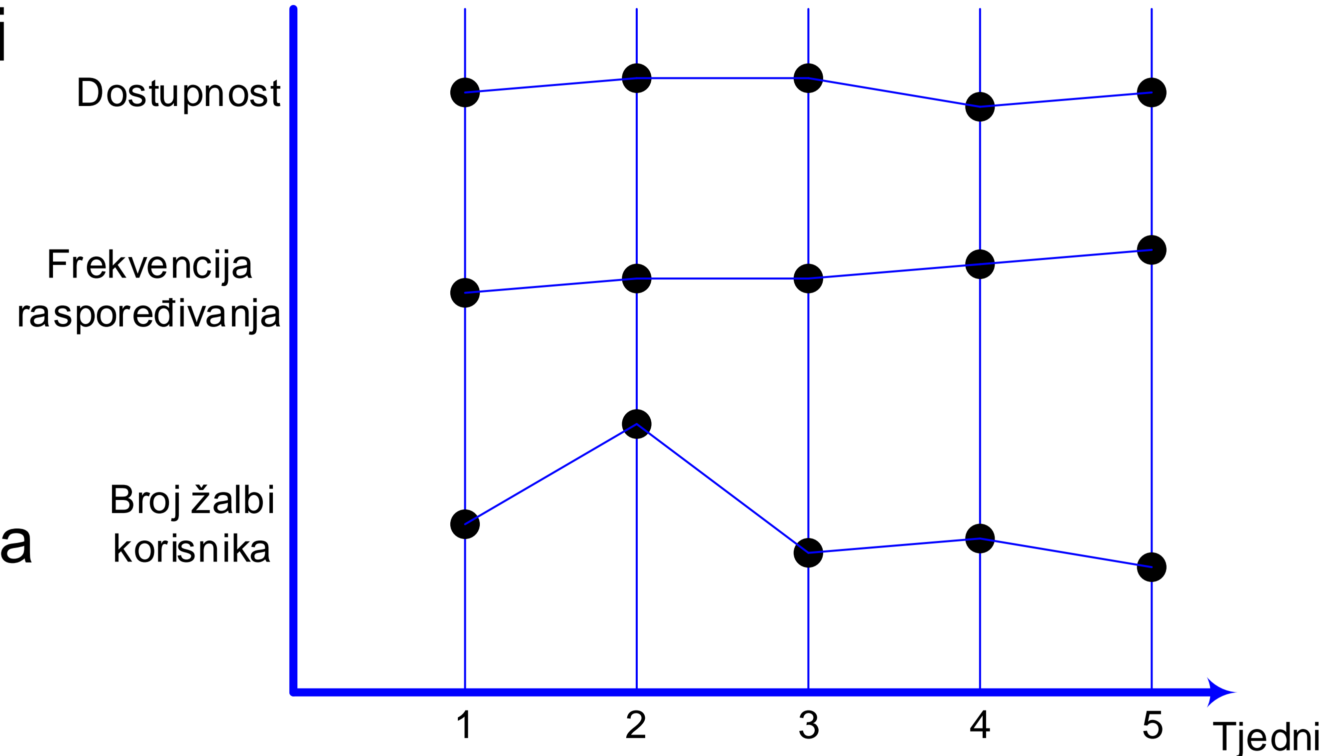
- Autorica Payal Chakravarty iz IBM-a preporučuje sljedećih 9 metrika.
  - Srednje vrijeme za oporavak nakon rušenja softvera
  - Postotak neuspješnih raspoređivanja novih izdanja
  - Frekvencija raspoređivanja novih izdanja
  - Opseg promjena (prosječni broj promijenjenih linija koda u novom izdanju)
  - Prosječno vrijeme koje prođe od razvoja novog izdanja do raspoređivanja tog izdanja
  - Performanse softvera
  - Dostupnost softvera
  - Broj žalbi korisnika u nekom vremenskom razdoblju
  - Postotak povećanja broja korisnika u nekom vremenskom razdoblju

Metrike u DevOps (2)

- Navedene metrike posebno su relevantne za softver koji se isporučuje kao usluga u oblaku.
- Prvih 5 metrika odnose se na mjerenje procesa.
  - Voljeli bismo vidjeti smanjenje:
    - vremena za oporavak
    - postotka neuspješnih raspoređivanja
    - vremena koje prođe od razvoja do raspoređivanja.
  - Voljeli bismo vidjeti povećanje:
    - frekvencije raspoređivanja
    - opsega promjena.
- Zadnje 4 metrike služe za mjerenje kvalitete usluge.
  - Voljeli bismo vidjeti:
    - da su performanse i dostupnost stabilne ili se poboljšavaju
    - da se broj žalbi korisnika smanjuje
    - da se broj novih korisnika povećava.

Metrike u DevOps (3)

- Chakravarty predlaže da se skupljeni podatci analiziraju na tjednoj bazi.
  - Prikazuje se tekući tjedan i nekoliko prethodnih tjedana.
  - Poželjni su grafički prikazi iz kojih se vide trendovi.
- Slika sadrži primjer analize trendova.
  - Vide se mala poboljšanja triju metrika.



Metrike u DevOps (4)

- Za skupljanje ovakvih podataka, potrebni su odgovarajući alati. Npr:
  - Vlasnici oblaka imaju “monitoring” softvere, npr. Cloudwatch od Amazona, koji skupljaju podatke o dostupnosti i performansama.
  - Podatci koje šalju korisnici (npr. žalbe) mogu se skupljati preko sučelja koje smo im mi osigurali u našoj aplikaciji.
  - Podatci o frekvenciji raspoređivanja može nam davati alat kojim smo automatizirali “continuous deployment”.

Mjerenje načina korištenja softvera

- Za mjerenje načina na koji korisnici koriste naš softver (3. kategorija mjerenja) koriste se log-datoteke.
  - Zapisi u log datoteci su zapisi događaja s vremenskim žigom koji opisuju akcije korisnika i/ili odgovore softvera.
  - Obično je riječ o stotinama događaja u sekundi, dakle o vrlo velikim količinama podataka.
- Log-datoteke potrebno je analizirati uz pomoć odgovarajućih alata.
  - Takvi alati dostupni su na Internetu.
  - Analiza treba iz velike količine zabilježenih podataka izvući korisnu informaciju.