Osijek, 2022.

ELPROS
REGIONALNI CENTAR KOMPETENTNOSTI
ELEKTROTEHNIČKA I PROMETNA ŠKOLA OSIJEK

Marko Harambaša
Josip Stanešić
Matija Kolarić

PRISTUP RAZVOJA OPERACIJA
DevOps

Priručnik

ESF UČINKOVITI LJUDSKI POTENCIJALI
EUROPSKI STRUKTURNI I INVESTICIJSKI FONDOVI

Marko Harambaša
Josip Stanešić
Matija Kolarić

Urednik: Silvio Papić

Naslov: Pristup razvoja operacija (DevOps)

Izdanje: 1. izdanje

Grafički urednik: Lucijana Dujić Rastić; Studio Utopia, Zagreb

Nakladnik: Elektrotehnička i prometna škola Osijek - RCK Elpros

Za nakladnika: Antun Kovačić

Mjesto i godina izdanja: Osijek, 2022.

CIP zapis dostupan u računalnom katalogu
Nacionalne i sveučilišne knjižnice u Zagrebu pod brojem XXXX

ISBN XXXX

PRISTUP RAZVOJA OPERACIJA (DevOps)

Priručnik

Marko Harambaša

Josip Stanešić

Matija Kolarić

Osijek, 2022.

Projekt je sufinancirala Europska unija iz Europskog socijalnog fonda.

Sadržaj:

1. Aplikativna rješenja

U OVOM POGLAVLJU NAUČIT ĆETE:

- razlikovati vrste *frontend* aplikacija
- razlikovati tehnologije izrade *frontend* aplikacija
- razlikovati vrste *backend* aplikacija
- razlikovati relacijske i nerelacijske baze podataka
- razlikovati vrste mobilnih i *standalone* aplikacija.

1.1. Osnovni principi frontend *web*-aplikacija

Frontend web-aplikacija (engl. *World Wide Web*) korisničko je sučelje *web*-stranice (engl. *web pages*) koje se nalazi na strani klijenta. *Web*-stranice hipertekstualni su dokumenti koji mogu sadržavati tekstualne, slikovne, video i zvučne sadržaje. *Frontend* omogućuje korisniku pregled sadržaja *web*-stranice i interakciju sa samom *web*-stranicom. Naime, sav vizualni prikaz koji korisnik može vidjeti smatra se korisničkim sučeljem. Usavršavanje korisničkog sučelja utjecalo je na brzi razvoj i upotrebu interneta, što je omogućilo i brži razvoj informacijsko-komunikacijskih tehnologija (IKT) generalno. Prije razvoja intuitivnih sučelja za prikaz i korištenja mrežnih stranica korisnici su se za pristup mrežnim stranicama služili komandnim sučeljem za izravnu interakciju s pozadinskim kôdom, što je činilo korištenje mrežnih stranica vrlo kompliciranim. Razvojem *frontend web*-aplikacija korisnicima je prikaz informacija na mrežnim stranicama postao intuitivniji i jednostavniji za korištenje, a samim time poboljšana je pretraga mrežnih stranica na internetu. U današnje vrijeme postoje različiti

programski jezici koji se koriste pri izradi korisničkih sučelja. Dogovoren je standard po kojem će bilo koji programski jezik za izradu korisničkog sučelja mrežne stranice biti preveden u tri osnovna programska jezika koje mogu interpretirati i koristiti svi internetski *web*-preglednici. Za izradu i oblikovanje korisničkog sučelja mrežne stranice koriste se programski jezici HTML (engl. *HyperText Markup Language*), CSS (engl. *Cascading Style Sheets*) i JavaScript. Programski jezik HTML koristi se za izradu i oblikovanje vizualnog izgleda i strukture mrežne stranice, CSS za uljepšavanje mrežne stranice, dok JavaScript krajnjem korisniku daje mogućnost interakcije s mrežnom stranicom. Poznavanje ovih programskih jezika nužno je znanje svakog razvojnog inženjera (engl. *developer*). HTML, CSS i JavaScript tri su glavna programska jezika koje internetski *web*-preglednici koriste za prikaz mrežnih stranica koje na dnevnoj bazi posjećujemo i koristimo.

1.2. Prezentacijski jezik HTML

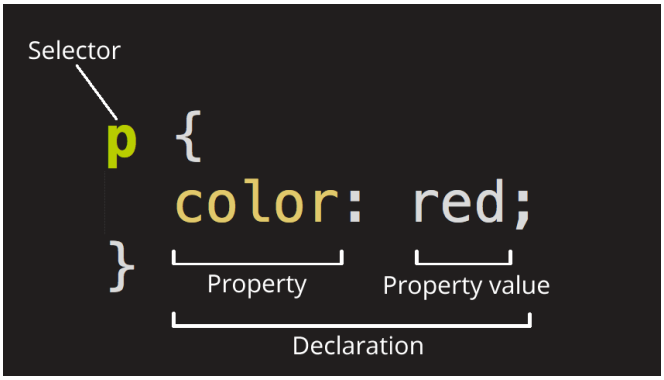
HTML je prezentacijski jezik koji se koristi za izradu i oblikovanje mrežnih stranica. Pomoću HTML-a izrađuju se osnovni gradivni blokovi koji čine mrežne stranice. Gradivni blokovi sastoje se od HTML oznaka (engl. *tag*) i atributa, koji definiraju strukturu mrežne stranice. Prikazani sadržaj mrežne stranice sastoji se od mnogo oznaka, s time da svaka oznaka ima svoje mjesto, stil i funkciju, dok se tekst, odnosno sadržaj stranice, nalazi unutar oznaka. Internetski *web*-preglednik koristi oznake kako bi na ispravan način mogao prikazati sadržaj koji se nalazi između oznaka, dok se same oznake ne prikazuju na mrežnoj stranici.

Slika 1.2.1. Primjer korištenja HTML oznaka

Preuzeto sa: https://linuxhint.com/html-basic-tags/

1.3. Stilski jezik CSS

CSS je stilski jezik koji se koristi za uređivanje, odnosno uljepšavanje izgleda mrežnih stranica. Razvojni inženjeri koriste CSS kako bi definirali npr. veličinu i stil teksta, veličinu tablica i fotografija te drugih elemenata koji se nalaze na samom sučelju mrežnih stranica ili *web*-aplikacija. HTML i CSS uvijek dolaze u paru. Kada se definiraju HTML oznake koje npr. koristi neka mrežna stranica, pomoću CSS stilova te se HTML oznake uređuju kako bi prikaz sadržaja bio ljepši i stilski prihvatljiviji. Glavna uloga koju ima CSS jest odvajanje logičke i stilske strukture mrežne stranice. CSS razvojnom inženjeru nudi mogućnost pisanja modularnog kôda, odnosno mogućnost da jedna mrežna stranica koristiti više CSS datoteka. Time je svaka CSS datoteka zadužena za stilski izgled pojedinog HTML dokumenta, što je u praksi jako korisno. Tako je na primjer kôd kompleksnih mrežnih stranica, gdje postoji veliki broj HTML dokumenata, poželjno da postoje i CSS dokumenti, koji definiraju izgled svakog pojedinog HTML dokumenta (koliko HTML dokumenata toliko i CSS dokumenata). Pri dodavanju novog stila potrebno je voditi računa o prethodnim stilovima jer se u suprotnom može „pregaziti" ili obrisati prethodni stil. Korištenjem CSS oznaka definira se koja HTML oznaka ima kakav stil.



Slika 1.3.1. Primjer korištenja CSS oznaka

Preuzeto sa: https://www.veritlabs.com/20-css-selectors-developers-know/

Na slici 1.3.1. može se vidjeti korištenje CSS-a za definiranje izgleda HTML oznake p koja se koristi na nekoj *web*-stranici. U ovom slučaju, gdje god se na *web*-stranici pojavi HTML oznaka „p", taj će tekst biti crvene boje. Isto možemo raditi i za bilo koje druge karakteristike, poput veličine slova, tipa fonta itd.

1.4. Skriptni jezik JavaScript

JavaScript je jedan od najpopularnijih skriptnih programskih jezika. Korištenjem JavaScripta mrežna stranica, koja bi inače bila samo skup statičnih elemenata, postaje interaktivna i dinamična. Naime, kada kažemo da je mrežna stranica interaktivna i dinamična, to znači da ona omogućuje dohvaćanje podataka s poslužitelja u realnom vremenu i prikazuje ih korisniku na sučelju (dok statična mrežna stranica znači da nema komunikacije s nekakvim poslužiteljem, već se svi podaci koji se prikazuju korisniku nalaze u samoj mrežnoj stranici). JavaScript ima vrlo značajnu funkciju u korištenju mrežnih stranica jer upravo JavaScript omogućuje ovu dinamiku i interaktivnost mrežnih stranica, što ih čini korisnijima i efikasnijima za korištenje. JavaScript programski jezik piše se unutar posebne HTML oznake **<script>**.

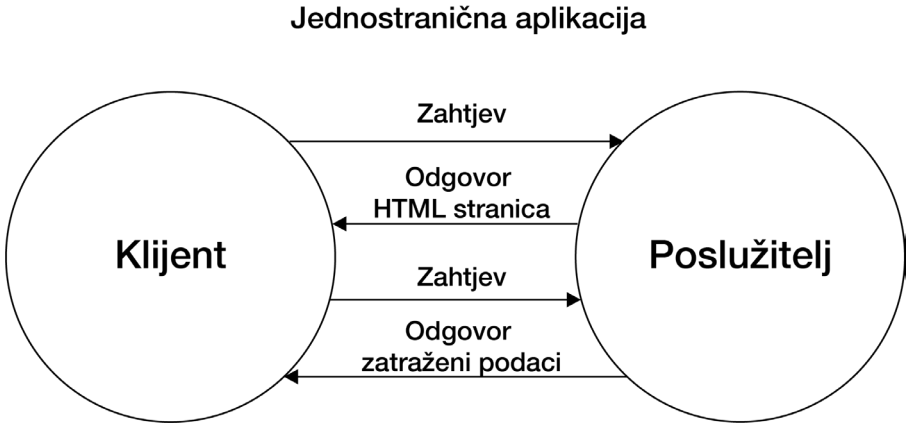
```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Hello Wolrd</title>
  </head>
  <body></body>
  <script>
    const message = "Hello World";
    console.log(message);
  </script>
</html>
```

Slika 1.4.1. Korištenje script taga

1.5. Jednostranične i višestranične aplikacije

Jednostranična *web*-aplikacija (engl. *Single Page Application – SPA*) vrsta je *frontend web*-aplikacije koja se koristi kada su mrežne stranice kompleksne, jer se tada mora paziti na performanse i brzinu *frontend web*-aplikacije. Kôd jednostranične *web*-aplikacije koristi se samo jedna HTML stranica, koja služi za prikaz svih korisničkih sučelja.

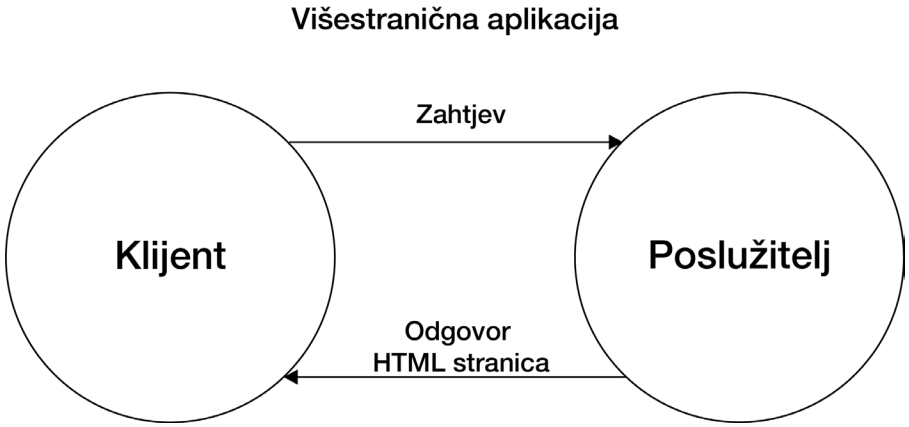
To znači da se većina resursa angažira samo jednom i to prilikom inicijalnog učitavanja mrežne stranice. Ovisno o zahtjevu koji je klijent uputio prema poslužitelju prikazat će mu se određeni podaci. Svakim sljedećim zahtjevom klijenta poslužitelj će dostaviti nove podatke. To znači da će se time ažurirati samo zatraženi podaci, dok će ostali podaci ostati neizmijenjeni. Ovim načinom na jednoj HTML stranici prikazuje se više podataka ovisno o zahtjevima klijenta.



Slika 1.5.1. SPA *frontend web*-aplikacija

Na slici 1.5.1. prikazan je ciklus odvijanja zahtjeva između klijenta i poslužitelja kad je u pitanju SPA vrsta *frontend web*-aplikacije. Usporedbom MPA i SPA *frontend web*-aplikacija vidi se razlika u zahtjevima i odgovorima koji se izmjenjuju između klijenta i poslužitelja. Jedan ciklus MPA aplikacija završava prilikom dobivanja odgovora od poslužitelja, dok ciklus SPA aplikacije traje i ovisi o radu korisnika koji koristi korisničko sučelje. SPA aplikacija koristi AJAX (engl. *Asynchronous JavaScript*) format za slanje zahtjeva prema poslužitelju, a kao odgovor poslužitelj vraća podatke u obliku JSON (engl. *JavaScript Object Notation*) formata, koje klijent prihvaća i prikazuje na određenom mjestu na HTML stranici.

Za razliku od jednostraničnih *web*-aplikacija, višestranične *web*-aplikacije (engl. *MultiPage Application – MPA*) vrsta su *frontend web*-aplikacija koje se koriste kada su mrežne stranice jednostavne i nemaju puno funkcionalnosti. Višestranična *web*-aplikacija radi tako da se za svaki zahtjev upućen prema poslužitelju dohvaća nova HTML mrežna stranica koja se onda prikazuje korisniku.



Slika 1.5.2. MPA frontend web-aplikacija

Na slici 1.5.2. prikazana je komunikacija između klijenta i poslužitelja prema principu višestranične *web*-aplikacije. Poslužitelj posjeduje upute za pronalaženje sadržaja da bi uz pomoć tih uputa mogao udovoljiti zahtjevima koje mu je uputio klijent. Na poslužitelju se nalaze potrebne metode za dohvaćanje podataka i generiranje sadržaja. Istovremeno klijent je zadužen za dohvat i prikaz sadržaja koji mu upućuje poslužitelj. Na svaki zahtjev klijenta dohvaća se nova HTML stranica.

SPA *frontend web*-aplikacija radi samo jedan zahtjev za dohvat HTML stranice koja se prikazuje na korisničkom sučelju. Ovisno o korisnikovim potrebama podaci se postepeno dohvaćaju s poslužitelja te se ažuriraju samo zatraženi podaci, a ne cijela HTML stranica, dok se kod MPA *frontend web*-aplikacije za svaki zahtjev dohvaća nova HTML stranica, što znači da kada korisnik ima potrebu za prikaz nekih drugih podataka, radit će se novi zahtjev, koji će kao odgovor vratiti novu HTML stranicu sa zatraženim podacima.

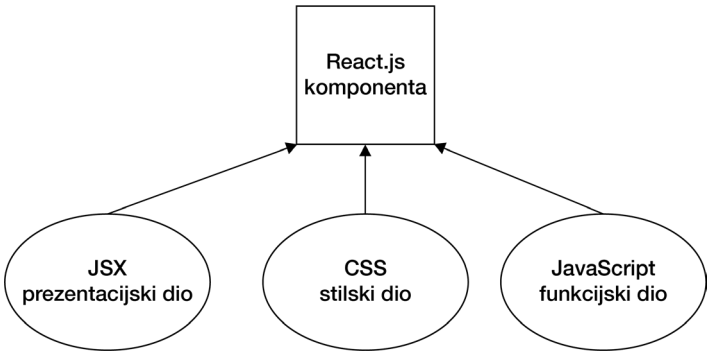
1.6. Uvod u JavaScript razvojne okvire

Zbog složenosti izrada *frontend web*-aplikacija zahtijeva posebna znanja i stručnost kako bi se postigla najbolja kvaliteta i skalabilnost. Pri tome značajnu ulogu imaju JavaScript razvojni okviri. JavaScript razvojni okvir skup je JavaScript biblioteka napisanih od strane mnogih razvojnih inženjera, koji nam olakšavaju izradu kompleksnih *frontend web*-aplikacija. Razvojni okvir omogućuje korištenje ugrađenih pre-

definiranih funkcionalnosti koje se obično koriste kako bi se ubrzao proces izrade *frontend web*-aplikacije. Na današnjem tržištu neki od najpopularnijih JavaScript razvojnih okvira za izradu *frontend web*-aplikacija jesu React.js, Angular i Vue.js.

1.7. Razvojni okvir React.js

Kreator React.js je Jordan Walke, razvojni inženjer Facebooka. React.js baziran je na izradi korisničkih sučelja u obliku komponenta, gdje se svaka komponenta sastoji od funkcijskog, prezentacijskog i stilskog dijela.

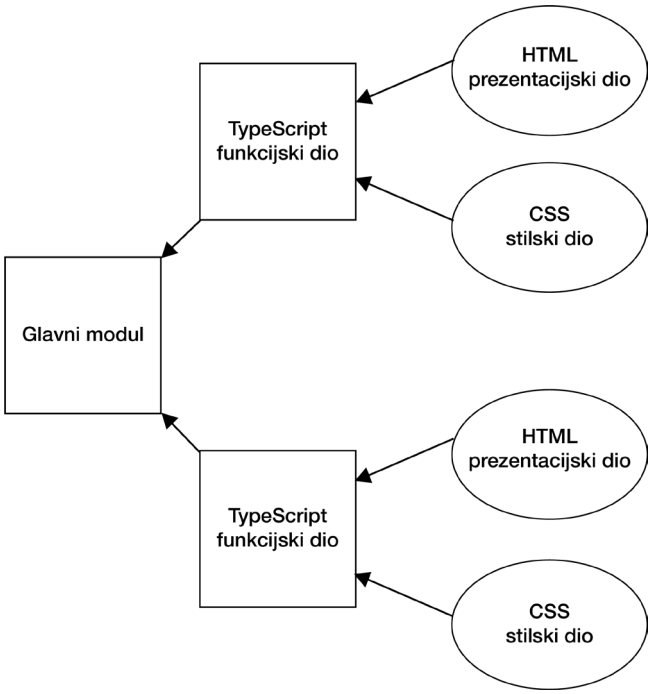


Slika 1.7.1. Elementi React.js komponente

React.js omogućuje pisanje HTML elemenata unutar JavaScript kôda korištenjem sintaksnog jezika JSX (engl. *JavaScript XML*). Da bi se *frontend web*-aplikacija dinamično ažurirala, na primjer klikom na gumb kojim mijenjamo tekst na mrežnoj stranici, koristimo stvarni DOM (engl. *Document Object Model*). Stvarni DOM hijerarhijski je prikaz elemenata mrežne stranice korištenjem JavaScript objekata. Koristi ga JavaScript programski jezik kako bi se dinamično ažurirali elementi na mrežnoj stranici. Ažuriranje elemenata u stvarnom DOM-u sporo je. Tomu je tako jer kada ažuriramo jedan element u stvarnom DOM-u, ažurira se cijeli stvarni DOM i ponovno se prikazuju svi elementi na mrežnoj stranici. Kako bi se ubrzao način ažuriranja stvarnog DOM-a, koristi se virtualni DOM, koji ujedno koristi React.js. Virtualni DOM kopija je stvarnog DOM-a. Kada se element ažurira u virtualnom DOM-u, radi se preslika samo ažuriranog elementa u stvarni DOM. Time se samo ažurirani elementi ponovno prikazuju na mrežnoj stranici, a ostali, neažurirani elementi ostaju nepromijenjeni. Ovim načinom ubrzan je proces ažuriranja elemenata na mrežnoj stranici.

1.8. Razvojni okvir Angular

Angular je JavaScript razvojni okvir koji je razvio Google. Napisan je u programskom jeziku TypeScript. TypeScript nije jezik koji internetski preglednici mogu razumjeti, stoga se TypeScript prevodi u JavaScript kako bi svaki internetski preglednik na ispravan način mogao razumjeti TypeScript kôd. Korištenjem TypeScripta ostvarena je tipizacija varijabli kako bi se lakše detektirale logičke greške u kôdu. Tipizacija je skup tipova vrijednosti koji se pohranjuju u varijable. Tipovi kao što su na primjer: znakovni niz, znak, broj ili decimalan broj. Jedna od značajnih vrijednosti TypeScripta jest to da podržava objektno orijentirano programiranje, čime je ubrzan razvoj *web*-aplikacija. Angular koristi CSS, HTML i TypeScript, gdje CSS predstavlja stilski dio, HTML prezentacijski dio, a TypeScript funkcijski dio.

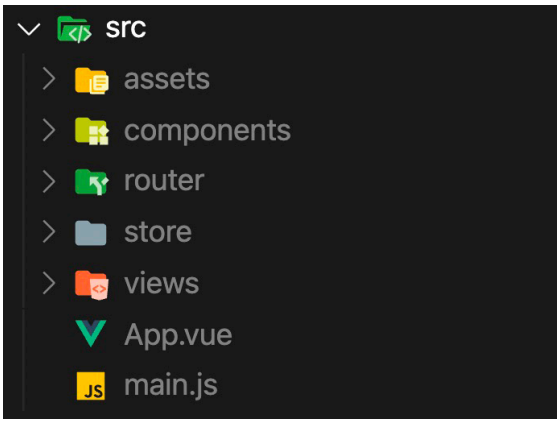


Slika 1.8.1. Ovisnost elemenata u Angularu

Aplikacija napisana u Angularu pokreće se pozivanjem glavnog modula, koji zatim prezentira sadržaj aplikacije internetskom pregledniku i reagira na interakcije korisnika prateći instrukcije koje su mu zadane. Glavni modul centralno je mjesto u kôdu aplikacije, gdje je zapisano kako se aplikacija pokreće i koji se elementi koriste.

1.9. Razvojni okvir Vue.js

Vue.js je razvojni okvir baziran na komponentama, pri čemu se svaka komponenta sastoji od funkcijskog, prezentacijskog i stilskog dijela. Vue.js koristi virtualni DOM kako bi se na brži način ažurirali elementi unutar aplikacije.



Slika 1.9.1. Struktura Vue.js datoteka

Vue.js također pruža razvojnom inženjeru brz način izrade SPA aplikacija jer dolazi sa zadanom organizacijom strukture datoteka, što, zbog predefinirane strukture datoteka koje se koriste, omogućuje da velike *frontend web*-aplikacije koje prikazuju kompleksnu strukturu mrežnih stranica budu lakše za održavanje. Kada je potrebno izraditi novu funkcionalnost, razvojni inženjer s obzirom na jasno definiranu strukturu datoteka točno zna gdje se koji dio *frontend web*-aplikacije nalazi.

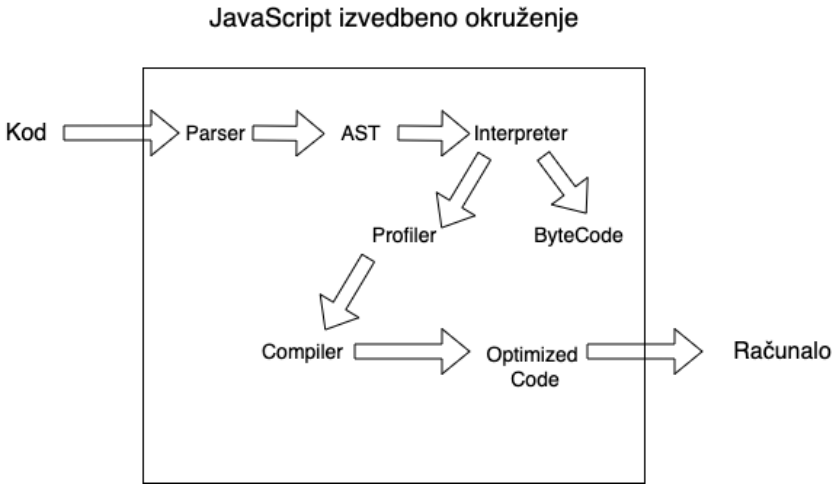
1.10. Statičke *web*-stranice

Statički mrežna stranica jednostavna je *frontend web*-aplikacija napisana korištenjem HTML i CSS programskih jezika. Kod ovih aplikacija kada poslužitelj primi zahtjev za prikaz određene HTML stranice, poslužitelj je pronalazi na tvrdom disku i potom je vraća klijentu u obliku odgovara. HTML stranica prikazuje se klijentu unutar ekrana internetskog preglednika. Statička mrežna stranica sadrži podatke koji su neprom-jenjivi, a podatke je moguće promijeniti jedino u izvornom kôdu samih datoteka. S obzirom na to da sadržaj koji se prikazuje ne ovisi o određenom klijentu, već se za

svakog klijenta prikazuje isti sadržaj, sadržaj se može unaprijed generirati, što dovodi do boljih performansi, a u nekim slučajevima i lakše upotrebe.

1.11. Prevođenje JavaScripta

Kod objave nove verzija JavaScript programskog jezika zajednica razvojnih inženjera uglavnom koristi najnoviju verziju, iako ona još nije podržana od strane svih internetskih preglednika. U takvim situacijama, a kako bi se omogućilo da internet preglednici podržavaju noviju verziju, dolazi do prevođenja najnovije verzije JavaScripta u stariju verziju, koju internetski preglednici podržavaju. Jedan od popularnijih JavaScript prevodioca jest Babel. Babel ne prevodi JavaScript kôd na razinu nula i jedinica, odnosno binarnog kôda, već umjesto toga prevodi JavaScript u sintaksu koja je razumljiva svim internetskim preglednicima. Kako bi računalno moglo razumjeti znakove koji su razvojnom inženjeru razumljivi, potrebno je izvedbeno okruženje (engl. *engine*) koje te znakove prevodi u nule i jedinice. JavaScript programski jezik koristi svoje izvedbeno okruženje za pretvorbu kôda u nule i jedinice. Bez JavaScript izvedbenog okruženja kôd koji bi razvojni inženjer napisao ne bi imao svrhu jer računalno ne bi imalo uputu za daljnje postupanje s kôdom. Postoji nekoliko vrsta JavaScript izvedbenih okruženja. Svako od izvedbenih okruženja zaduženo je za pojedinu platformu. Neke od vrsti JavaScript izvedbenih okruženja na primjer su: Chakra, Microsoft IE/Edge i SpiderMoneky, FireFox te V8, Chrome. Skup JavaScript izvedbenog okruženja naziva se ESMAScript izvedbeno okruženje. Izvedbeno okruženje je programski kôd, koji je razvio razvojni inženjer, najčešće služeći se programskim jezikom C++.



Slika 1.11.1. JavaScript izvedbeno okruženje

Slika 1.11.1. prikazuje od kojih se sve elemenata sastoji JavaScript izvedbeno okruženje. Vidljivo je da kôd putuje kroz nekoliko koraka kako bi ga računalno moglo razumjeti. Prvi je korak parser, parser čita liniju po liniju kôda te provjerava njegovu ispravnost. Ako je kôd neispravan, sadrži krivu sintaksu, parser kao rezultat vraća grešku razvojnom inženjeru o neispravnost i prekida daljnje izvođenje. Ako je kôd ispravan, parser kao produkt vraća strukturu podataka Abstract Syntax Tree (u daljnjem tekstu AST) i nastavlja se daljnje izvođenje. AST je izvorni kod *frontend web*-aplikacije koji se predaje interpreteru. Interpreter ima dvije mogućnosti: prva je mogućnost pročitati liniju po liniju kôda i prevesti kôd u niz bajtova na razinu nula i jedinica, koje računalno može razumjeti, a druga je mogućnost proslijediti kôd *profileru* na daljnju optimizaciju. Mogućnost koju interpreter odabere ovisi o razvojnom inženjeru koji postavlja postavke izvedbenog okruženja. *Profiler* na temelju zaprimljenog AST-a provjerava koji dio kôda koristi najviše performansi te kao rezultat prevodiocu (engl. *compiler*) dostavlja AST i izvještaj koji ukazuje na to koji dio kôda koristi najviše performansi. Prevodilac na temelju zaprimljenih podataka čita cijeli kôd koji se nalazi u AST-u, napravi optimizaciju kôda na temelju zaprimljenog izvještaja i prevodi kôd na razinu nula i jedinica, koje računalno može razumjeti.

1.12. Korištenje varijabli i konstanti kod razvoja frontend aplikacija

Svaka *frontend web*-aplikacija sastoji se od dvije temeljene komponente za čuvanje podataka. To su varijable i konstante. Kao što u matematičkoj jednadžbi postoje nezavisne varijable koje mogu poprimiti bilo koju vrijednost, tako se i u aplikacijama koriste varijable i konstante za spremanje podataka u memoriji. Varijable i konstante predstavljaju jedinstvene memorijske lokacije koje sadržavaju podatke, a koje aplikacija koristi. Vrijednosti koje se pohranjuju u varijable mogu se mijenjati tijekom izvođenja aplikacije, dok se vrijednosti koje su pohranjene u konstante ne mogu mijenjati nakon što su jednom definirane – to je osnovna razlika između varijabli i konstanti. Razlikuju se varijable na razini aplikacije i na razini operativnog sustava. Varijable na razini aplikacije zovu su aplikacijske varijable, a na razini operativnog sustava varijable okoline (engl. *environment variables*). Varijable okoline vrijednosti su koje se nalaze izvan aplikacije. Svaka varijabla okoline predstavljena je pomoću para imena i vrijednosti, gdje ime predstavlja naziv varijable pomoću kojeg aplikacija može dohvatiti vrijednost pohranjenu u varijablu okoline. Vrijednosti koje se pohranjuju u varijable oko-

line najčešće su vrijednosti koje se ne mijenjaju tijekom rada aplikacije, a od značaja su za njezin rad. U varijable okoline pohranjuju se različiti konfiguracijski podaci za određene vanjske module koje aplikacija koristi, kao što su na primjer: podaci za spajanje na bazu podataka i sigurnosni ključevi za kreiranje tokena. Prednost koju pružaju varijable okoline jest sigurnost aplikacije. To znači da u slučaju da aplikacija postane kompromitirana zbog napadača koji je uspio probiti sigurnosne mehanizme, on neće moći vidjeti vrijednosti varijable okoline jer se one ne nalaze unutar aplikacije. Varijable okoline mogu se dodatno zaštititi konfiguracijom tako da samo određena razina korisnika koja koristiti operativni sustav može čitati varijable okoline. Time se dodaje još jedan sloj zaštite oko varijabli okoline.

1.13. Optimizacija *frontend web*-aplikacije za web-pretraživače

Izrađena *frontend web*-aplikacija postavlja se na internet tako da je svima dostupna. Prilikom pretraživanja, a kako bi se aplikacija pojavljivala među prvim ponuđenim aplikacijama, mora se ugraditi tzv. SEO (engl. *Search Engine Optimization*). SEO je mehanizam pomoću kojeg se optimizira *frontend web*-aplikacija kako bi internetski pretraživači (tražilice), npr. Google, bolje rangirali mrežnu stranicu koja stoji iza *frontend web*-aplikacije. Ovaj mehanizam optimizacije bazira se na ključnim riječima. Ključne riječi odnosne se na naslove, podnaslove, grafikone i fotografije koji se nalaze na mrežnoj stranici, a do kojih tražilice dolaze kroz *frontend web*-aplikaciju. Prednosti SEO optimizacije jesu: brendiranje i povećanje pretraživosti, povećanje prometa i prepoznavanje interesa korisnika.

1.14. Cross-Origin Resource Sharing problemi kod razvoja *frontend* aplikacija

CORS (engl. *Cross-Origin Resource Sharing*) koncept je koji omogućuje *frontend web*-aplikacijama koje se nalaze na različitim domenama da međusobno razmjenjuju informacije. Ovaj mehanizam postoji iz sigurnosnih razloga, jer internet preglednici blokiraju dohvaćanje podataka iz različitih domena. Domena je znakovni niz koji se koristi za pristup *frontend web*-aplikaciji korištenjem internetskog preglednika. Među-

tim, mogu postojati slučajevi u kojima se *frontend web*-aplikacijama želi omogućiti da iz različitih domena međusobno razmjenjuju informacije. U takvim slučajevima koristi se CORS. Da bismo koristili CORS, potrebno je znanje na koji način dvije *frontend web*-aplikacije na različitim domenama komuniciraju. Najčešći protokol za komunikaciju jest HTTP (engl. *Hypertext Transfer Protocol*) protokol, koji koristi različite metode slanja podataka između *frontend web*-aplikacija. Koriste se četiri glavne metode za slanje podataka. To su: GET, POST, PUT i DELETE. CORS određuje HTTP metode pomoću kojih će dvije *frontend web*-aplikacije iz različitih domena komunicirati.

1.15. Testiranje frontend aplikacija

Za kvalitetnu *frontend web*-aplikaciju nužno je provoditi testiranje korisničkog sučelja. S obzirom na pouzdanost proizvoda koja mora biti na visokom nivou, u današnje vrijeme postoje različite metodologije testiranje svakog segmenta *frontend web*-aplikacije. Tijekom provođenja testiranja *frontend web*-aplikacije potrebno je imati dobro razrađen plan o provođenju testiranja kako bi se svi segmenti aplikacije provjerili. Testiranje aplikacije poželjno je kada više programera ili timova programera radi na jednoj aplikaciji. Može se dogoditi da dva razvojna inženjera rade na istoj metodi koja obrađuje neku operaciju, a ne rade to na isti način ili nemaju isto shvaćanje što se želi postići. U tom slučaju, ako ne postoje testovi koji definiraju što metoda prima kao ulazne vrijednosti i što metoda vraća kao izlazne vrijednosti, može doći do preklapanja kôda s jedne i druge strane, što može uzrokovati neispravan rad aplikacije. Dvije najčešće metodologije testiranja aplikacije jesu integracijski i jedinični testovi. Integracijski testovi (engl. *Integration test*) imaju za cilj testirati zajednički rad različitih komponenti aplikacije. Koriste se kada se želi testirati komunikacija između klijenta i poslužitelja i kada postoji više komponenti izrađenih od strane više razvojnih inženjera. Ovaj je način testiranje sporiji, zahtijeva više resursa (CPU, memorija), ali veći je opseg funkcionalnosti koje testiramo. Za izradu integracijskih testova postoje razne biblioteke, s time da ne postoje univerzalne, već svaki programski jezik ima svoje specifične biblioteke. Jedinični testovi (engl. *Unit test*) koriste se za testiranje izoliranog kôda, koji je predstavljen kao zasebna jedinica. Kod jediničnih testova testiraju se različite metode koje koristi *frontend web*-aplikacija te svojstva i interaktivni korisnički elementi aplikacije. U praksi se mnogo vremena posvećuje pisanju jediničnih testova, budući da oni omogućuju provjeru rada ispravnosti kôda i jamče bolje performanse

frontend web-aplikacije u korištenju. Naime, testiranjem pozitivnih i negativnih načina rada kôda jasno se i precizno definira očekivano ponašanje aplikacije. Pozitivan način rada kôda jest način testiranja kod kojeg se koristi ispravan skup podataka, dok je negativan način rada kôda testiranja kod kojeg se koristi neispravan skup podataka. Ispravan skup podataka podaci su koji kao rezultat testiranja daju željeno ponašanje *frontend web*-aplikacije, dok su neispravan skup podataka podaci koji kao rezultat testiranja daju neželjeno ponašanje *frontend web*-aplikacije. Testiranjem pozitivnog i negativnog načina rada kôda uočavamo kako *frontend web*-aplikacija radi s ispravnim i neispravnim podacima te se pomoću rezultata testiranja aplikaciju može prilagoditi kako bi se izbjegla neželjena ponašanja. Jedinično je testiranje brzo testiranje, zahtijeva malo resursa, ali opseg testiranja puno je manji nego kod integracijskog testiranja, pa se može dogoditi da se ne otkriju svi nedostaci *frontend web*-aplikacije.

PITANJA ZA PONAVLJANJE:

1. Što je *frontend* web-aplikacija?
2. Koje su temeljene tehnologije koje prepoznaju internetski preglednici?
3. Kako biste opisali *web*-aplikaciju?
4. Što nam pruža MPA vrsta *frontend web*-aplikacije, a što nam pruža SPA vrsta *frontend web*-aplikacije?
5. Koje JavaScript razvojne okvire poznajete?
6. Koja je uloga prevoditelja?
7. Koja je uloga varijabli okoline u *web*-aplikacijama?
8. Koje vrste HTTP metoda postoje?
9. Koja je razlika između integracijskog i jediničnog testa?

1.16. Backend aplikacije

Jeste li se ikad zapitali na koji način *frontend web*-aplikacija dohvaća i prikazuje različite podatke različitim korisnicima? Odgovor na ovo pitanje daje *backend* aplikacija. Svaka *web*-aplikacija osim *frontend* dijela sadrži i tzv. backend dio. *Backend* je termin u programiranju koji razvojni inženjeri koriste kako bi se referirali na programsku logiku i funkcionalnost *web*-aplikacije koja se nalazi u pozadini. Kao što je već poznato, svaka *web*-aplikacija sastoji se od poslužiteljskog dijela i klijentskog dijela *web*-aplikacije. Komunikacija između poslužitelja i klijenta ključna je za rad *web*-aplikacije jer se razmjenom informacija stvara jedna zaokružena cjelina. Poslužiteljski dio *web*-aplikacije tj. *backend* dio zaslužan je za logiku *web*-aplikacije, dohvaćanje podataka iz baze podataka i dostavljanje podataka klijentskom dijelu *web*-aplikacije, odnosno *frontendu*. Kako bi se ostvarila komunikacija između *frontenda* i *backenda*, mora se posjedovati određeno znanje u primjeni aplikacijskog programskog sučelja (engl. *Application Programming Interface*). Razlog tome jest to što je aplikacijsko programsko sučelje standard koji u današnje vrijeme većina *web*-aplikacija koristi.

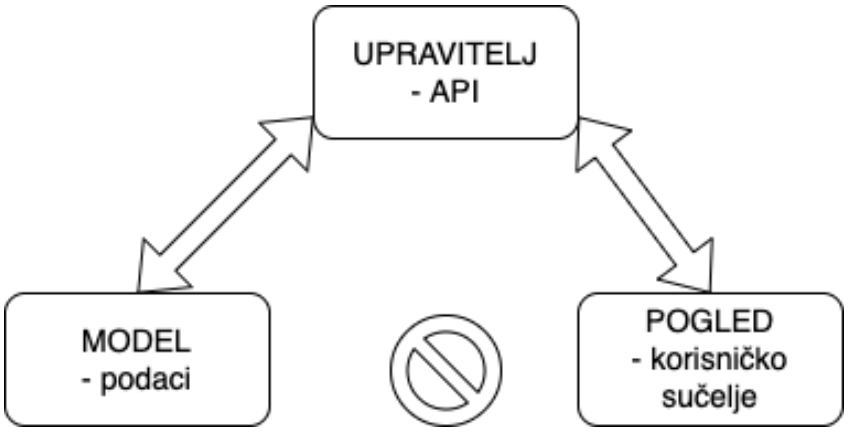
1.17. Uvod u aplikacijska programska sučelja

Aplikacijsko programsko sučelje (u daljnjem tekstu skraćeno API) skup je pravila koja razvojni inženjeri koriste u radu s *web*-aplikacijama. API u *backend* aplikacijama predstavlja određene putanje koje *frontend* dijelu omogućuju komunikaciju s *backend* dijelom aplikacije. Počeci primjene API-ja datiraju iz 2000. godine, kada je Roy Fielding, američki računalni znanstvenik, uz pomoć tehnoloških giganta Salesforce, eBay i Amazon oblikovao i dizajnirao standarde API-ja koje i danas koristimo. Nakon toga, 2004. godine dolazi do znatne distribucije i korištenja API-ja od strane ostalih tehničkih kompanija, koje su te standarde koristile za razvoj svojih proizvoda. Tako su 2010. godine tvrtke koje upravljaju društvenim mrežama postavile javne API-je. Na taj je način svakom razvojnom inženjeru omogućeno korištenje API-ja za vlastite potrebe. Kako bi razvojni inženjer koristio takvu vrstu API-ja, potreban mu je tzv. token. Token predstavlja jedinstveni ključ putem kojega se prepoznaje koji razvojni inženjer koristi koji API. U tokenu su zapisani podaci o razinama prava razvojnog inženjera koji se njime koristi. API se koristi u mnogim granama programiranja, kao što su programiran-

ja za desktop aplikacije, *web*-aplikacije i mobilne aplikacije. Svrha API-ja dohvaćanje je i pristup podacima koji se koriste na dnevnoj bazi u svrhu svakog pojedinog posla. U današnje vrijeme postoje različiti programski jezici pomoću kojih razvojni inženjeri dizajniraju i razvijaju *backend* aplikacije. Među najmodernijim programskim jezicima za razvijanje *backend* aplikacija ističu se JavaScript, Python, Java i Golang. Svaki od ovih programskih jezika koristi svoju specifičnu sintaksu za izradu API-ja.

1.18. Arhitekturni obrazac MVC

Arhitekturni obrazac način je dizajna *web*-aplikacije koji jasno definira od kojih se elemenata sastoji web-aplikacija i koji način komunikacije koristi. Arhitekturni obrazac *Model-view-controller* (u daljnjem tekstu skraćeno MVC) arhitekturni je obrazac koji je vođen idejom da se aplikacija organizira u tri sloja, svaki prema svojim odgovornostima. Korištenjem MVC arhitekturnog obrasca ubrzan je razvoj izrade *web*-aplikacije jer postoji gotov predložak koji razvojni inženjer koristi. Ujedno je i jedan od popularnijih predložaka kojim se pri izradi aplikacija koriste razvojni inženjeri jer odvaja poslovnu logiku od prezentacijskog djela. MVC se sastoji od tri ključna sloja: model, pogled (engl. *View*) i upravitelj (engl. *Controller*). Model komunicira s bazom podataka iz koje dohvaća podatke. Pogledi se koriste za prikaz podataka korisniku na sučelju, a u *web*-aplikacijama najčešće se sastoje od programskih jezika HTML (engl. *HyperText Markup Language*) i CSS (engl. *Cascading Style Sheets*). Upravitelj je sloj čija je zadaća povezivanje slojeva model i pogled kako bi se generirao odgovor krajnjem korisniku.



Slika 1.18.1. MVC arhitektura

Na slici 1.18.1. nalazi se prikaz modula MVC arhitekture i njihova međusobna komunikacija. Iz prikazanog vidljivo je da model i pogled nisu međusobno povezani, već informacije razmjenjuju preko upravitelja. Na taj je način prezentacijski sloj odvojen od sloja poslovne logike.

Prednosti korištenja MVC arhitekture:

- funkcionalnosti su međusobno podijeljene na modele, poglede i upravitelje
- poslovna logika razdvojena je od prezentacijskog dijela
- smanjuje se redundancija programskog kôda
- jednom izrađena aplikacija u MVC arhitekturi održiva je i skalabilna
- svaki sloj MVC arhitekture moguće je testirati neovisno o drugome.

Komunikacija korištenjem MVC arhitekture započinje zahtjevom pogleda za prikaz podataka iz baze podataka. Nakon toga pogled šalje zahtjev prema upravitelju za dohvat podataka, koji ga onda prosljeđuje modulu za izvršavanje. Na temelju zaprimljenog zahtjeva model pomoću procedura dohvaća podatke iz baze podataka, iste vraća upravitelju, koji podatke prosljeđuje pogledu. Tako zaprimljene podatke pogled prikazuje korisniku na sučelju. Procedure su skup naredbi u bazi podataka kojima je svrha kreirati, čitati, ažurirati ili brisati podatke; svaka procedura ima svoje jedinstveno ime. Pomoću imena procedure procedura se poziva na izvođenje. Komunikacija između upravitelja i pogleda ostvaruje se upotrebom API-ja.

1.19. Varijable okoline kod razvoja backend aplikacija

U svakoj *backend* aplikaciji obveza je razvojnog inženjera implementacija sigurnih varijabli okoline. Naime, u većini slučaja *backend* aplikacije sadržavaju osjetljive podatke koje je potrebno zaštiti od „trećih osoba” i za tu se svrhu upotrebljavaju varijable okoline. Memorijska lokacija varijabli okoline nalazi se unutar operativnog sustava, što znači da su one odvojene od aplikacije. Podaci koji se ne mijenjaju tijekom izvođenja aplikacije, a potrebno ih je čuvati na sigurnom mjestu, pohranjuju se u varijable okoline. Varijable okoline pohranjuju se u posebnoj datoteci na operativnom sustavu, pod nazivom „env”. Osim „env” datoteke varijable okoline mogu se spremati i u druge da-

toteke, na primjer: inicijalizacijske datoteke, JSON ili XML datoteke i YAML datoteke. Datoteka postoji neovisno o operativnom sustavu na kojem se nalaz web-aplikacija, bilo da je riječ o operativnim sustavima Windows ili Linux. Svaka varijabla okoline sastoji se od ključa i vrijednosti. Ključ je oznaka za naziv, odnosno ime varijable okoline pomoću kojeg se dohvaća vrijednost ključa. U varijable okoline pohranjuju se npr. podaci za spajanje na bazu podataka: korisničko ime, lozinka, ime baze podataka i TCP (engl. *Transmission Control Protocol*) port, koji se koristi za povezivanje na bazu podataka, različiti konfiguracijski podaci koji se koriste za spajanje na vanjske module koje koristi *web*-aplikacija, sigurnosni ključevi za kreiranje tokena itd.

Varijable okoline mogu se čitati i pozivati s „obje strane", dakle i s backend i frontend strane, s time da postoje određene razlike. Kada se koristi varijabla okoline na *frontend* strani, korisnik može vidjeti ključ, odnosno naziv varijable okoline. To dalje znači da u takvom slučaju korisnik može pročitati i vrijednost varijable okoline, a to svakako ne predstavlja dobru praksu, jer se time narušava sigurnost osjetljivih podataka. Varijable okoline rijetko se koriste na *frontend* strani. U praksi uglavnom se koriste na *backend* strani, jer se na *backendu* nalazi cijela logika *web*-aplikacije. *Backend* je zadužen za spajanje na bazu podataka i za spajanje s ostalim vanjskim modulima koje koristi *web*-aplikacija.

1.20. Dijeljenje informacija između web-sjedišta

CORS (engl. *Cross-Origin Resource Sharing)* mehanizam je koji se koristi kako bi *web*-aplikacije na različitim domenama mogle razmjenjivati informacije. Dijeljenje informacija između *web*-aplikacija izuzetno je važno jer najčešće u praksi jedna *web*-aplikacija koristi podatke druge aplikacije. Primjerice, *web*-aplikacija koristi sustav za plaćanje putem interneta PayPal, gdje se u takvoj komunikaciji razmjenjuju podaci o statusu plaćanja računa. Problem dijeljenja informacija između *web*-aplikacija različitih domena rješava CORS i to tako da u postavkama *web*-aplikacije definira s koje je domene dopušten prolazak HTTP paketa. Paket se sastoji od zaglavlja i tijela. Zaglavlje sadržava polazište i odredišta paketa, gdje je polazište domena *web*-aplikacije s koje se šalje paket, a odredište je domena *web*-aplikacije kojoj se dostavlja paket. Zaglavlje sadržava i podatak koji označuje kojom se HTTP (engl. *HyperText Transfer*

Protocol) metodom šalje paket. U tijelu se nalaze podaci koji se šalju. HTTP paketi mogu se transportirati na različite načine. Neki od načina transporta jesu obrasci (engl. *Form data*), teretni zahtjevi (engl. *Request Payload*) i binarni način transporta.

U nastavku se daje primjer razmjene informacija između dviju *web*-aplikacija na različitim domenama.

Web-aplikacija „a" nalazi se na domeni HTTP://domain-a.com, ona šalje paket prema drugoj *web*-aplikaciji, „b", koja se nalazi na domeni HTTP://domain-b.com. Nakon toga *web*-aplikacija „b" provjerava s koje je domene došao paket i ako paket nije došao s domene *web*-aplikacije, „b", ona će taj paket odbaciti zbog nepovjerenja. Rješenje problema razmjene paketa između *web*-aplikacija „a" i „b" jest da se u postavkama *web*-aplikacije „b" pomoću kodiranja definira domena *web*-aplikacije „a" kao domena kojoj se vjeruje.

1.21. Dijeljenje informacija između web-sjedišta

CORS (engl. *Cross-Origin Resource Sharing*) mehanizam je koji se koristi kako bi *web*-aplikacije na različitim domenama mogle razmjenjivati informacije. Dijeljenje informacija između *web*-aplikacija izuzetno je važno jer najčešće u praksi jedna *web*-aplikacija koristi podatke druge aplikacije. Primjerice, *web*-aplikacija koristi sustav za plaćanje putem interneta PayPal, gdje se u takvoj komunikaciji razmjenjuju podaci o statusu plaćanja računa. Problem dijeljenja informacija između *web*-aplikacija različitih domena rješava CORS i to tako da u postavkama web-aplikacije definira s koje je domene dopušten prolazak HTTP paketa. Paket se sastoji od zaglavlja i tijela. Zaglavlje sadržava polazište i odredišta paketa, gdje je polazište domena *web*-aplikacije s koje se šalje paket, a odredište je domena *web*-aplikacije kojoj se dostavlja paket. Zaglavlje sadržava i podatak koji označuje kojom se HTTP (engl. *HyperText Transfer Protocol*) metodom šalje paket. U tijelu se nalaze podaci koji se šalju. HTTP paketi mogu se transportirati na različite načine. Neki od načina transporta jesu obrasci (engl. *Form data*), teretni zahtjevi (engl. *Request Payload*) i binarni način transporta.

U nastavku se daje primjer razmjene informacija između dviju *web*-aplikacija na različitim domenama.

Web-aplikacija „a” nalazi se na domeni HTTP://domain-a.com, ona šalje paket prema drugoj *web*-aplikaciji, „b”, koja se nalazi na domeni HTTP://domain-b.com. Nakon toga *web*-aplikacija „b” provjerava s koje je domene došao paket i ako paket nije došao s domene *web*-aplikacije, „b”, ona će taj paket odbaciti zbog nepovjerenja. Rješenje problema razmjene paketa između *web*-aplikacija „a” i „b” jest da se u postavkama *web*-aplikacije „b” pomoću kodiranja definira domena *web*-aplikacije „a” kao domena kojoj se vjeruje.

1.22. Standard za prijenos podataka XML

XML je prvi osmišljeni standard za prijenos podataka, koji se primjenjuje od 1990. godine. XML je znakovni jezik baziran na HTML prezentacijskom jeziku. Njegovom primjenom dolazi do napretka u prijenosu podataka. Uz pomoć XML-a svaki podatak koji se prosljeđuje između web-aplikacija može se provjeriti po podatkovnom tipu. Tako na primjer XML omogućuje provjeru je li podatak broj, znakovni niz, decimalan broj itd. S obzirom na to da se XML temelji na HTML prezentacijskom jeziku, od njega je preuzeo i neka njegova sintaksna svojstva. Podaci se unutar XML-a definiraju pomoću oznaka (engl. tag), gdje svaka oznaka sadrži svoju početnu oznaku (matematička oznaka za manje <) i završnu oznaku (matematička oznaka za više >). Naziv oznake razvojni inženjer zadaje proizvoljno i na temelju toga definira strukturu podataka koji će se slati pomoću XML-a. Svakoj *web*-aplikaciji koja koristi XML potreban je parser. Parser služi za čitanje sadržaja XML-a i najčešće je predstavljen kao XML shema pomoću koje se provjerava valjanost podataka koji se nalaze unutar XML-a. Ukoliko XML struktura ne zadovoljava definirana pravila XML sheme, podaci nisu važeći i *web*-aplikacija može odbiti te podatke i odgovoriti povratnom porukom da dostavljeni podaci nisu važeći. Prevoditelj ovisno o veličini i složenosti XML strukture može sporije ili brže provjeravati valjanost podataka koji se nalaze unutar XML-a, a što se tiče veličine i složenosti strukture, tu ne postoje nikakva ograničenja. U svakom slučaju, uputno je izbjegavati velike XML strukture jer se manjim strukturama štede resursi i dobivaju bolje performanse. Standard prijenosa podataka XML zbog razvoja novih tehnologija sve se manje koristi, a postupno ga zamjenjuje JSON standard za prijenos podataka, koji se lakše održava i omogućuje jednostavnije čitanje podataka.

1.23. Standard za prijenos podataka JSON

Razvoj JSON standarda započinje 2001. godine. Američki razvojni inženjeri Sam Douglas Crockford i Chip Morningstar poslali su prvi paket u JSON formatu. Temelj za izradu JSON formata objekti su JavaScript programskog jezika, koji zbog lakoće pisanja i razumijevanja olakšavaju brži i jednostavniji prijenos podataka. Razvojem JavaScripta ovaj je programski jezik propisao da u svojim aplikacijama razvojni inženjeri koriste JSON format za prijenos podataka. Time se je utjecalo i na širinu primjene JSON formata u ostalim programskim jezicima i na postupni prelazak s XML formata u JSON format za prijenos podataka. Razvojem JSON formata prijenos postaje lakši za primjenu razvojnim inženjerima jer je njegova struktura lako prihvatljiva i razumljiva. Struktura JSON formata sastoji se od objekata koji su definirani vitičastim zagradama. Objekt JSON strukture sastoji se od ključa i vrijednosti. Ključevi su jedinstveni na razinama na kojima se nalaze, a isti ključevi se mogu nalazi na različitim razinama. To znači da dva ista ključa ne mogu postojati na istoj razini. Vrijednosti se dohvaćaju pomoću ključeva. Kako bi web-aplikacija mogla pročitati JSON format, također joj je potreban parser. Parser pretvara podatke koji se nalaze u JSON formatu u objekte. Objekt predstavlja skup podataka koji čine zaokruženu cjelinu, kao što je na primjer skup podataka o čovjeku. *Web*-aplikacija definira veličinu JSON formata koju može primiti. Predefinira veličina JSON formata iznosi 2 gigabajta, ali može se mijenjati. Brzina prijenosa podataka putem JSON formata ovisi o brzini internetske veze i količini podataka koji se nalaze unutar JSON formata. Ukoliko JSON format ima 10 kilobajta, on može puno brže prenijeti podatke od jedne *web*-aplikacije do druge, za razliku od JSON formata veličine 2 gigabajta. Zbog svoje strukture i jednostavnosti u korištenju te olakšanog i bržeg prijenosa podataka između web-aplikacija JSON format koristi se sve više u novim i modernim aplikacijama.

1.24. Arhitektura backend aplikacija

Arhitektura *backend* aplikacije temelj je cijele *web*-aplikacije. Ona definira način na koji će *frontend* i *backend* komunicirati, odnosno kojim će se protokolom služiti u toj komunikaciji. Stoga je arhitektura *backend* aplikacije prvi zadatak svakog razvojnog inženjera. Pri odabiru arhitekture *backend* aplikacije potrebno je analizirati obujam

aplikacije i način na koji će *frontend* povlačiti podatke od *backenda*. Jednom kada se arhitektura postavi, kasnije tijekom izvedbe aplikacije teže ju je mijenjati. U stvaranje arhitekture aplikacije uključeno je stvaranje njene strukture i logike, a što opet uključuje stvaranje svih pozadinskih procesa aplikacije koji korisniku nisu vidljivi. Pri stvaranju arhitekture *backend* aplikacije potrebno je biti pažljiv i odgovoran jer odabir arhitekture utječe i na frontend i na ukupnu funkcionalnost i efikasnost *web*-aplikacije. Uspješna arhitektura *web*-aplikacije omogućuje skalabilnost, dok istovremeno odvaja internu logiku *web*-aplikacije od prezentacijskog dijela. Razvijanjem ispravne arhitekture aplikacije razvija se moderna *web*-aplikacija, koju korisnici mogu koristiti bez angažiranja velike procesorske snage svog računala. Razlikujemo tri arhitekture aplikacije koje se danas uglavnom koriste. To su REST (engl. Representational State Transfer), GraphQL i SOAP (engl. Simple Object Acccess Protocol). REST i GraphQL moderne su arhitekture koje se zbog svoje jednostavnosti u korištenju češće koriste, za razliku od SOAP-a kod kojega postoje ograničenja u prijenosu podataka.

1.25. Arhitektura aplikativnog programskog sučelja REST

REST predstavlja način dizajna API arhitekture. Značajna karakteristika korištenje REST-a jest fleksibilnost i mogućnost prilagođavanja aplikacije u kasnijem razvoju. REST definira na koji će način komunicirati *frontend* i *backend*. To znači da kada *frontend* zatraži određene podatke od *backenda*, REST ovisno o postavljenom zahtjevu *frontendu* vraća određeni odgovoru u XML ili JSON formatu. *Backend* prilikom primanja i vraćanja podataka može koristi oba formata, iako je praksa da se koristi samo jedan. *Backend* koji koristi oba formata za razmjenu podataka najčešće imaju *web*-aplikacije koje su javno dostupne svim razvojnim inženjerima. REST koristi HTTP (engl. *Hypertext Transfer Protocol*) protokol za razmjenu informacija između *frontenda* i *backenda*, a podaci se razmjenjuju pomoću HTTP paketa. HTTP paketi sastoje se od zaglavlja (engl. *headers*) i tijela (engl. *body*). U zaglavlju svakog paketa nalazi se putanja koja definira polazište paketa i njegovo odredište, format podataka koji se nalaze u paketu te metodu zahtjeva i šifru statusa (engl. *status code*). U tijelu paketa nalaze se podaci koji se odnosne na *frontend* i *backend*. REST radi po principu zahtjeva i odgovora, gdje zahtjev upućuje *frontend*, a odgovor pruža *backend*. HTTP protokol sadrži različite metode prijenosa podataka. Svaka metoda ima svoju svrhu korištenja i namjenu. To je dogovoreni standard kojeg se pridržavaju svi razvojni inženjeri.

Razlikujemo sljedeće najčešće korištene metode:

- **GET** -> koristi se za dohvaćanje podataka
- **POST** -> koristi se za kreiranje novog resursa
- **PUT** -> koristi se za ažuriranje postojećeg resursa
- **DELETE** -> koristi se za brisanje resursa.

Osim navedenih metoda postoje i druge koje se rjeđe koriste, kao što su npr. PATCH i OPTIONS.

1.26. Arhitektura aplikativnog programskog sučelja GraphQL

Kao i REST, GraphQL je arhitektura *backend* aplikacije koja nam omogućuje razmjenu informacija između *frontenda* i *backenda*. GraphQL su osmisli razvojni inženjeri Facebooka. Cilj mu je napraviti dodatnu razinu apstrakcije oko REST arhitekture korištenjem upita za razmjenu podataka između *frontenda* i *backenda*. Za razliku od REST arhitekture gdje *backend* definira koje će podatke pojedina putanja vratiti *frontendu* na njegov zahtjev, kod GraphQL arhitekture *frontend* sam definira koje podatke želi od *backenda*. Dakle kod GraphQL arhitekture *backend* nije u obvezi definirati koje će podatke vratiti pojedina putanja, već *frontend* u zahtjevu traži podatke koji su mu potrebni. To je ujedno i primarna razlika između GraphQL i REST arhitekture. Neke od prednosti korištenja GraphQL-a jesu dodatno odvajanje *frontenda* od *backenda*, izostanak prekomjernih ili nedovoljno dohvaćenih podataka i kraće izvršavanja zahtjeva. Korištenjem GraphQL-a nije potrebno navoditi dodatne putanje za dohvat podataka, već se koristi jedna apstraktna putanja za sve zahtjeve, dok korištenjem REST arhitekture pojedina putanja može vratiti nedovoljnu količinu podataka, što onda zahtijeva kreiranje novih putanja. Kada govorimo o podacima koje GraphQL vraća *frontendu*, važno je napomenuti da GraphQL za slanje podataka koristi jedino JSON format. GraphQL se najviše koristi kod velikih aplikacija u kojima dolazi do razmjene velike količine podataka između *frontenda* i *backenda*. U takvim situacijama GraphQL omogućuje *backendu* potpunu prilagodbu *frontendu*.

1.27. Arhitektura aplikativnog programskog sučelja SOAP

SOAP je vrsta arhitekture *backend* aplikacija koja se temelji na razmjeni podataka između *frontenda* i *backenda* korištenjem XML formata za razmjenu podataka. Primjenjuje se od 1998. godine, kad su ga razvili razvojni inženjeri Microsofta. SOAP kao i REST koristi HTTP protokol za razmjenu podataka između *web*-aplikacija. SOAP vrsta arhitekture može koristiti bilo koji programski jezik; osim HTTP protokola za razmjenu podataka podržava i druge protokole kao što su SMTP (engl. *Simple Mail Transfer Protocol*) i JMS (engl. *Java Message Service*) te sadržava ugrađeno upravljanje greškama. Koristi se kada se želi ostvariti komunikacija između *web*-aplikacija koje su izrađene pomoću različitih programskih jezika. Zbog ograničenosti uzrokovane primjenom prijenosa podataka samo putem XML formata koriste ga rijetke *web*-aplikacije.

1.28. Testiranje web-aplikacija

S ciljem da se isporuči pouzdana *web*-aplikacija s adekvatnim prikazom sadržaja prije svake isporuke potrebno je provesti njeno testiranje. Za svaku *web*-aplikaciju trebaju se napisati testovi koji provjeravaju ispravnost rada *frontenda* i *backenda*. Provođenjem testiranja definiraju se pravila kako se *web*-aplikacija mora ponašati te što je kod nje dopušteno, a što nije. Testovi su zaslužni za poboljšanje performansi *web*-aplikacije jer se pomoću njih testira brzina izvođenja API poziva, što dalje utječe na zaključak koji su API-ji dobro optimizirani. Testovi su ključ cijele *web*-aplikacije jer omogućuju otkrivanje logičkih grešaka. Postoji nekoliko vrsta testova koje je potrebno imati u sklopu *backend* aplikacije. Izdvajaju se integracijski testovi (engl. *Integration testing*), jedinični testovi (engl. *Unit testing*), funkcijsko testiranje (engl. *Functional testing*) i nefunkcijsko testiranje (engl. *Non-function testing*). Integracijski test vrsta je testiranja kod koje testiramo kako različiti dijelovi aplikacije rade zajedno. Kada su dijelove aplikacije razvili različiti razvojni inženjeri, gdje svaki razvojni inženjer na svoj način razvija svoj dio aplikacije, važno je provoditi integracijsko testiranje kako bi se provjerilo rade li dijelovi aplikacije u skladu s očekivanjima. Jedinični test provjerava kako pojedini dio aplikacije radi samostalno. Funkcijsko testiranje vrsta je testiranja koja provjerava rad aplikacije u usporedbi sa specifikacijskom funkcionalnosti aplikacije. Specifikacija funkcionalnosti dokument je koji definira kako se aplikacija mora ponašati u određen-

im situacijama i koje funkcionalnosti aplikacija mora sadržavati. Nefunkcijsko testiranje vrsta je testiranja koja se koristi kada se testiraju performanse, upotrebljivost i pouzdanost aplikacije. U ovom testiranju ne provjerava se funkcionalnost aplikacije. Primjer provođenja nefunkcijskog testiranja predstavlja provjera broja korisnika koje u isto vrijeme može podržati aplikacija.

PITANJA ZA PONAVLJANJE:

1. Što je to *backend* aplikacija?
2. Što je API?
3. Koje programske jezike za izradu *backend* aplikacija poznajete?
4. Koje podatke tipično spremamo u varijable okoline?
5. U kontekstu MVC-a, što je uloga upravitelja (engl. *Controller*)?
6. Koje vrste *backend* arhitekture postoje?
7. Nabrojite najčešće korištene HTTP metode?
8. Koju ulogu ima testiranje u *backend* aplikacijama?

1.29. Uvod u baze podataka

Baza podataka organiziran je skup povezanih podataka koji su pohranjeni na vanjskoj memoriji, a istovremeno su dostupni razvojnom inženjeru. Svaka web-aplikacija u pravilu koristi bazu podataka. Bazu podataka koristimo za spremanje skupova podataka na strukturiran način. Baze podataka nastale su 1960. godine zbog potrebe za automatskom obradom podataka. Pojavom baza podataka razvijaju se dinamičke mrežne stranice. Dinamičke mrežne stranice jesu stranice koje mijenjaju svoj sadržaj u skladu s ponašanjem korisnika. Za upravljanje bazom podataka koristi se strukturni upitni jezik (engl. *Structured Query Language*) ili skraćeno SQL, koji služi za čitanje, ažuriranje, spremanje i brisanje podataka iz baze podataka. Baze podataka razlikuju se po strukturi i načinu na koji spremaju podatke. Razlikujemo dvije osnovne vrste baze podataka. To su relacijske baze podataka i nerelacijske baze podataka.

1.30. Relacijske baze podataka

Relacijska baza podataka skup je podataka koji su logično organizirani kako bi sačinjavali jednu veliku logičnu cjelinu. Podaci koji se pohranjuju u relacijsku bazu podataka u međusobnim su odnosima i na taj način čine sliku stvarnog života. Primjer toga pohrana je podataka o roditeljima i djeci, gdje jedan roditelj može imati više djece, a jedno dijete može imati više roditelja. Baza podataka sastoji se od tablica u koje se spremaju podaci. Svaka tablica sastoji se od stupaca, polja i redaka. Svaki stupac u tablici pohranjuje određenu vrstu podataka, a vrijednosti se pohranjuju u polja. Redak se sastoji od skupa vrijednosti polja. Svaki redak u tablici predstavljen je jedinstvenim identifikatorom koji se naziva primarni ključ (engl. *primary key*). Retke između više tablica možemo povezati pomoću stranih ključeva (engl. *foreign key*) te se i odnosi, tj. veze između više tablica ostvaruju pomoću stranih ključeva. Zbog odnosa, odnosno veza između tablica, ova baza podataka naziva se relacijskom bazom podataka. Relacijske baze podataka imaju svoje prednosti i nedostatke. Prednosti su brzina, sigurnost, jednostavnost, preciznost i pristupačnost, dok se kao nedostaci mogu istaknuti lošije performanse, velika količina memorije koju koriste takve baze podataka i prostorna ograničenost na hard disku. Kako bi upravljanje relacijskom bazom podataka bilo uspješno, potrebno je poznavati sustave za njihovo upravljanje. Neki od

sustava za upravljanje relacijskim bazama podataka jesu npr. Microsoft SQL Server, PostgreSQL, MySQL ili MariaDB.

1.31. Nerelacijske baze podataka

Nerelacijska baza podataka ili skraćeno NoSQL jest baza podataka koja je manje strukturirana i manje ograničena u formatu podataka. To joj omogućuje da bude fleksibilna i prilagodljiva. U situacijama kada podaci s kojima se radi nisu jasno definirani te se njihova struktura međusobno razlikuje nerelacijska baza podataka olakšava upravljanje takvim podacima. Naime, nerelacijska baza podataka ne koristi tablice za pohranjivanje podataka, već podatke pohranjuje u JSON formatu ili XML formatu. Korištenjem ovih formata podataka isključena su ograničenja strukture podataka, tako da je time omogućeno jednostavno i brzo upravljanje nestrukturiranim podacima. Nerelacijske baze podataka koriste se uglavnom u sustavima u kojima je prisutna velika razmjena podataka, kao što su na primjer aplikacije Facebook, Instagram, eBay ili Amazon. Korištenjem nerelacijske baze podataka ovim sustavima omogućena je skalabilnost te brzina spremanja i dohvata podataka, što im osigurava popularnost u korištenju. Isto kao i relacijske baze podataka, tako i nerelacijske baze podataka imaju svoje prednosti i nedostatke. Prednosti nerelacijske baze podataka brzina su izvođenja upita, cijena korištenja i fleksibilnost. Od nedostataka mogu se navesti izostanak standardiziranog upitnog jezika, nedosljednost podataka i manjak standardiziranih pravila. Isto tako i za rad s nerelacijskom bazom podataka potrebno je poznavanje sustava za upravljanje. Neki od sustava koji se koriste za upravljanje nerelacijskom bazom podataka jesu MongoDB, Cassandra, Redis i Firebase.

1.32. Povezivanje *web*-aplikacije s bazom podataka

Za povezivanje *web*-aplikacije s bazom podataka potrebno je poznavanje veznog znakovnog niza (engl. *connection string*). Pomoću veznog znakovnog niza definiraju se upute koje omogućuju aplikaciji spajanje sx bazom podataka. Vezni znakovni niz omogućuje *web*-aplikaciji čitanje, ažuriranje, spremanje i brisanje podataka iz baze podataka. Kako bi se *web*-aplikacija povezala s bazom podataka, potrebno je stvoriti

konekciju uz pomoć veznog znakovnog niza. Konekciju je potrebno napraviti kod bilo koje baze podataka koju koristi *web*-aplikacija. Ovisno o vrsti baze podataka koju koristi *web*-aplikacija razlikuje se i vezni znakovni niz. Svaki vezni znakovni niz sastoji se od naziva poslužitelja, TCP porta, naziv baze podataka te korisničkog imena i lozinke. TCP port krajnja je točka (engl. *endpoint*) na poslužitelju koja nam omogućuje komunikaciju s poslužiteljem. U stvarnosti TCP port nekakav je broj. Kako bi baze podataka i *web*-aplikacija mogle međusobno komunicirati, potrebno je postaviti određena svojstva s obje strane, dakle i za bazu podataka i za *web*-aplikaciju. Kod baze podataka, bez obzira na to gdje se ona nalazi – lokalno na računalu ili na poslužitelju na internetu – baza podataka koristi iste postavke, s tim da se vrijednosti postavki mogu razlikovati. Nadalje, kod baze podataka potrebno je konfigurirati određeni TCP port, korisničko ime i lozinku. Važno je napomenuti da svaka *web*-aplikacija na poslužitelju mora imati jedinstven TCP port pomoću kojeg razmjenjuje informacije s drugim *web*-aplikacijama. Tako npr. unaprijed postavljeni TCP port za bazu podataka SQL Server iznosi 1433, no isti se može i promijeniti. Što se tiče *web*-aplikacije, potrebno je instalirati određeni upravljački program (engl. *driver*) za bazu podataka, a kojim se koristi *web*-aplikacija. Upravljački program razlikuje se ovisno o bazi podataka koja se koristi, što znači da ne postoji univerzalni upravljački program za sve baze podataka.

Slika 1.32.1. Komunikacija *web*-aplikacije i baze podataka

Slika 1.32.1. prikazuje proces odvijanja komunikacije između *web*-aplikacije i baze podataka, pri čemu zahtjevi uvijek dolaze s *web*-aplikacije, dok baza podataka daje odgovore na postavljene zahtjeve i iste vraća *web*-aplikaciji. Komunikacija se uspostavlja i odvija preko upravljačkog programa koji prevodi zahtjeve, odnosno odgovore, između *web*-aplikacije i baze podataka.

1.33. Sigurnost baze podataka

Za osiguravanje sigurnosti baze podataka potrebno je definirati pravila o ponašanju korisnika kao i prava korisnika koji pristupaju bazi podataka. Za razumijevanje pravila ponašanja korisnika nad bazom podataka potrebno je razumjeti pojmove autentifikacije (engl. *authentication*) i autorizacije (engl. *authorisation*). Način pomoću kojeg se korisnik predstavlja bazi podataka zovemo autentifikacija, dok autorizacija definira koju razinu prava nad bazom podataka ima autentificirani korisnik. Ovisno o bazi podataka koju koristimo postoji više načina autentifikacije korisnika, kao što su na primjer tzv. *windows* autentifikacija i autentifikacija pomoću korisničkog imena i lozinke. Kada se korisnik uspješno autentificirao, dodjeljuje mu se razina prava odnosno autorizacija. Razina prava određuje pravila ponašanja korisnika kada pristupa bazi podataka. Razlikujemo dvije osnovne kategorije razine prava. Prva kategorija jesu prava pomoću kojih korisnik može pristupiti samo određenoj bazi podataka, a druga kategorija jesu prava koja definiraju radnje koje korisnik može raditi s bazom podataka, a koje se odnose na čitanje, ažuriranje, spremanje i brisanje podataka.

1.34. Upravljanje bazom podataka

Upravljanje bazom podataka predstavlja način na koji se mogu izvršavati upiti u bazi podataka. Upit predstavlja zahtjev koji upućuje *web*-aplikacija ili razvojni inženjer prema bazi podataka radi čitanja, ažuriranja, spremanja ili brisanja podataka iz baze podataka. Razlikuju se dva načina izvršavanja upita: sinkrono i asinkrono izvršavanje. Kada se upit izvršava sinkrono to znači da se samo jedan upit može izvršavati u jednom vremenskom trenutku i da baza podataka mora pričekati rezultat upita koji se izvršava. U ovoj situaciji za vrijeme izvršavanja upita baza podataka blokira ostale korisnike tako da se bilo koji drugi upit ne može izvršavati, već se mora pričekati rezultat upita koji se trenutno izvršava kako bi se sljedeći upit mogao početi izvršavati. Prednosti sinkronog načina upravljanja upitima jest to što je jednostavno pratiti kada je koji upit završen. Međutim, nedostatak ovog načina upravljanja jest sporost izvršavanja upita jer se samo jedan upit izvršava te je potrebno sačekati njegov završetak kako bi se moglo nastaviti sa sljedećim.

Asinkrono upravljanje upitima primjenjuje se kod složenih aplikacija i kod aplikacija koje zahtijevaju simultanu obradu podataka. To je složeniji način izvršavanja upita jer omogućuje istovremeno izvršavanje više upita. Prednost asinkronog izvršavanja upita jest brzina izvršavanja. Nedostatak asinkronog izvršavanja upita u složenosti je dohvaćanje podataka. Tako je na primjer moguća situacija da je jedan upit odgovoran za kreiranje podataka, dok je istovremeno drugi upit odgovoran za dohvat podataka, a onda postoji mogućnost da se drugi upit izvrši prije prvog. U tom slučaju drugi upit neće vratiti novokreirani podatak.

PITANJA ZA PONAVLJANJE:

1. Što je baza podataka?
2. Za što koristimo bazu podataka?
3. Gdje se pohranjuju podaci u relacijskim bazama podataka?
4. Na koji način povezujemo bazu podataka i web-aplikaciju?
5. Što je autentifikacija korisnika?

1.35. Mobilne *web*-aplikacije

Mobilna aplikacija vrsta je aplikacije namijenjena za instaliranje i pokretanje na pametnim telefonima. U odnosu na *web*-aplikaciju, mobilne aplikacije uglavnom imaju umanjen skup funkcionalnosti te su dizajnirane da budu što jednostavnije za upotrebu. Modernizacija i razvoj društva potaknuo je ljude na sve veću upotrebu pametnih telefona, a to je posljedično utjecalo na razvoj popularnosti mobilnih aplikacija. Slijedom toga došlo je i do razvoja mnogih programskih jezika, razvojnih okvira i praksi u svrhu razvoja modernih mobilnih aplikacija. U početku se za izradu mobilne aplikacije upotrebljavala samo tzv. nativna vrsta aplikacije, razvijena pomoću programskog jezika koji je podržavao samo određeni operativni sustav. Tako se za razvoj mobilnih aplikacija za Android platformu koristio jedan skup alata, biblioteka i programskih jezika, dok se za razvoj iOS mobilnih aplikacija koristio potpuno drugi skup specifičnih biblioteka, programskih jezika i alata. Osim nativne aplikacije razvojem programskih jezika došlo je do pojave novih vrsta aplikacija, kao što su hibridne aplikacije i progresivne web-aplikacije. Kao što je već navedeno, nativna aplikacija razvijena je u programskom jeziku koji je podržavao samo određeni operativni sustav. Među najpopularnijim operativnim sustavima u kojima se razvijaju mobilne aplikacije su Android i iOS. Ukoliko je nativna aplikacija razvijena za operativni sustav Android, ona mora biti razvijena upotrebom programskih jezika Java ili Kotlina. Nativna aplikacija koja je razvijena za operativni sustav iOS mora biti razvijena u programskim jezicima Objective-C ili Swift. Ova vrsta mobilne aplikacije može koristiti sve funkcionalnosti operativnog sustava. Budući da svaki operativni sustav ima svoje definirane programske jezike koje podržava, ista aplikacija ne može biti podržana na više operativnih sustava. Drugim riječima, aplikacije razvijene za operativni sustav Android ne mogu se koristiti na operativnom sustavu iOS i obrnuto. Nativne aplikacije pružaju visok standard korisničkog sučelja jer razvojni inženjeri koriste izvorno korisničko sučelje operativnog sustav. Nativna aplikacija ima svoje prednosti i nedostatke. Kao prednosti mogu se istaknuti brzina razvoja aplikacije i bolja iskorištenost operativnog sustava. Nedostaci ove vrste aplikacije potreba su izrade zasebne aplikacije za svaki operativni sustav i ograničenost u programskim jezicima. Ova vrsta aplikacije koristi se za izradu mobilne aplikacije za specifičan operativni sustav. Izrada nativne aplikacije u današnje je vrijeme rijetka jer se njome ograničava korisnika u korištenju operativnog sustava. Naime, potrebe korisnika zahtijevaju da aplikacija bude dostupna na bilo kojem operativnom sustavu, što znači da razvojni timovi moraju raditi više verzija aplikacija, za svaku platformu koju žele podržati.

1.36. Hibridne *web*-aplikacije

Razvojem hibridnih aplikacija omogućena je njihova dostupnost na svim operativnim sustavima kojima se koriste korisnici. To znači da razvojni inženjer nema potrebe za poznavanjem programskih jezika za pojedine operativne sustave, već mu je dovoljno poznavati osnovne programske jezike HTML, CSS i JavaScript. Izrađena se aplikacija korištenjem gore navedenih programskih jezika, pomoću prevoditelja (engl. *compiler*), prevodi u kôd koji je razumljiv operativnom sustavu. Prednosti hibridne aplikacije brzina su i jednostavnost izrade te dostupnost aplikacije na više operativnih sustava bez potrebe da razvijamo više inačica. Kod hibridnih aplikacija ne mogu se u potpunosti iskoristiti funkcionalnosti operativnog sustava, a osim toga imaju slabe performanse, tako da to možemo promatrati kao njihove nedostatke. U situacijama kada zbog pomanjkanja vremena nije moguće razviti mobilnu aplikaciju za svaki operativni sustav pojedinačno, tada su hibridne aplikacije jedan od najboljih izbora. Nativne aplikacije imaju prednost nad hibridnim aplikacijama jer su usmjerene na operativne sustave i u potpunosti koriste funkcionalnosti operativnog sustava, za razliku od hibridne aplikacije, kod koje su funkcionalnosti operativnog sustava djelomične iskorištene.

1.37. Progresivne *web*-aplikacije

Progresivne *web*-aplikacije unaprijeđene su *web*-aplikacije. Obuhvaćaju sve karakteristike *web*-aplikacije i nativne aplikacije. Koriste responzivni dizajn prilagodljiv bilo kojem operativnom sustavu ili veličini ekrana. Na taj je način omogućena dostupnost aplikacije neovisno o operativnom sustavu. Za izradu progresivnih *web*-aplikacija koriste se programski jezici koji se koriste za izradu *web*-aplikacija. To su HTML, CSS i JavaScript. Budući da su progresivne *web*-aplikacije jednim dijelom nativne aplikacije, one također koriste funkcionalnosti uređaja kao što je upravljanje kamerom, obavijestima, geolokacijom i drugo. Progresivne *web*-aplikacije lako su dostupne te postoji nekoliko načina kako im pristupiti. Tako im se može pristupiti putem linka, pretraživanja pomoću internetskog preglednika ili instalacijom na mobilnom uređaju. Prednosti ove vrste mobilne aplikacije jesu pouzdanost, brzina i dostupnost, dok su nedostaci ograničenost, problemi sa starijim uređajima i izostanak njihove dostupnosti na svim internetskim preglednicima. Progresivne *web*-aplikacije koriste se kako bi se uštedilo vrijeme izrade mobilne aplikacije za svaki operativni sustav posebno.

1.38. Operativni sustav Android

Jedan od operativnih sustava za mobilne uređuje koji se najviše koristi zove se Android. Popularnost Android operativnog sustava i njegova velika upotreba u svakodnevnom životu utječe na razvoj mobilnih aplikacija. Android operativni sustav počinje se razvijati 2007. godine od strane Googlea. Te iste godine Google je izdao SDK (engl. *Software Development Kit*) dokumentaciju koja sadrži različite alate za lakše razvijanje mobilnih aplikacija. Od svojih početaka pa do danas razvijene su mnoge verzije Android operativnog sustava, a svaka verzija imala je poboljšan i unaprijeđen sustav. Važno je napomenuti da mobilne aplikacije ne mogu biti dostupne na svim verzijama operativnog sustava. Razlog tome su razlike između pojedinih verzija u pogledu arhitekture operativnog sustava. Prije nego što započne s razvojem mobilne aplikacije, razvojni inženjer mora odrediti verzije operativnog sustava na kojima će njegova aplikacija biti dostupna. Radi toga Google je propisao minimalnu verziju koju mobilna aplikacija mora podržavati, a u trenutku pisanja ovog priručnika radi se o Android verziji 5.0 ili većoj. Android operativni sustav razvijen je pomoću jezgre (engl. *kernel*) Linux operativnog sustava. Karakteristika tog operativnog sustava jednostavnost je otvorenog kôda, što razvojnim inženjerima omogućuje izradu i korištenje različitih vanjskih biblioteka, čime je olakšan razvoj samih mobilnih aplikacija. Razlikuju se dva primarna programska jezika koja se koriste u razvoju mobilnih aplikacija. To su Java i Kotlin. Programski jezik Java razvijen je 1995. godine i bio je prvi programski jezik za Android mobilne aplikacije. Programski jezik Kotlin razvijen je 2010. godine te se njegovom pojavom isti počinje sve više koristiti u razvoju mobilnih aplikacija. Zbog svoje popularnosti Google je Kotlin prihvatio kao standardizirani jezik za razvoj mobilnih aplikacija. Kotlin i Java slični su programski jezici, pa se tako aplikacija izrađena programskim jezikom Java može jednostavno prepisati u programski jezik Kotlin. Za oba programska jezika razvojni inženjeri razvijaju razne alate i brojne biblioteke koji olakšavaju razvoj mobilnih aplikacija. Da bi se dobio odgovor na pitanje koji programski jezik treba izabrati za izradu mobilne aplikacije, prethodno je potrebno analizirati zahtjeve aplikacije. Tako je Java programski jezik podoban kod izrade velikih i složenih aplikacija, gdje se koristi velik broj vanjskih biblioteka. Za razliku od Jave, programski jezik Kotlin ima jednostavniju sintaksu kôda, što ga čini lakšim za korištenje.

Svaku mobilnu aplikaciju nakon razvojnog proces potrebno je izgraditi (engl. *build*) jer bez tog procesa aplikacija se ne može instalirati na mobilni uređaj ni koristiti. Razliku-

jemo tri glavna sustava za izgradnju aplikacije: Gradle, Maven i Ant. Iako se za izgradnju bilo koje aplikacije može koristiti sustav za izgradnju, takav pristup u izgradnji aplikacija nije preporučljiv. Prihvatljivo je da se koristi samo jedan sustav za izgradnju aplikacije. Sustav izgradnje datoteka je koja sadrži razne postavke o načinu izgradnje aplikacije i popisuje vanjske biblioteka koje želimo koristiti u aplikaciji.

1.39. Operativni sustav iOS

Osim operativnog sustava Android postoji i operativni sustav iOS (*iPhone Operating System*). Ovaj operativni sustav razvila je tvrtka Apple. iOS je jedinstveni Appleov interni operativni sustav. To znači da aplikaciju izrađenu za operativni sustav iOS nije moguće instalirati na Android operativni sustav; može se koristiti isključivo na iOS operativnom sustavu. Za razvoj iOS aplikacije razvojnom inženjeru potreban je Apple uređaj, najnovija verzija iOS operativnog sustava, alat Xcode za izradu iOS aplikacije i poznavanje programskih jezika Objective-C ili Swift. Programski jezik Objective-C nastao je na temelju programskog jezika C tijekom 1980. godine te je to ujedno i prvi programski jezik za izradu aplikacija za operativni sustav iOS. Neke od značajnih obilježja ovog programskog jezika jesu: pouzdanost, stabilnost, kompatibilnost s C/C++ vanjskim bibliotekama, zahtjevna sintaksa kôda te korištenje kod male zajednice razvojnih inženjera. Ovaj programski jezik koristio se sve do pojave programskog jezika Swift, koji je razvijen 2014. godine. Programski jezik Swift jednostavniji je za učenje i razumijevanje te posjeduje jednostavniju sintaksu, zbog čega ima prednost pred programskim jezikom Objective-C. Osim toga, programski jezik Swift brzo je rastući programski jezik i programski jezik otvorenog kôda, koji koristi velika zajednica razvojnih inženjera. Neki od nedostataka ovog programskog jezika jesu spora brzina prevođenja i nepodržavanje C/C++ vanjskih biblioteka. Programski jezik Objective-C koristi se za izradu razvojnog okvira i kada je u pitanju niska razina (engl. low level) razvoja aplikacije. S druge strane, Swift programski jezik koristi se za izradu brzih i pouzdanih aplikacija, kao što je na primjer Facebook ili WhatsApp.

Kao i kod aplikacija izrađenih za operativni sustav Android, tako i kod aplikacija za operativni sustav iOS nakon razvojnog procesa slijedi proces izgradnje aplikacije. Jedan od sustava za izgradnju mobilnih aplikacija za operativni sustav iOS jest Cocaopods.

Cocaopods je datoteka u aplikaciji koja sadržava listu vanjskih biblioteka koje aplikacija koristi i definira način izgradnje aplikacije.

1.40. Višeplatformski pristup

Dva primarna operativna sustava koja dominiraju mobilnim aplikacija jesu Android i iOS. Da bi aplikacija bila uspješna na tržištu, najčešće je potrebno izraditi aplikaciju za oba operativna sustava. Budući da ovakav način traži puno resursa i ponekad je neisplativ, razvijeni su razvojni okviri i procesi koji olakšavaju izradu aplikacije za više operativnih sustava na način da se izrađuje aplikacija za jedan operativni sustav te se pomoću prevoditelja ista prevodi na drugi operativni sustav. Neki od popularnijih razvojnih okvira koji se danas koriste jesu React Native, Flutter, Cordova i Xamarin. Svi nabrojeni razvojni okviri nude iste mogućnosti višeplatformskog pristupa. Razvojni okviri razlikuju se po programskim jezicima kojima se koriste. Tako na primjer, ukoliko razvojni tim preferira JavaScript programski jezik, služit će se razvojnim okvirima React Native ili Cordova koji su bazirani na tom programskom jeziku. Ako pak razvojni tim više koristi C# programski jezik, tada će se koristiti Xamarin razvojnim okvirom, a oni koji žele koristiti programski jezik Dart radit će s Flutterom. U svakom slučaju, odabir razvojnog okvira ovisi o interesima razvojnog tima i njegovom poznavanju programskih jezika.

1.41. Razvojni okvir React Native

React Native razvojni je okvir namijenjen za razvoj mobilnih aplikacija za Android i iOS operativne sustave. Godine 2013. razvio ga je Facebook prema uzoru na React.js razvojni okvir. React Native koristi nativne komponente kako bi korisnička sučelja bila prikaza na isti način na oba operativna sustava. Budući da se za razvoj iOS i Android aplikacije koristi isti kôd, potrebna su znanja programskih jezika HTML, CSS i JavaScript. React Native isto kao i React.js koristi istu sintaksu pisanja kôda, što znači da prelazak iz jednog u drugi razvojni okvir ne predstavlja veliki problem. Za React Native razvijene su mnoge vanjske biblioteke koje razvojni inženjeri mogu koristiti kako bi ubrzali proces izrade aplikacije. React Native prevodi se u izvorni kôd, koji operativni

sustav razumije. To znači da aplikacija može raditi na oba operativna sustava, dakle na Androidu i iOS-u, i da su funkcionalnosti aplikacije na oba sustava iste.

1.42. Razvojni okvir Flutter

Flutter je razvio Google tijekom 2017. godine. To je razvojni okvir otvorenog kôda koji se koristi za izradu mobilnih aplikacija, *web*-aplikacija i desktop aplikacija. Osnovna ideja Flutter razvojnog okvira izrada je mobilnih aplikacija za iOS i Android operativne sustave korištenjem samo jednog kôda i jednog programskog jezika. Flutter razvojni okvir koriti programski jezik Dart. Sastoji se od SDK-a (engl. *Software Development Kit*) i biblioteka za upravljanje korisničkim komponenata koje se koriste za razvoj mobilnih aplikacija. Komponente kao što su gumbi, tekstualni predlošci, klizači i interakcijski elementi sastavni su dio biblioteke za upravljanje korisničkih komponenata. Osim toga, razvojni inženjer može razviti svoje vlastite komponente, koje se sastoje od vizualnog i logičkog dijela. Flutter svojom brzinom razvoja aplikacija i održavanja istih postavlja nove i više standarde u razvoju mobilnih aplikacija i time se ističe među konkurencijom.

1.43. Razvojni okvir Cordova

Razvojni okvir Cordova nudi mogućnosti razvoja hibridnih mobilnih aplikacija korištenjem HTML, CSS i JavaScript programskih jezika. Cordova je kao i ostali razvojni okviri baziran na principu jednog kôda koji je dostupan svim operativnim sustavima. Aplikacije koje su izrađene pomoću Cordove mogu se instalirati na mobilni uređaj i kao nativne aplikacije unatoč tomu što su izgrađene pomoću internetskih programskih jezika. Kod Cordova razvojnog okvira uz pomoć različitih vanjskih biblioteka razvojni inženjeri mogu se koristi različitim funkcionalnostima nekog uređaja, kao što su na primjer obavijesti, kamera i geolokacija. Cordova je kompatibilan sa široko korištenim alatima i integriranim razvojnim okolinama (engl. *integrated development environments*) tako da zbog toga razvojni inženjeri nisu ograničeni po pitanju izbora alata i razvojnih okolina, što znači da se za korištenje Cordova razvojnog okvira mogu koristiti alatima i razvojnim okolinama koji im najviše odgovaraju.

1.44. Razvojni okvir Xamarin

Za izradu višeplatformskih aplikacija koristi se razvojni okvir Xamarin. Razvio ga je Microsoft 2011. godine te je to jedan od najstarijih razvojnih okvira koji se koriste za izradu višeplatformskih aplikacija. Xamarin koristi programske jezike C# i XAML (engl. *Extensible Application Markup Language*). Programski jezik C# koristi se za dohvat podataka iz baze podataka i logiku aplikacije, dok se XAML koristi za izradu korisničkog sučelja. Xamarin omogućuje razvoj mobilnih aplikacija za operativne sustave iOS i Android s jednom bazom kôda. To znači da se kôd, a koji je napisan u C# i XAML, prevodi u izvorni kôd koji pojedini operativni sustav razumije.

1.45. Distribucija *web*-aplikacija

Sljedeći korak nakon razvoja aplikacije njena je distribucija. Distribucija aplikacija proces je stavljanja aplikacije na tržište, čime je omogućeno njeno korištenje. Cilj distribucije promoviranje je i korištenje same aplikacije, a pritom je potrebno proučiti kojim će se kanalima ili putem kojih platformi aplikacija distribuirati. Postoje četiri najčešća načina kojima se distribuira aplikacija. Tako postoje trgovina za aplikacije (engl. *app store*), društvene mreže, marketinške kampanje koje se temelje na e-pošti te različiti oglasnici na mobilnim uređajima i slično. Najzastupljeniji način distribucije jest pomoću trgovina za aplikacije i društvenih mreža. U današnje vrijeme svaki mobilni uređaj ima unaprijed instaliranu aplikaciju za trgovanje aplikacijama putem koje korisnik može pregledavati najnovije aplikacije i preuzimati ih na mobilni uređaj. Neke od popularnijih trgovina za aplikacije jesu Google Play, App store, Samsung Galaxy Store i Huawei AppGallery. Kako bi razvijenu aplikaciju mogli distribuirati pomoću navedenih trgovina, potrebno je zadovoljiti određene kriterije postavljene od strane pojedine trgovine. Pomoću kriterija trgovine definiraju koje su aplikacije pouzdane za korištenje, a koje nisu. Najčešći kriteriji trgovina jesu izrada police privatnosti, izrada dokumenata o načinu korištenja, logičke greške u aplikaciji, (ne)primjeren sadržaj, sumnjivi način poslovanja aplikacije i drugi. Pomoću unaprijed određenih kriterija trgovina korisnicima garantira sigurnost aplikacija koje se nalaze u trgovini. Kada se aplikacije distribuiraju korištenjem jednog ili više načina disturbacije, tada se radi o javnoj disturbaciji aplikacije. Nasuprot javnoj distribuciji aplikacije postoje i privatne ili

samostalne distribucije. Kod privatne distribucije radi se o aplikacijama koje su dizajnirane u privatne svrhe. Prednost privatne distribucije izostanak je troškova trgovine, budući da je aplikacija dostupna na nekoj mrežnoj stranici. Nedostatak ove distribucije jest u tome što korisnik ne može biti siguran u aplikaciju koju preuzima jer ne postoji treća strana koja garantira sigurnost aplikacije. Također je potrebno napomenuti da se aplikacija koja se preuzima s mrežnih stranica neće moći automatski preuzeti na mobilni uređaj jer isti ima ugrađeni mehanizam sigurnosti. Naime, kad korisnik preuzima aplikaciju s nepovjerljivog izvora ovaj mehanizam sigurnosti aktivira se i korisnika upozorava da se radi o nepovjerljivom izvoru za preuzimanje aplikacije.

PITANJA ZA PONAVLJANJE:

1. Što su to mobilne aplikacije?
2. Koje vrste mobilnih aplikacija poznajete?
3. Za koji je operativni sustav namijenjen programski jezik Kotlin?
4. Što je Maven?
5. Koje programske jezike koristi razvojni okvir Xamarin?
6. Na koje načine možemo distribuirati mobilnu aplikaciju?

1.46. Samostalne aplikacije

Prije razvoja mobilnih i *web*-aplikacija povijest aplikacija započinje razvojem aplikacija koje su se izvorno koristile na osobnim računalima. Takva vrsta aplikacija naziva se desktop aplikacija. Za instaliranje aplikacije na računalu potreban je instalacijski program koji možemo preuzeti putem interneta, USB-a (engl. *Universal Serial Bus*) ili CD-a (engl. *Compact disk*). Ne postoji univerzalan instalacijski program. To znači da je za svaki operativni sustav, Windows, iOS i Linux, potreban zaseban instalacijski program. Razlog su tome drugačija jezgra operativnog sustava i drugačiji upravljački programi koje koriste operativni sustavi.

Među desktop aplikacijama jedna je od najznačajnijih samostalna aplikacija (engl. *standalone application*). To je vrsta desktop aplikacije koja ne ovisi o drugim aplikacijama instaliranima na računalu niti joj je potreban pristup internetu. Samostalna aplikacija pruža svoje usluge lokalno, što znači da je se može koristiti jedino na računalu na kojem je instalirana te ne postoji način pristupanja takvoj aplikaciji korištenjem udaljenog pristupa. Pokreće se pomoću posebne izvršne (engl. *executable*) datoteke. Izvršna datoteka zapravo je produkt instalacijskog programa, koji je potrebno izvršiti kako bi se instalirale sve potrebne datoteke koje koristi aplikacija. U današnje vrijeme ne postoji velika potražnja za samostalnim aplikacijama jer nisu dostupne putem interneta. Osim toga, kod samostalnih aplikacija pojavom nove verzije aplikacije potrebno je staru verziju izbrisati s računala, potom preuzeti novu verziju instalacijskog programa s interneta i instalirati novu verziju aplikacije na računalu, što je također razlog zbog kojeg samostalne aplikacije nisu popularne. Uobičajeno je korištenje samostalnih aplikacija kod pružanja usluga za točno određeni problem jer nude kompaktno rješenje. Neke od samostalnih aplikacija koje danas imamo na računalima jesu kalkulator, Paint, Windows Media Player i Notepad. Svaka od navedenih aplikacija ima svoju svrhu te se točno zna za što se koristi. Samostalne aplikacije imaju svoje prednosti i nedostatke. Neke od prednosti usmjerenost su prema problemu i rješenju problema te sigurnost od vanjskih prijetnji. Kao nedostaci mogu se izdvojiti izostanak samostalne aplikacije na internetu i teško održavanje zbog novijih verzija koje je uvijek potrebno iznova instalirati. Iako je primjena samostalnih aplikacija ograničena, postoje situacije u kojima je njihova primjena nužna.

Tako se na primjer samostalne aplikacije koriste:

- kod potrebe za visokim standardom sigurnosti podataka, gdje podaci ne putuju putem mreže
- kada se žele postići visoke performanse u radu s velikim skupovima podataka
- kad imamo interne aplikacije dostupne unutar neke tvrtke koje nemaju pristup vanjskoj mreži, već su izolirane
- kad je fokus aplikacije izrada i manipulacija lokalnih datoteka.

Kako bi aplikaciju proglasili samostalnom, ona treba ispunjavati određene zahtjeve:

- radi izvan mreže
- nije dio nekog aplikacijskog paketa
- nije proširenje postojećeg procesa, već se izvodi kao posebni računalni proces
- može biti instalirana na računalo.

PITANJA ZA PONAVLJANJE:

1. Što je to samostalna aplikacija?
2. Što nam je potrebno da bismo instalirali samostalnu aplikaciju na računalo?
3. Koja nam je vrsta datoteke potrebna kako bismo pokrenuli samostalnu aplikaciju?
4. Koje su karakteristike samostalne aplikacije?

2. Infrastrukturna rješenja

U OVOM POGLAVLJU NAUČIT ĆETE:

- razlikovati vrste infrastrukture za aplikativna rješenja
- razlikovati vrste hosting usluga za aplikativna rješenja
- razlikovati vrste operativnih sustava i njihove distribucije
- razlikovati usluge i mogućnosti okruženja računarstva u oblaku
- razlikovati dostupna rješenja za kreiranje infrastrukture bez servera
- razlikovati mikroservise od monolitnih aplikacija
- razlikovati protokole za pristupanje serverskoj infrastrukturi
- analizirati mogućnosti pristupa serverskoj infrastrukturi.

2.1. Uvod u tipove infrastrukture

Kada pričamo o tipovima infrastrukture, bitno je napomenuti zašto uopće postoje infrastruktura i njezini elementi: poslužitelji, mrežni uređaji i klijenti. Prva računalna mreža ARPANET („Advanced Research Projects Agency Network") nastala je 1969. kako bi omogućila povezivanje računala u istraživačkim institucijama u Sjedinjenim Američkim Državama putem telefonskih linija. Godine 1974. izumljen je TCP („Trans mission Control Protocol") – protokol za kontrolu prijenosa koji je omogućio standardizaciju komunikacije između računala. Računalna mreža širila se te, kako bi se olakša-

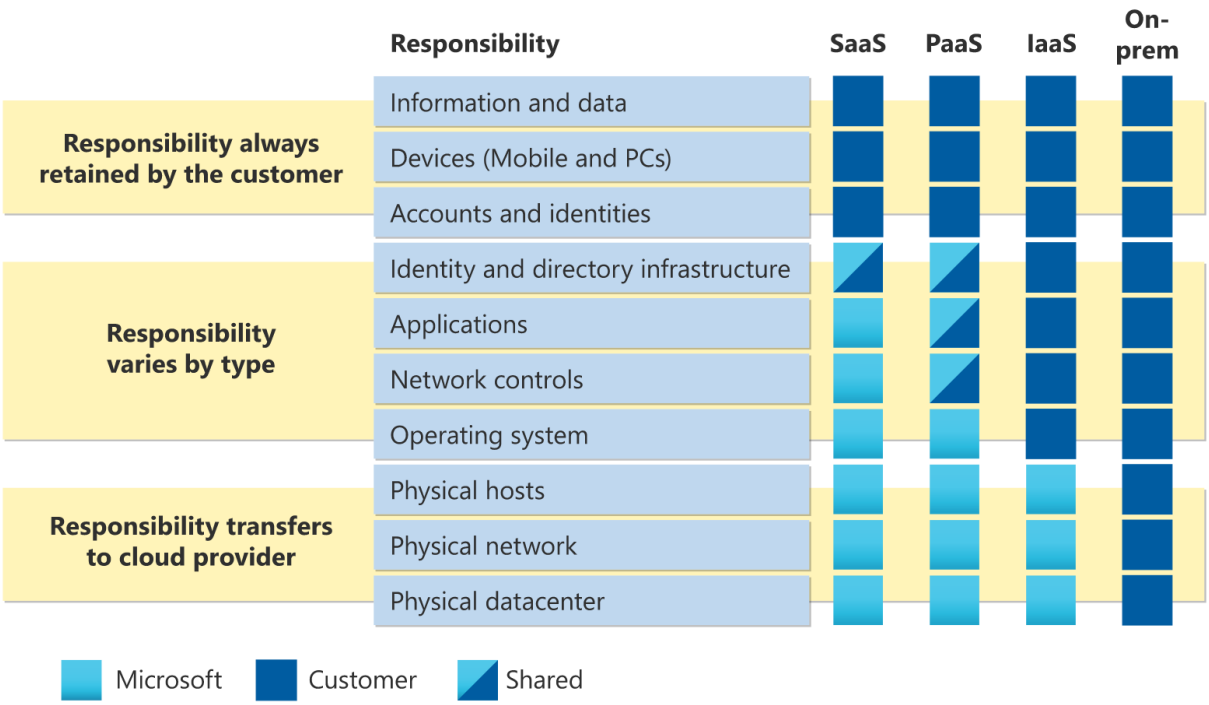
lo pamćenje adresa računala, izmišljen je DNS („Domain Name System”), protokol koji omogućuje povezivanje numeričke IP adrese računala alfanumeričkim imenom. Računalne mreže nastavile su se širiti te se njihov rast najviše ubrzao kreiranjem prvog *web*-poslužitelja. Prvi *web*-poslužitelj kreirao je britanski znanstvenik Tim Berners-Lee kako bi olakšao pronalaženje informacija na različitim računalima. Prva *web*-stranica dostupna je na linku http://info.cern.ch/hypertext/WWW/TheProject.html [1].

Za pristup internetu potrebno je bilo izgraditi infrastrukturu, a kako bi se omogućilo posluživanje više sadržaja i usluga, broj poslužitelja je rastao. Poslužitelji pružaju različite usluge kao što su e-mail, internet trgovina, forumi, društvene mreže itd. Prvi poslužitelji bili su obična računala povezana na internet koja su pružala određenu uslugu ostalim računalima povezanima na istu mrežu. Ubrzo se pojavila potreba za dodatnim resursima i dodatnim kapacitetom za pružanje usluga i kreirane su posebne računalne komponente za poslužitelje. Tvrtke su same održavale svoju infrastrukturu u vlastitim podatkovnim centrima ili iznajmljenim ormarima u podatkovnim centrima. Računarstvo u oblaku nastalo je 2006. kad je tvrtka Amazon Web Services počela nuditi infrastrukturu kao uslugu svojim klijentima. Uskoro su se pojavile i druge tvrtke sa sličnim rješenjima, pa se tako 2008. na tržištu pojavljuje Google Cloud, a 2010. Microsoft Azure usluga.

Računalna infrastruktura može se podijeliti u tri kategorije:

- *self-hosted* infrastruktura
- infrastruktura u oblaku
- hibridna infrastruktura.

Self-hosted infrastruktura lokalna je infrastruktura za koju je u potpunosti odgovorna organizacija koja je koristi. Infrastruktura u oblaku podrazumijeva infrastrukturu u kojoj se odgovornost za infrastrukturu dijeli između korisnika usluge računarstva u oblaku i pružatelja infrastrukture u oblaku ovisno o odabranom modelu. Hibridna infrastruktura sastoji se od dvije komponente: *self-hosted* infrastrukture i infrastrukture u oblaku.



Slika 2.1.1. Model odgovornosti za različite modele usluga u oblaku

Preuzeto s https://docs.microsoft.com/en-us/azure/security/fundamentals/shared-responsibility 30.6.2022.

Prilikom odlučivanja o tipu infrastrukture koji treba koristiti potrebno je uzeti u obzir zahtjeve poslovanja, zahtjeve aplikacije i postojeću infrastrukturu. Zahtjevi poslovanju sastoje se od više komponenti, kao što su usklađenost s različitim regulativama, potrebna dostupnost aplikacije i budžet – oni su različiti za svaku organizaciju. Zahtjevi aplikacije podrazumijevaju tehničke uvjete koje infrastruktura mora zadovoljiti kako bi aplikacija imala odgovarajuće performanse. Kako bi se odabrala adekvatna infrastruktura bitna je komunikacija s razvojnim timom. Preporučeno je odlučiti se za arhitekturu infrastrukture na početku projekta tijekom dizajniranja aplikacije.

2.2. Self-hosted infrastruktura

Self-hosted infrastruktura sastoji se od mrežnih uređaja i poslužitelja koji su najčešće smješteni u podatkovnim centrima zbog tehničkih zahtjeva poslužitelja. Kako bi se osigurala visoka dostupnost, podatkovni centri trebali bi imati dva potpuno odvojena dovoda struje i dvije veze prema internetu različitih pružatelja. Poslužitelji, mrežna

oprema i sustavi za pohranu podataka za poslovnu upotrebu imaju dva napajanja od kojih je svako povezano na drugi dovod struje. Zbog neefikasnosti vodiča svi električni uređaji dio energije pretvaraju u toplinu, što prilikom visokog opterećenja različitih komponenti infrastrukture može dovesti do njihova pregrijavanja. Kako bi se to spriječilo, podatkovni centri moraju imati kvalitetne sustave hlađenja. *Self-hosted* infrastruktura najčešće se sastoji od poslužitelja, mrežnih usmjernika i preklopnika za ostvarivanje mrežne povezanosti, vatrozida za kontrolu pristupa infrastrukturi i sustava za pohranu podataka.

Self-hosted infrastruktura predstavlja kapitalno ulaganje u kojem se sve komponente moraju unaprijed platiti, ali dugoročno može biti jeftinija od infrastrukture u oblaku. Važno je napomenuti da nabava komponenti *self-hosted* infrastrukture može potrajati i više mjeseci, ovisno o stanju na tržištu, što nije uvijek prihvatljivo ovisno o projektu. Zbog toga što sva odgovornost o održavanju infrastrukture pada na organizaciju potrebno je zaposliti sistemske administratore koji će je održavati. Kupljene komponente u nekom će se trenutku pokvariti, s vremenom komponente će postati prespore u odnosu na nova tehnološka dostignuća te će se broj zamjenskih dijelova smanjiti. Zbog toga je potrebno periodički mijenjati komponente. *Self-hosted* infrastruktura omogućuje organizaciji potpunu kontrolu nad podacima i njihovom rezidentnošću, dok u slučaju korištenja infrastrukture u oblaku smještaj podataka u određenoj nije moguć ukoliko pružatelj usluga nema podatkovni centar u toj zemlji. Jedan od najvećih nedostataka *self-hosted* infrastrukture nemogućnost je skaliranja resursa. Određivanje potrebne količine resursa organizacijama predstavlja veliki izazov; ako implementiraju infrastrukturu visokog kapaciteta koji nije optimalno iskorišten dok se ne započnu novi projekti, organizacija plaća resurse koje ne koristi, a ukoliko se odluče za infrastrukturu s premalim kapacitetom, čim dođe novi projekt potrebno je dodatno proširiti infrastrukturu. Potražnja za IT resursima nije konstantna i oscilira u vremenu, jedini načini da se sazna točna potreba za resursima praćenje je dostupnih metrika o aplikaciji i razini iskorištenosti različitih komponenti.

Prednosti *self-hosted* infrastrukture:

- potpuna kontrola nad infrastrukturom
- kontrola nad rezidentnosti podataka
- može biti dugoročno jeftinija od infrastrukture u oblaku.

Nedostaci *self-hosted* infrastrukture:

- dug proces nabave
- potreba za periodičkom zamjenom komponenti
- nemogućnost skaliranja.

Self-hosted infrastruktura najisplativija je za tvrtke koje imaju konstantnu potrebu za IT resursima ili striktne regulatorne zahtjeve te financijske mogućnosti za veliko ulaganje u infrastrukturu. Ako se radi o projektima u trajanju od par tjedana ili mjeseci, moguće je iskoristiti prazan kapacitet postojeće infrastrukture, ali najčešće je neisplativo za kratke projekte nabavljati hardver.

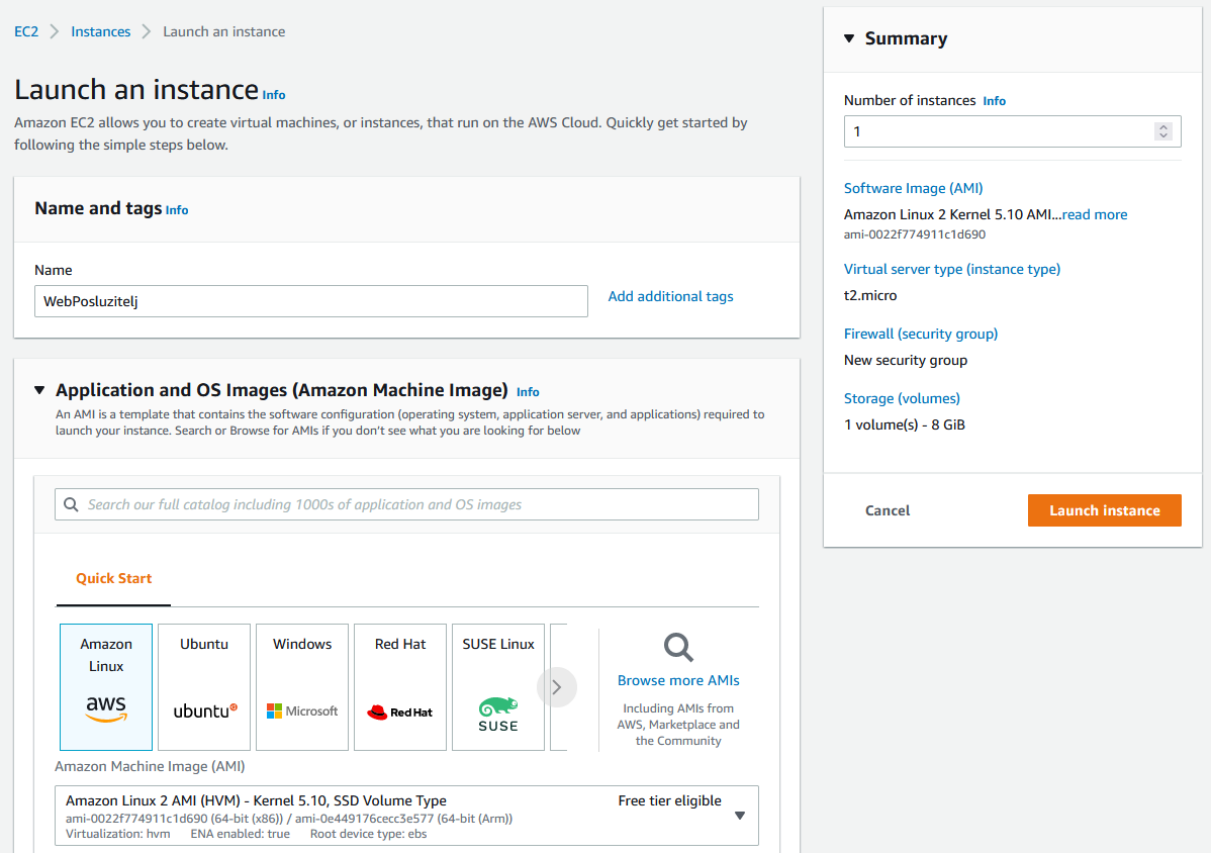
2.3. Infrastruktura u oblaku

Postoje tri modela korištenja infrastrukture u oblaku:

- **IaaS** – infrastruktura kao usluga
- **PaaS** – platforma kao usluga
- **SaaS** – softver kao usluga.

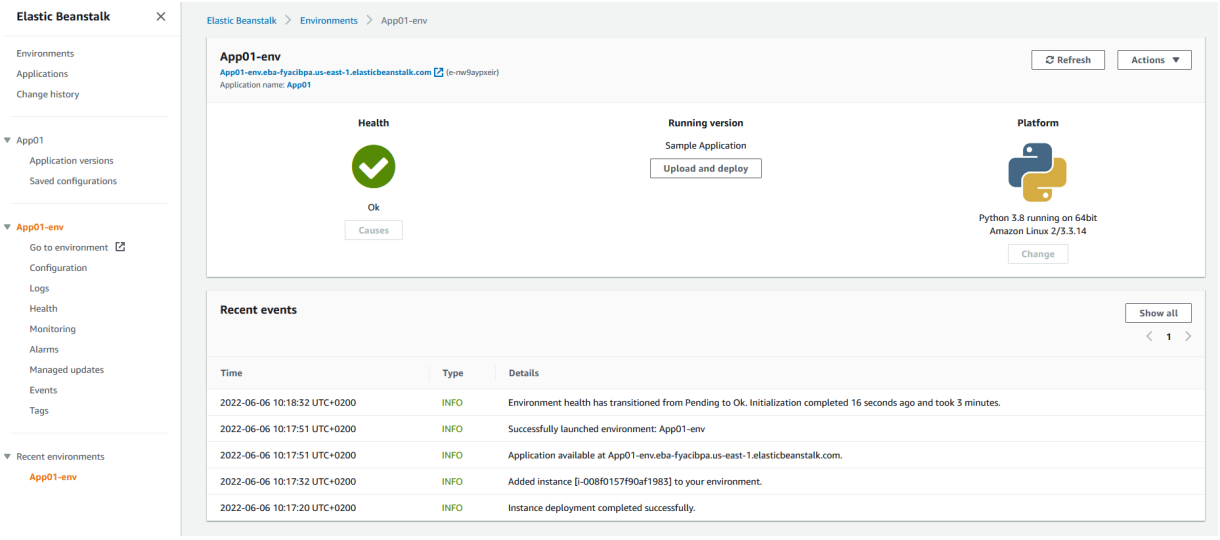
U IaaS modelu pružatelj usluga u oblaku preuzima odgovornost za brigu o fizičkim elementima infrastrukture kao što su fizički poslužitelji i mrežni uređaji. IaaS model usluge omogućio je razvoj virtualizacije i virtualizacije računalnih mreža. Korištenjem IaaS modela korisnici pokreću svoje virtualne mašine i grade virtualne mreže. Iako nisu odgovorni za fizičke resurse, korisnici su odgovorni za operativne sustave virtualnih mašina, konfiguraciju virtualnih mreža kao i sve ostale dužnosti navedene u modelu odgovornosti. Pružatelji usluga u oblaku kupuju velike količine hardvera i grade svoje podatkovne centre. Apstrakcijom fizičke infrastrukture korisnicima nude različite usluge koje koriste dostupne fizičke resurse. U uobičajenom modelu korištenja usluga infrastrukture u oblaku različiti korisnici dijelit će fizičke resurse istog poslužitelja, ali zbog logičkog razdvajanja korisnika unutar sustava neće biti svjesni jedan drugoga. Određeni pružatelji usluga u oblaku nude mogućnost zakupa dediciranih poslužitelja kako bi se zadovoljile regulatorne norme i uvjeti licenciranja pojedinih pružatelja

softvera. Primjeri IaaS usluga jesu Elastic Compute Cloud (EC2) tvrtke Amazon Web Services, S3 (engl. *Simple Storage Service*) te virtualne mašine u Azure oblaku.



Slika 2.3.1. Kreiranje EC2 instance u AWS oblaku putem *web*-sučelja

U PaaS modelu pružatelj usluga u oblaku pruža izvršno okruženje te se korisnik ne mora brinuti o operativnom sustavu virtualne mašne i njegovu instaliranju zakrpi. Korisnik odabere tip operativnog sustava koji želi koristiti za usluge poslužitelja aplikacije i učita kod aplikacije. Korisnici imaju manje kontrole nego u IaaS modelu, ali i manje odgovornosti. Primjeri PaaS usluge jesu Elastic Beanstalk u AWS-u i Azure App Services.



Slika 2.3.2. Konfiguriranje aplikacije u Elastic Beanstalk okolini

U SaaS modelu korisnici ne učitavaju svoj kôd aplikacije koju žele pokrenuti, već koriste instancu postojeće aplikacije kao što su Salesforce ili Atlassian Jira, koje održava pružatelj usluga računarstva u oblaku. Korisnici nemaju kontrolu nad operativnim sustavom niti mogu skalirati aplikaciju. Primjeri SaaS usluge jesu Zoom, Netflix ili Atlassian Jira.

Uz podjelu prema modelu korištenja postoji i podjela prema tipu oblaka:

- privatni oblak
- javni oblak
- dijeljeni oblak.

Pod pojmom privatni oblak podrazumijeva se *self-hosted* infrastruktura koja se nalazi u podatkovnom centru, ali zbog upravljačkih programa korisnici imaju iskustvo slično javnim oblacima. Usluge infrastrukture u oblaku dostupne svima od strane pružatelja usluga kao što su AWS, GCP (engl. *Google Cloud Platform*) i Azure spadaju pod javni oblak. Dijeljeni oblak zapravo je privatni oblak dijeljen između više organizacija.

Iako bi se mogao steći drugačiji dojam, resursi kojima raspolažu pružatelji usluga u oblaku također su ograničeni i u periodima visoke potražnje postoji mogućnost nedostupnosti dodatnog zakupa određenih usluga dok se kapacitet ne proširi. Kao i *self-hosted* infrastruktura infrastruktura koja pokreće oblak povremeno se kvari i

moguća je kratka nedostupnost zakupljene usluge dok se ne pokrene ponovno na drugom poslužitelju. Svaka usluga u oblaku ima garantiranu dostupnost koja je najčešće između 99,9 % i 99,99 %. U slučaju kršenja dogovorene dostupnosti korisnici najčešće dobije kredite koje mogu iskoristiti za usluge kod tog pružatelja usluga u oblaku. Ukoliko je potrebna visoko dostupna infrastruktura ili sekundarna infrastruktura u slučaju prestanka rada određenog podatkovnog centra, pružatelj usluga u oblaku pruža mehanizme postizanja visoke dostupnosti ili oporavka na drugoj lokaciji, koji moraju biti prethodno omogućeni i konfigurirani. Moguće je kombinirati usluge različitog modela korištenja. Pružatelji usluga u oblaku nude popust na korištenje određenih usluga ako se korisnici obavežu na njihovo korištenje na duži period vremena. Kako bi olakšali predviđanje troškova, pružatelji usluga u oblaku najčešće kreiraju vlastite kalkulatore cijena korištenja usluga. Kod odlučivanja o modelu korištenja usluga bitno je odvagnuti između znanja, mogućnosti organizacije i zahtjeva aplikacije, potrebe za dodatnom konfiguracijom i cijene pojedine usluge.

Infrastruktura u oblaku nudi veliku fleksibilnost organizacijama prilikom skaliranja. Pružatelji infrastrukture u oblaku omogućuju svojim korisnicima da iznimno brzo kreiraju i obrišu svoje IT resurse. Važno je napomenuti da ponekad pružatelji infrastrukture u oblaku također nemaju dovoljno kapaciteta na određenim lokacijama. To nije česta pojava, ali treba je uzeti u obzir tijekom planiranja projekata. Koristeći prednosti oblaka organizacije mogu skalirati svoje aplikacije u jako kratkom vremenskom roku. Kako bi olakšali skaliranje, pružatelji infrastrukture u oblaku omogućuju svojim korisnicima da skaliraju resurse ručno ili automatski prema određenim uvjetima.

Trošak infrastrukture u oblaku predstavlja operativni trošak koji organizacijama može olakšati financiranje IT infrastrukture jer nije potrebno plaćanje infrastrukture unaprijed, već se infrastruktura plaća mjesečno ovisno o zakupljenim uslugama i njihovoj količini. Tijekom korištenja infrastrukture u oblaku ponekad nije moguć zakup infrastrukture u istoj zemlji u kojoj se nalaze krajnji korisnici, što može biti nesukladno određenim regulativama kojima je organizacija podložna. Ako se trošak infrastrukture u oblaku ne kontrolira, organizacija može poprilično narasti. Zbog toga pružatelji usluga u oblaku nude alate za nadzor i pregled troškova. Prilikom konfiguriranja infrastrukture u oblaku preporučeno je koristiti alat za upravljanje infrastrukturom u obliku kôda kako bi sve postavke bile zapisane te kako bi se automatiziralo kreiranje okoline.

Prednosti infrastrukture u oblaku:

- fleksibilnost količine zakupljenih resursa
- skalabilnost
- brzo realiziranje infrastrukture.

Nedostaci infrastrukture u oblaku:

- nemogućnost potpune kontrole smještaja podataka
- veća potreba za upravljanje troškovima.

Infrastruktura u oblaku idealna je za tvrtke koje tek započinju sa svojim poslovanjem i hitno su im potrebni IT resursi. Korištenjem infrastrukture u oblaku tvrtke mogu korištenjem jednog pružatelja usluga kreirati globalnu infrastrukturu na svim lokacijama koje nudi pružatelj usluga u oblaku. Ukoliko *self-hosted* infrastruktura nema dovoljno kapaciteta za kratkoročne projekte ili tzv. *proof of concept* projekte, moguće ih je realizirati korištenjem infrastrukture u oblaku te nakon što se pokažu isplativima migrirati na *self-hosted* infrastrukturu.

2.4. Hibridna infrastruktura

Hibridna infrastruktura sastoji se od *self-hosted* infrastrukture i infrastrukture u oblaku – koje su međusobno povezane. Za međusobno povezivanje dviju infrastruktura potrebno je koristiti VPN (engl. *Virtual Private Network*) tunele ili direktnu vezu s pružateljem usluge u oblaku. Primjeri usluge direktnog povezivanja s pružateljem usluge u oblaku jesu AWS Direct Connect ili Azure ExpressRoute, koji omogućuju vezu koja nije putem javnog interneta već korištenjem dediciranih kablova između infrastrukture u oblaku i infrastrukture u podatkovnom centru. Hibridna infrastruktura omogućuje organizacijama kombiniranje *self-hosted* infrastrukture i infrastrukture u oblaku kako bi se iskoristile prednosti obje infrastrukture, ali to je najkompleksnija za realizaciju. Organizacija mora imati stručnjake koji su spremni upravljati *self-hosted* i infrastrukturom u oblaku. Velike tvrtke najčešće imaju hibridnu infrastrukturu te koriste više pružatelja usluga u oblaku. Hibridna infrastruktura dobra je opcija za tvrtke koje imaju nagle skokove u prometu, gdje se dodatne instance kreiraju u oblaku kad za njima

nastane potreba. Također, organizacije mogu povjerljive podatke, odnosno podatke koji ne smiju napustiti zemlju, čuvati u svojim podatkovnim centrima, a drugi dio infrastrukture u oblaku. Pojedine aplikacije kao što je Atlassian Jira mogu biti optimalne za korištenje u oblaku. Organizacije mogu koristiti infrastrukturu u oblaku za kratko testiranje, kreirajući infrastrukturu od nule prilikom svakog pokretanja testa i uništavajući je na kraju izvođenja.

PITANJA ZA PONAVLJANJE:

1. Navedi prednosti i nedostatke svih tipova infrastrukture.
2. Što treba uzetu u obzir prilikom odlučivanja o tipu infrastrukture?
3. Koje su komponente self-hosted infrastrukture?
4. Navedi tri pružatelja usluge računarstva u oblaku.
5. Koji su modeli usluge računarstva u oblaku?
6. Koji tipovi oblaka postoje?
7. Navedi primjere IaaS, PaaS i SaaS usluga.
8. Od čega se sastoji hibridna infrastruktura?
9. Koji su nedostaci hibridne infrastrukture?
10. Kako se skaliraju resursi u oblaku, a kako u self-hosted infrastrukturi?

2.5. Vrste hosting usluga

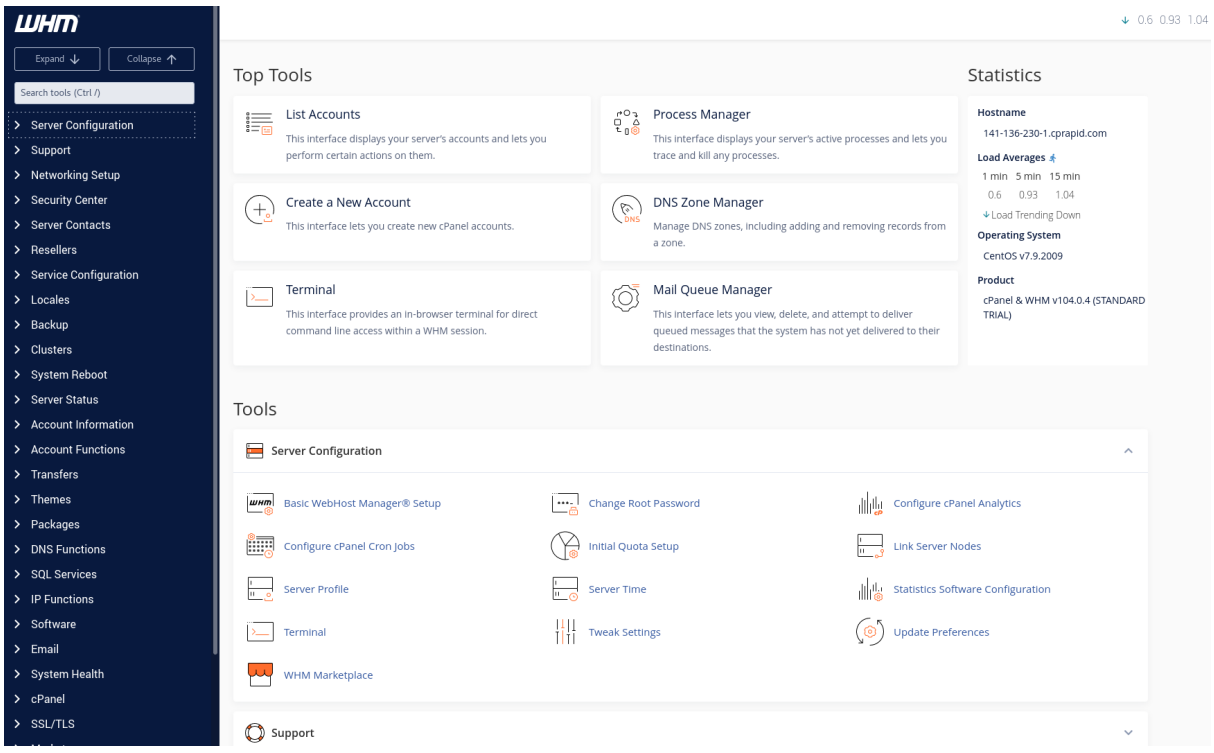
Tim Bernes-Lee uz prvi internetski poslužitelj i *web*-klijent kreirao je i HTML (engl. *HyperText Markup Language*) jezik, koji definira kako se internetska stranica prikazuje u pregledniku. Popularizacijom interneta sve više organizacija i osoba željelo je imati vlastitu internetsku stranicu. Broj internetskih stranica rastao je eksponencijalno i od samo jedne internetske stranice 1991. godine do 2000. kreirano je preko 17 milijuna internetskih stranica. Takav rast ne bi bio moguć da je svaka internetska stranica zahtijevala dedicirani poslužitelj. Pružanje usluge poslužitelja (engl. *hosting*) internetskih stranica omogućilo je onima koji nemaju tehničko znanje potrebno za održavanje poslužitelja mogućnost objavljivanja svojih internetskih stranica. Godine 1994. osnovana je tvrtka GeoCities, koja je omogućila svojim korisnicima da putem njihove platforme kreiraju besplatne internetske stranice. Kroz godine sve više tvrtki nudilo je uslugu dijeljenog hostinga. Dijeljeni hosting vrsta je hostinga u kojemu je više internetskih stranica smješteno na istom poslužitelju. WordPress je CMS (engl. *Content Management System*) i omogućuje korisnicima da korištenjem raznih dodataka (engl. *plugin*) i tema kreiraju internetske stranice i sadržaj različitih namjena na jednostavan način. Wordpress je objavljen 27.5.2003. i od tada se nezadrživo širi tržištem. Procjenjuje se da je od svih stranica kreiranih putem CMS sustava 45 % njih kreirano WordPressom. Dijeljeni hosting dugo je bio najjeftiniji način hostanja *web*-stranice zbog dijeljenja troška velikog broja *web*-stranica na jednom poslužitelju. Ukoliko su korisnici željeli veću razinu kontrole nad svojim internetskim poslužiteljem ili im je trebao poslužitelj koji će pružati drugu uslugu – a da to nije internet, već neka druga usluga kao što je poslužitelj za igre s više igrača – pružatelji usluga nudili su VPS (engl. *Virtual Private Server*). VPS usluga omogućuje korisnicima korištenje virtualne mašine koja je smještena u podatkovnom centru pružatelja usluga i dodijeljen joj je dio resursa dostupnog fizičkom poslužitelju.

Slika 2.5.1. Prikaz vrsta hostinga

Rješenja bez poslužitelja (engl. *serverless*) nov su pristup razvoju aplikacija koji se temelji na računarstvu u oblaku u kojemu se određeni programski kod izvrši prilikom dolaska zahtjeva. Prema modelu odgovornosti dijeljeni hosting spada u PaaS kategoriju usluga, VPS spada u IaaS kategoriju usluga dok rješenje bez poslužitelja može biti svrstano u PaaS kategoriju uz razlike u plaćanju i životnom ciklusu. Preporučljivo je prilikom odabira arhitekture aplikacije odabrati vrstu hostinga zbog toga što kasnija odluka može zahtijevati promjenu postojećeg koda. Performanse određenog hostinga ovisit će o prilagođenosti aplikacije tom tipu hostinga. Dijeljeni hosting idealan je za uporebu manjim organizacijama i pojedincima za hostanje vlastite internetske stranice. VPS je preporučljiv za sve vrste aplikacija osim za internetske stranice. Rješenja bez poslužitelja preporučljiva su za organizacije koje imaju potrebna znanja za kreiranje aplikacije prikladne za arhitekturu rješenja bez poslužitelja.

2.6. Dijeljena hosting usluga

Dijeljeni hosting najčešće je jeftiniji od ostalih vrsta hostinga zbog ranije spomenutog dijeljenja resursa s ostalim korisnicima istog poslužitelja. Korisnici imaju kontrolu



Slika 2.6.1. cPanel prikaz pružatelja hostinga

samo nad svojom internetskom stranicom. Mogućnosti i performanse određene su mjesečnim planom na koji je korisnik pretplaćen. Jedina mogućnost skaliranja pretplata je na mjesečni plan s više resursa. Dijeljeni hosting najjednostavniji je za konfiguriranje i korištenje. Može biti korišten za hostanje *web*-stranica. U odabranom paketu korisnicima će biti dodijeljeni brzina propusnosti, prostor na disku te razina podrške. Ukoliko korisnici dijeljenog hostinga imaju visoke zahtjeve za performansama, moguće im je dodijeliti dedicirani poslužitelj i ta se usluga naziva dedicirani hosting.

Tijekom korištenja ove mogućnosti korisnici mogu dobiti prava glavnog administratora nad dediciranim poslužiteljem, ovisno o tome žele li upravljani ili neupravljani dedicirani poslužitelj. Upravljani poslužitelji kontrolirani su od strane pružatelja usluga i korisnik upravlja svojim internetskim stranicama alatom kao što je npr. cPanel. Alat cPanel grafičko je rješenje za upravljanje hostanja internetske stanice. Ovisno o instaliranim modulima i proširenjima cPanel i slična rješenja mogu biti korištena za upravljanje jednom stranicom, više stranica ili cijelim poslužiteljem.

Prednosti:

- jednostavnost korištenja
- niska cijena.

Nedostaci:

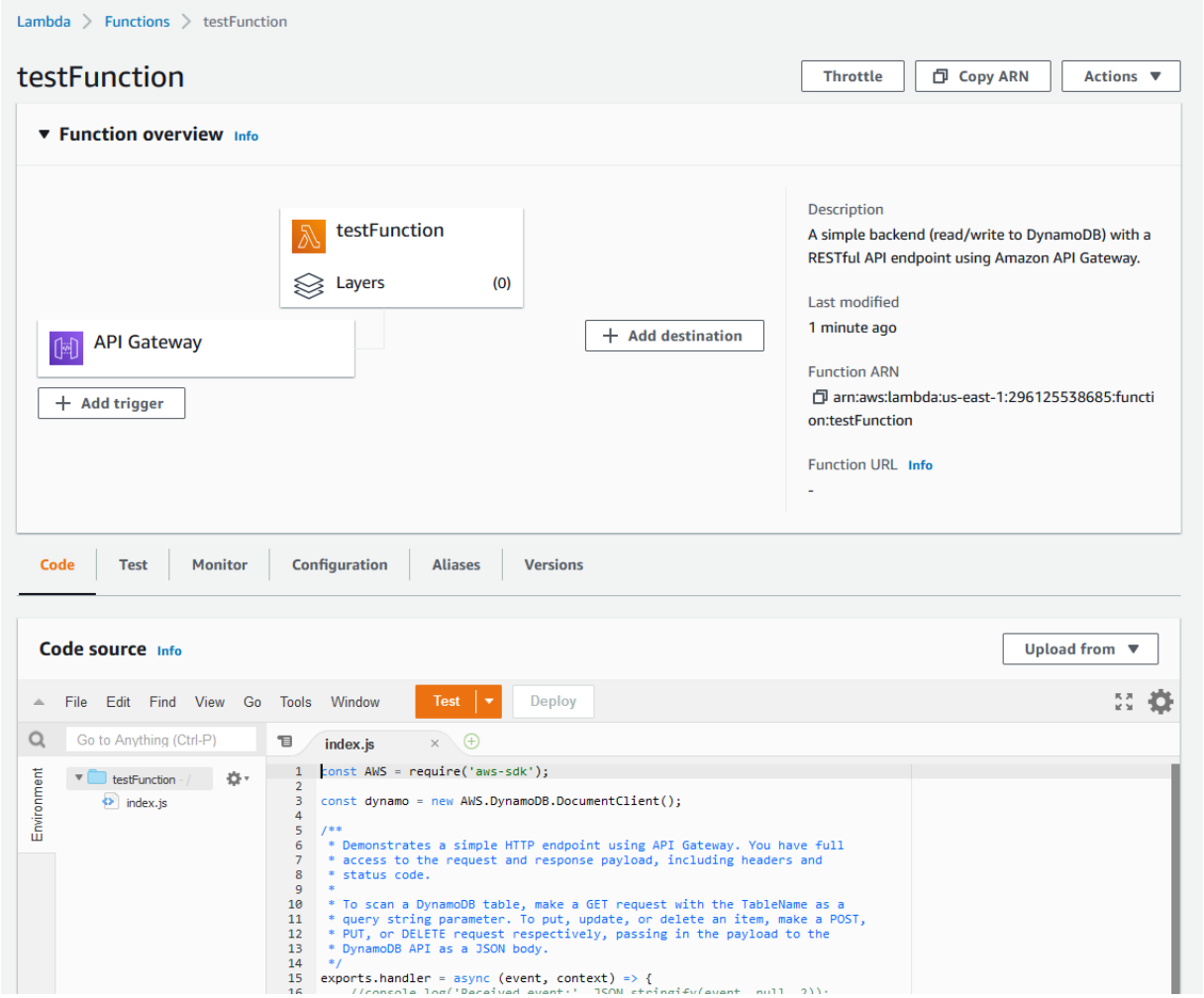
- niske performanse
- sigurnost ovisi o pružatelju usluga
- limitirana skalabilnost
- niska mogućnost konfiguracija OS-a.

2.7. Rješenje bez poslužitelja

Arhitektura rješenja bez poslužitelja, ukoliko je korištena za odgovarajuću svrhu, može pružiti iznimne performanse i uštedu nasuprot tradicionalnih rješenja zbog svog jedinstvenog modela naplaćivanja i načina rada. Za svaki zahtjev kreira se jedno izvođenje zadanog koda i naplaćuju se samo resursi korišteni za pojedini zahtjev. AWS u svom portfelju nudi Lambda uslugu koja je odličan primjer usluge funkcije implementirane kao rješenje bez poslužitelja. Funkcije pružaju korisnicima mogućnost specificiranja koda koji žele izvršiti na zahtjev klijenta. Lambda se u trenutku pisanja teksta naplaćuje prema sljedećim kriterijima i cijenama za US East regiju:

- broj zahtjeva $0,20 za milijun zahtjeva
- radna memorija/vrijeme izvođenja za x86 arhitekturu $0.0000166667 po GB-sekundi
- privremena pohrana/vrijeme izvođenja $0.0000000309 za svaku GB-sekundu
- prijenos podataka.

Gigabit-sekunda predstavlja korištenje 1 GB memorije u periodu od jedne sekunde. Arhitektura rješenja bez poslužitelja omogućuje aplikaciji da u jednom trenutku opslužuje jednog korisnika, a nakon samo par trenutaka 1000 korisnika pružanjem jednakih performansi upravo zbog odvojenog izvođenja za svaki zahtjev korisnika. Takav način rada zahtijeva posebnu arhitekturu internetske stranice ili aplikacije te migracije postojećih aplikacija na takav način rada, što nije jednostavno. Aplikacije na takvoj arhitekturi prilikom prvog pokretanja ili pokretanja nakon određenog vremena mogu imati loše performanse zbog potrebe pripreme infrastrukture od strane pružatelja usluge u oblaku. Njihovo izvođenje određeno je vremenskim limitom koji ovisi o pružatelju usluga, sukladno tome npr. AWS Lambda ograničena je na izvođenje unutar 15 minuta, dok usluga Azure Functions u naprednijim planovima nema takvo ograničenje, ali zahtjev je da funkcija pokrenuta putem HTTP zahtjeva mora odgovoriti unutar 230 sekundi.



Slika 2.7.1. Konfiguriranje AWS Lambda funkcije

Programski kod sadržan u funkciji može biti pokrenut na različite okidače koji ovise o pružatelju usluga računarstva u oblaku, mogu komunicirati s ostalim sustavima i dohvaćati podatke iz raznih baza te pokretati druge funkcije. Odličan primjer korištenja arhitekture rješenja bez poslužitelja jest kreiranje funkcije koja nakon uspješnog pohranjivanja slike uslugom za pohranu u oblaku obradi sliku i generira sličicu (engl. *thumbnail*) slike. Ukoliko bi takva aplikacija bila kreirana standardnom arhitekturom, sve vrijeme dok korisnici ne pohranjuju nove slike infrastruktura ne bi bila korištena, ali bi njeni resursi bili zakupljeni. Jedno pokretanje funkcije naziva se invokacijom.

Ukoliko internetske stranice ne sadrže elemente koje je potrebno renderirati na strani poslužitelja, može ih se nazvati statičkim internetskim stranicama i moguće je njihov HTML i CSS kôd pohraniti kod nekih od pružatelja pohrane kao što je AWS S3 (engl. *Simple Storage Service*) ili u git repozitoriju koji podržava funkcionalnost statičkih in-

ternetskih stranica. Kreiranje takvih stranica olakšano je generatorima statičkih stranica te raznim radnim okvirima kao što su npr. Angular ili ReactJS. Statički hosting može biti iznimno jeftin ili čak i besplatan, ovisno o pružatelju usluge. Kako bi iskoristili benefite statičkog hostinga i izvođenja kôda na rješenjima bez poslužitelja, promet se usmjerava ovisno o elementima od kojih se sastoji stranica te se korisniku prikaže statička internetska stranica s dinamičkim dijelovima koji su dohvaćeni kao rezultat izvođenja jedne ili više funkcija.

Prednosti:

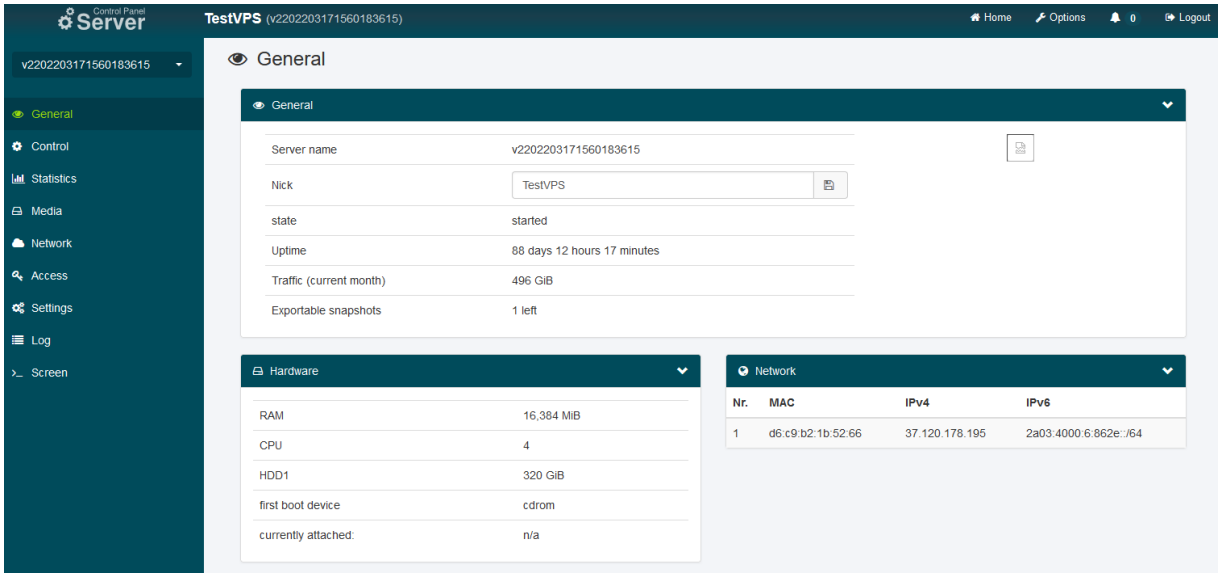
- jednostavno skaliranje
- globalno distribuirana korištenjem infrastrukture pružatelja usluga u oblaku
- integracija s ostalim uslugama pružatelja usluga u oblaku
- odgovornost samo za programski kod i konfiguraciju funkcije koja se izvodi.

Nedostaci:

- tehnička zahtjevnost
- zahtijeva posebnu arhitekturu aplikacija
- komplicirano računanje cijene
- limitirano vrijeme izvođenja i ograničeni resursi.

2.8. Virtualni privatni poslužitelj

VPS predstavlja najprilagodljiviji tip hostinga, ali zbog toga korisnik ima najveći stupanj odgovornosti. Cijena ovisi o tipu/veličini virtualne mašine (skraćeno VM) koju korisnik izabere. Skalira se na način da se promijeni tip VM-a, što uglavnom zahtijeva period nedostupnosti trajanja, kao da se poslužitelj ponovno pokrenuo, ili dodavanjem dodatnih virtualnih mašina i balansiranjem prometa. Naplaćuje se po satu ili mjesečno i cijena varira ovisno o tipu VM-a. VPS može biti korišten za hostanje svih vrsta aplikacija i internetskih stranica. Nakon zakupa VPS-a korisnici će od pružatelja usluga dobiti podatke za pristup poslužitelju i podatke za pristup upravljačkoj ploči putem koje mogu upravljati svojim poslužiteljem. Način pristupa poslužitelju ovisit će o njegovu operativnom sustavu (skraćeno OS) te kasnijoj korisničkoj konfiguraciji.



Slika 2.8.1. VPS upravljačka ploča netcup pružatelja usluga

Mogućnosti upravljačkog panela ovise o pružatelju usluga. Neke od čestih mogućnosti jesu ponovno pokretanje VPS-a, proširenje pohrane, promjena veličine VM-a i instalacija OS-a. Kod korištenja VPS-a korisnik je sam zadužen za brigu o sigurnosnim kopijama svog VM-a kao i o sigurnosti OS-a i aplikacija.

Prednosti:

- potpuna kontrola nad OS-om
- dobre performanse
- cijena ovisi o odabranom modelu
- prikladan svim tipovima aplikacija.

Nedostaci:

- zahtijeva kontinuirano održavanje
- zahtijeva poznavanje OS-a
- sigurnost ovisi o znanju korisnika i konfiguraciji.

PITANJA ZA PONAVLJANJE:

1. Koja tri modela hostinga poznajete?
2. O čemu ovisi način pristupa VPS-u?
3. Koja je razlika između dijeljenog i dediciranog hostinga?
4. Koji je tip hostinga najjeftiniji?
5. Koji tip hostinga biste preporučili svom prijatelju za njegov blog?
6. Što je to statička web-stranica?
7. O čemu ovise načini pokretanja funkcije na rješenju bez poslužitelja?
8. Koji tip hostinga ima najniže performanse?
9. O čemu će ovisiti performanse aplikacije u odnosu na tip hostinga?
10. Koji tip hostinga najbrže skalira?

2.9. Operativni sustavi

Prva računala nisu imala OS (operativni sustav), već su početkom 50-ih godina komponente računala fizički prespajane kako bi se podesila zadana funkcija. Ubrzo nakon toga uvedene su bušene kartice koje su omogućile učitavanje koda u računala. Računala tada nisu bila dostupna široj javnosti zbog njihove astronomske cijene. Zbog visoke cijene potrebno je bilo optimalno iskoristiti vrijeme dostupno za izračune i zbog toga su kreirani sustavi za serijsko izvođenje više poslova. Godine 1956. kreiran je prvi operacijski sustav GMOS od strane General Motorsa za IBM-ov *mainframe*. Operacijski sustavi kreirani su kako programi ne bi morali sadržavati logiku koja se ponavlja, kao što je upravljanje hardverom, upravljanje procesima, upravljanje datotekama itd.

80-ih godina Bill Gates kupio je DOS (engl. *Disk Operating System*) i uz određene izmjene prodavao ga proizvođačima računala pod nazivom MS-DOS (*MicroSoft Disk Operating System*), koji su ih prodavali krajnjim korisnicima. Apple je uz svoje računalo Apple Macintosh korisnicima pružao i Mac OS s GUI-jem (*grafičkim sučeljem*). Kako bi mu konkurirao, Microsoft je kreirao operativni sustav Windows, koji je u svom početku predstavljao GUI za MS-DOS. Windows Server OS prvi je put izdan 2003. u izdanju Windows Server 2003.

Unix OS prvi je put predstavljen 1973. Zbog *antitrust* tužbe Bell Labsa UNIX je distribuiran akademskim i istraživačkim institucijama za cijenu medija i poštarine te nije bio dostupan javnosti kao proizvod. Godine 1975. kôd je podijeljen u obliku knjige. Zbog nepostojanja službene podrške korisnici su sami rješavali *bugove* i dijelili rješenja međusobno. Ubrzo su kreirane i druge verzije UNIX-a. Godine 1983. *antitrust* tužba riješena je i UNIX je licenciran i krenuo se prodavati kao proizvod. Linus Torvalds koristio je MINIX OS tijekom studiranja na Sveučilišta u Helsinkiju. Zbog svoje frustracije s licencom MINIX OS-a koja ga je ograničavala na edukacijsku upotrebu kreirao je i izdao Linux 17.9.1991. godine, kreirajući vlastitu jezgru OS-a (engl. *kernel*).

Jezgra OS-a obavlja kritične funkcije za OS, kao što su:

- upravljanje radnom memorijom
- upravljanje procesima

- upravljanje hardverom
- upravljanje vremenom
- međuprocesna komunikacija
- upravljanje prekidima
- kontrola prava pristupa.

Zbog otvorenosti koda, odnosno licenci koje dozvoljavaju pristup i ponovno korištenje koda, kreirane su mnoge različite distribucije Linux i Unix OS-ova. Kako bi se standardiziralo ponašanje raznih verzija Unixa i Linuxa, kreirana je POSIX obitelj standarda od strane IEEE-a (engl. *Institute of Electrical and Electronics Engineers*).

Razni operativni sustavi koji postoje mogu zbuniti administratore i razvojne inženjere prilikom odabira OS-a. Prilikom odabira OS-a bitno je razmotriti kompatibilnost aplikacija s određenim OS-om. Cijena korištenja određenog OS-a ovisit će o potrebnim resursima za sam OS i cijeni licence. Windows i Linux OS moguće je podesiti do zadovoljavajuće razine sigurnosti ukoliko se koristi verzija OS-a podržana od strane proizvođača.

Prilikom odabira OS-a moguće je odabrati OS koji podržava, odnosno ima instalirano grafičko sučelje. U korporativnim okruženjima nije uobičajeno instalirati grafičko sučelje na poslužitelje. Instalacija grafičkog sučelja zahtijeva instalaciju dodatnih komponenti koje mogu sadržavati ranjivosti i povećavaju površinu napada (engl. *attack surface*) na poslužitelj. Instalacijom grafičkog sučelja raste potrebna količina procesorskog vremena, količine radne memorije i diskovnog prostora. Unatoč tome dio administratora odlučuje se za instalaciju GUI-a zbog jednostavnosti konfiguracije.

Odabir OS-a poslužitelja uvelike ovisi o aplikaciji koja će na njemu biti smještena, postojećoj infrastrukturi i stupnju poznavanje određene distribucije ili OS-a tima koji će biti zadužen za njegovu administraciju. No neke aplikacije zbog programskog jezika ili neke druge ovisnosti ne podržavaju sve operacijske sustave.

2.10. Operativni sustav Linux

Neki od najčešće korištenih poslužiteljskih OS-ova jesu RHEL (Red Hat Enterprise Linux), Ubuntu Server i CentOS. RHEL je komercijalna *open-source* Linux distribucija. Za njegovo korištenje plaća se godišnja licenca prema broju instalacija ili prema broju fizičkih procesora, odnosno prema broju procesora virtualizacijskog poslužitelja. Licence namijenjene za produkcijsku upotrebu uključuju podršku od tvrtke Red Hat. Tvrtka Red Hat razvija RHEL za komercijalne korisnike. Ubuntu je Linux distribucija bazirana na Debianu. Debian Linux distribucija razvijana je od strane Debian projekta. Ubuntu koristi kôd Debiana te dodaje vlastite mogućnosti. Korištenje Ubuntua besplatno je, ali se podrška plaća. Sigurnosne zakrpe besplatno su dostupne 5 godina od izdavanja LTS (*Long Term Support*) izdanja, uz plaćenu podršku dostupne su 10 godina. CentOS (*Community Enterprise Operating System*) bio je besplatna Linux distribucija otvorenog kôda. Bio je kompatibilan sa svojim *upstream* izvorom RHEL-om. *Upstream* označava smjer distribuiranja programskog kôda. Nakon što bi izašlo novo izdanje RHEL-a iz njegova izvornog kôda, Red Hat brend bio bi zamijenjen CentOS-ovim i bilo bi kreirano novo izdanje CentOS-a. Nažalost, u prosincu 2020. Red Hat je prekinuo daljnji razvoj CentOS-a. Kao alternativu CentOS-u osnivač CentOS projekta kreirao je Rocky Linux distribuciju. Red Hat je CentOS zamijenio CentOS Stream OS-om. CentOS Stream više se ne izdaje na temelju prethodno izdane verzije RHEL-a, već su nove mogućnosti prvo dodane u CentOS Stream. CentOS Stream sada predstavlja *upstream* izvor RHEL-a. Rocky Linux i CentOS Stream kao i njihov prethodnik CentOS besplatni su za korištenje, ali se plaća podrška.

Prije uvođenja sustava za upravljanje paketima *softvare* se distribuirao u arhivama ili kao izvorni kôd. Pakiranje softvera u pakete omogućuje jednostavno instaliranje, nadogradnju i uklanjanje paketa.

RPM i dpkg osnovni su alati za upravljanje paketima, no najčešće se ne koriste direktno, već pomoću alata kao što su yum ili apt. yum i RPM koriste se za RHEL i CentOS, a apt i dpkg za Ubuntu. yum i apt u pozadini komuniciraju s RPM-om ili dpkg-om te olakšavaju njihovo korištenje dodatnim mogućnostima kao što je automatsko preuzimanje paketa potrebnih za rad odabranih paketa. Konfiguracijske datoteke različitih aplikacija najčešće se nalazi u etc mapi pune putanje /etc. Paketi se najčešće

preuzimaju iz repozitorija dostupnih na Internetu. Prilikom dodavanja dodatnog repozitorija potrebno je provjeriti radi li se doista o validnom repozitoriju kako ne bismo instalirali zloćudan softver. Za RHEL 8 i CentOS 8 yum je zamijenjen dnf-om. Kod Linux i Unix OS-ova zakrpe se primjenjuju na razini pojedinačnog paketa. Kod svih operacijskih sustava potrebno je osigurati vrijeme nedostupnosti poslužitelja za instalaciju ažuriranja. Ažuriranja ponekad sadržavaju greške, pa je prije njihove instalacije potrebno pročitati tzv. „release notes" i testirati ažuriranje na manjem broju poslužitelja.

Linux operativnim sustavima moguće je pristupiti lokalno ili udaljenim pristupom pomoću SSH, RDP ili VNC protokola. SSH protokol omogućuje udaljeni pristup putem konzole, dok RDP i VNC protokol omogućuju udaljeni pristup kroz grafičko sučelje.

sudo je program koji omogućuje administratorima poslužitelja da određenim korisnicima omoguće pokretanje određenih ili svih naredbi kao root ili neki drugi korisnik. root korisnik ima pravo pokrenuti sve programe, pristupiti i izmijeniti sve datoteke.

Prava pristupa svakoj datoteci i mapi u Linux OS-u određena su vlasništvom i pridruženim pravima. Svakoj datoteci i mapi dodijeljena su prava korisniku koji je vlasnik datoteke, grupi koja je vlasnik datoteke i ostalim korisnicima. Moguća prava jesu read, write i execute. Prava su zapisana kao set zastavica, što u dekadskom sustavu odgovara zbroju zastavica read = 4, write = 2 i execute = 1. U ispisu su često prezentirana prvim slovom r, w i x. Na slici 2.10.1. vidljiv je rezultat naredbe ls s argumentima a i l kako bi se prikazale i skrivene datoteke dužim formatom ispisa. Datoteku testfile1 može pokrenuti samo korisnik root. Svi korisnici mogu pročitati sadržaj svih datoteka. Pravo zapisivanja na mapi korisniku omogućuje da dodaje, uklanja ili preimenuje datoteke i podmape unutar te mape. Pravo izvršavanja na mapi omogućuje korisniku da uđe u mapu i pristupi njezinu sadržaju. Prava čitanja mape korisniku omogućuju da prikaže sadržaj mape.

```
[root@141-136-230-1 temp]# ls -al
total 4
drwxrwxr-x   5 student student  105 Jun 22 21:29 .
drwx------. 17 student student 4096 Jun 22 21:29 ..
drwxrwxr-x   2 root    root       6 Jun 22 21:29 testdir1
drwxrwxr-x   2 student student    6 Jun 22 21:29 testdir2
drwxrwxr-x   2 student student    6 Jun 22 21:29 testdir3
-rwxrw-r--   1 root    root       0 Jun 22 21:29 testfile1
-rw-rw-r--   1 student student    0 Jun 22 21:29 testfile2
-rw-rw-r--   1 student student    0 Jun 22 21:29 testfile3
```

Slika 2.10.1. Ispis prava pristupa

iptables je alat za upravljanje pravilima filtriranja IP paketa u Linux kernelu. iptables se sastoji od tri grupe tablica: MANGLE, NAT i FILTERING tablicama. MANGLE tablica služi za modifikaciju paketa. NAT tablica služi za translaciju IP adresa i prosljeđivanje prometa s jednog porta na drugi (engl. *port forwarding*). FILTERING tablica služi za filtriranje paketa prema definiranim pravilima. Svaka od tablica ima više lanaca koji predstavljaju skup pravila. FILTERING tablica ima *chainove* INPUT, OUTPUT i FORWARD. INPUT *chain* sadrži pravila za pakete koji dolaze, OUTPUT za pakete koji dolaze, dok FORWARD tablica sadrži pravila za pakete koji će biti proslijeđeni.

SELinux i **AppArmor** sustavi su za primjenu obavezne kontrole pristupa (engl. *Mandatory Access Control*, skraćeno MAC). SELinux je predinstaliran na RHEL OS, dok je AppArmor predinstaliran na Ubuntu. SELinux definira prava za međusobnu interakciju korisnika, aplikacija, procesa i datoteka. AppArmor ograničava kojim resursima određena aplikacija može pristupiti, ali ne ograničava korisnika. SELinux i AppArmor mogu biti u potpunosti onemogućeni (engl. *disabled*), raditi u načinu u kojemu ne brane nedozvoljeni pristup, već ga samo bilježe (engl. *permissive*) i zapravo braniti nedozvoljeni pristup (engl. *enforcing*). Prilikom konfiguracije aplikacije može doći do određenih grešaka u kojima aplikacija ne može pristupiti određenim datotekama, u takvim situacijama rješenje nije potpuno ugasiti zaštitu, već istražiti zašto je greška nastala i ukoliko je potrebno omogućiti pristup.

Prilikom istraživanja grešaka log datoteke mogu pružiti vrijedne informacije jer su u njima zabilježeni razni događaji sustava. Log datoteke nalaze se u direktoriju /var/log ukoliko nije drugačije konfigurirano. Ako nema dovoljno informacija u logovima, moguće je povećati razinu njihove detaljnosti za pojedinačnu aplikaciju ukoliko to sama aplikacija podržava.

2.11. Operativni sustav Windows

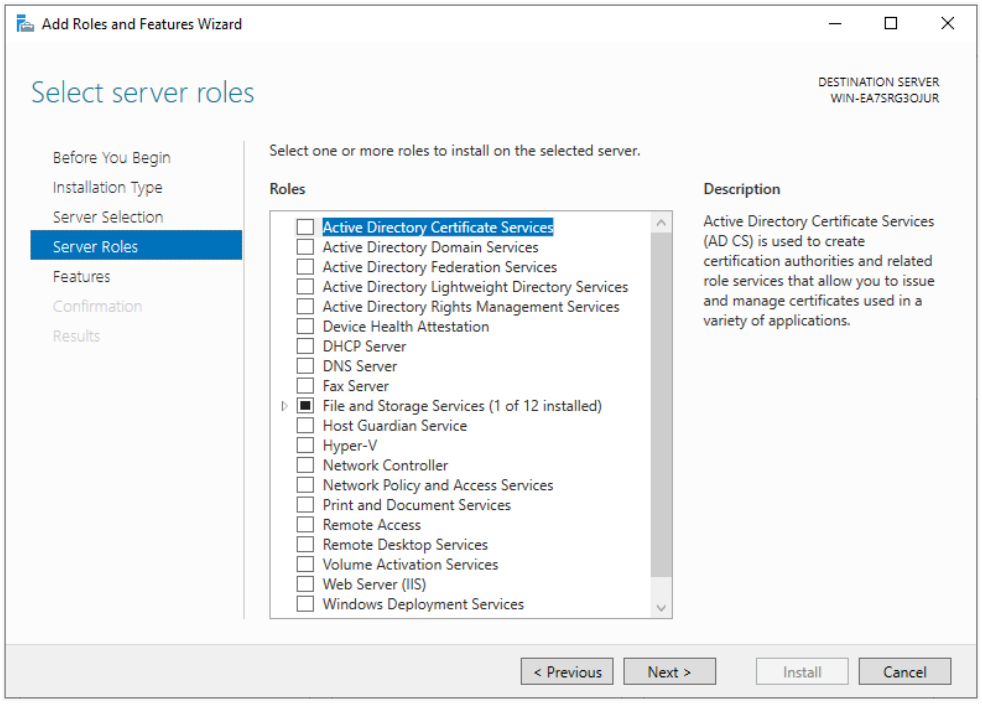
Operativni sustav Windows Server dolazi u tri izdanja:

- Essentials
- Standard
- Datacenter.

Izdanje Essentials preporučeno je za male tvrtke do 25 korisnika i 50 uređaja i ne zahtijeva CAL (engl. *Client Access Licenses*) licence. CAL licence prodaju se u dva izdanja prema broju korisnika ili broju uređaja koji pristupaju poslužitelju. Standard i Datacenter izdanja licenciraju se prema broju procesorskih jezgri, ali također zahtijevaju CAL licence. Moguće je licencirati VM-ove licenciranjem virtualizacijskih poslužitelja Datacenter licencom. Za korištenje softverski definirane mreže i softverski definirane pohrane podataka mogućnosti potrebno je koristiti Datacenter licencu. Softverski definirana mreža i pohrana podataka predstavljaju virtualizaciju mreže i prostora za pohranu podataka na način da se resursi dostupni na više poslužitelja grupiraju zajedno te se korisnici izoliraju jedan od drugoga na način da svi koriste resurse iz iste grupe resursa, ali nisu svjesni jedni drugih. Velike organizacije ne kupuju licence zasebno, već s Microsoftom potpisuju ugovor u kojem se definira broj licenci i poslužitelja. Najčešće se koristi Standard i Datacenter izdanje.

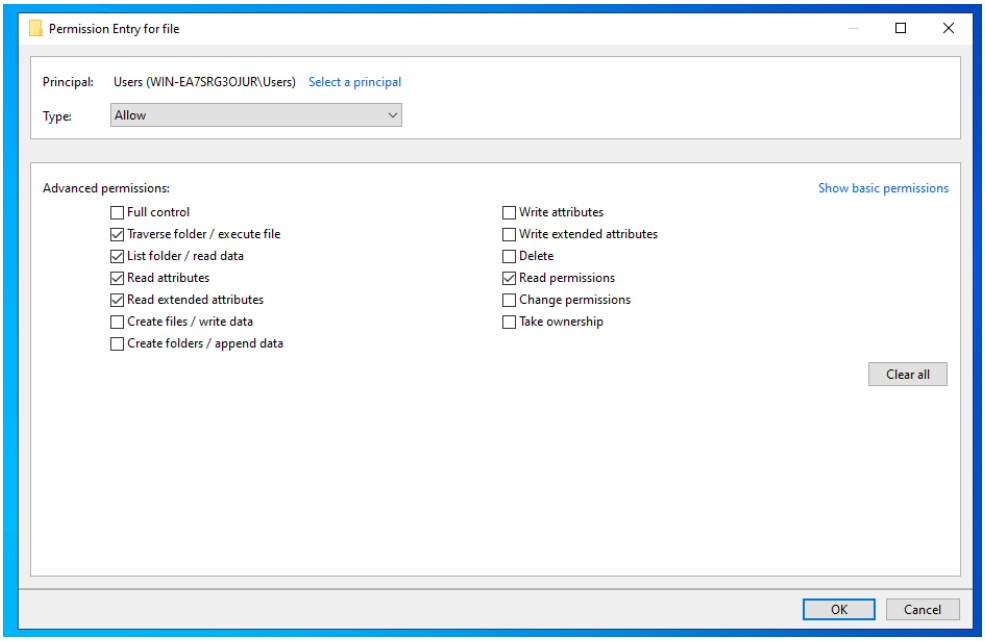
Role (uloge) Windows poslužitelja predstavljaju set aplikacija potrebnih da poslužitelj obavlja određenu funkciju. Jedna od rola je i Hyper-V rola, koja omogućuje poslužitelju da obavlja funkciju virtualizacijskog poslužitelja. Osim rola mogu se proširiti i mogućnosti poslužitelja (engl. *features*). Sve role i mogućnosti nisu predinstalirane kako bi se smanjila veličina OS-a i broj instaliranih komponenti. Role i mogućnosti mogu se instalirati kroz komandnu liniju (engl. *Command Line Interface* ili skraćeno CLI) ili grafičkim sučeljem.

Web Server (IIS) rola je čijom instalacijom poslužitelj dobiva mogućnost posluživanja mrežnih stranica. Sama rola sastoji se od više opcionalnih usluga kojima se mogu dodati razne funkcionalnosti kao što su različiti tipovi autentifikacije i mogućnosti bilježenja.



Slika 2.11.1. Instalacija dodatnih rola na poslužitelj

Prilikom instalacije OS-a poslužitelja administratori mogu odabrati instalaciju poslužitelja sa ili bez grafičkog sučelja. Prilikom instalacije Windows Server OS-a verzije s grafičkim sučeljem označene su s „Desktop Experience" u nazivu. Verzija bez grafičkog sučelja naziva se Server Core. Određene role i mogućnosti nisu podržane u Server Core verziji Windows Server OS-a.



Slika 2.11.2. Prikaz napredne mogućnosti dodijele NTFS prava

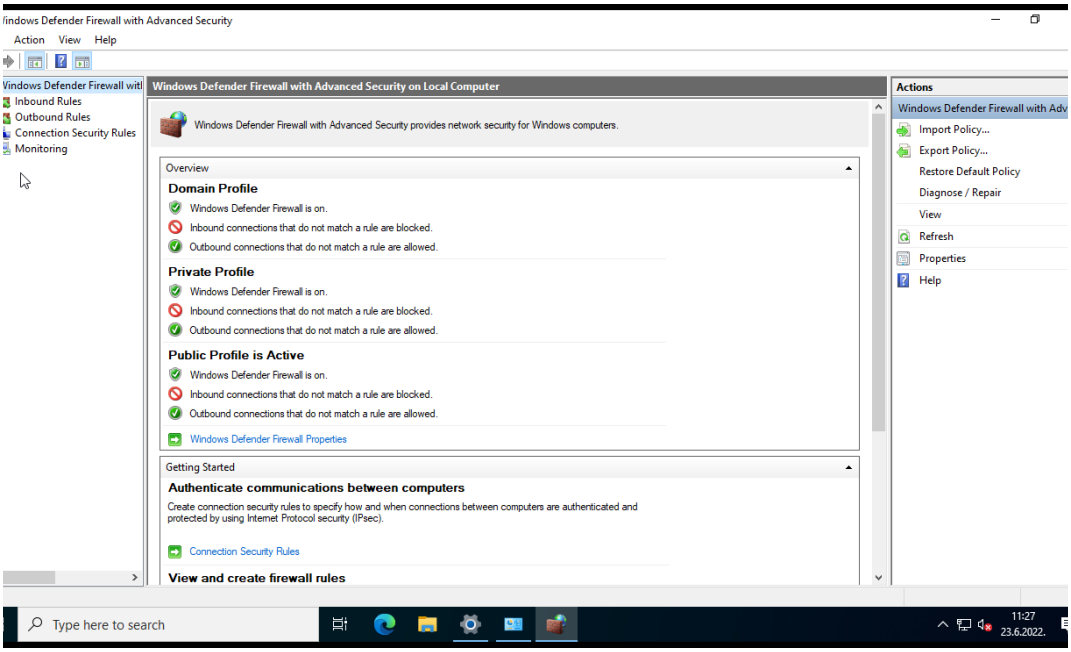
Kod Windows OS-a zakrpe za sve komponente OS-a primjenjuju se u kumulativnoj zakrpi (engl. *cumulative update*). Zakrpe je moguće instalirati na svaki poslužitelj zasebno ili koristiti neku od mogućnosti za upravljanje instalacijom zakrpi.

Prava pristupa datotekama na Windows OS-u dodjeljuju se na razini datotečnog sustava. Najčešće korišteni datotečni sustav za Windows OS je NTFS. NTFS omogućuje precizno dodjeljivanje prava naprednim mogućnostima kao što je vidljivo na slici 2.11.2.

Osnovna prava koja se mogu dodijeliti jesu:

- potpuna kontrola
- pravo izmjene
- pravo čitanja i izvršavanja
- pravo čitanja
- pravo zapisa.

Moguće je dodijeliti i zabranu određene akcije. Sva prava zbrajaju se u listu akcija koje određeni korisnik može poduzeti, ali ukoliko je eksplicitno određena akcija zabranjena, korisnik je neće moći poduzeti, odnosno zabrana je uvijek jača od dozvole.



Slika 2.11.3. Windows Firewall

Prava se također mogu nasljeđivati kroz strukturu mape od mape roditelja do podmape. Prava pristupa mogu biti dodijeljena i grupama radi lakšeg upravljanja pravima. Na Windows OS-u ne postoji root korisnik, ali postoji Administrator korisnik, koji je ugrađeni račun za administraciju. Njegova lozinka podešava se tijekom instalacije OS-a. Administrativna prava moguće je dodijeliti ostalim korisnicima dodavanjem u grupu Administrators. Windows OS ima više zadanih grupa koje mogu poslužiti za delegaciju određenih prava nad samim poslužiteljem.

Alat Windows Firewall služi za kontrolu mrežnog prometa poslužitelja. Za razliku od iptablesa može kreirati i IPSec tunele. Pravila za upravljanje prometom podijeljena su u dolazna i odlazna pravila. Pravila se mogu primijeniti za jedan ili više profila. Profil određuje tip mreže na koju je računalo spojeno. Prilikom prvog spajanja na mrežu OS će upitati korisnika želi li tretirati mrežu na koju se spojio kao privatnu ili javnu. Domenski profil automatski se primjenjuje kada je mrežni adapter u domenskoj mreži. Zadane postavke privatnog profila dozvoljavaju promet po više portova. Kao i kod prava pristupa na datoteke, pravila se zbrajaju i eksplicitna zabrana uvijek „pobjeđuje" dozvolu.

2.12. Vrste sučelja operativnih sustava

OS može, ali ne mora nužno imati grafičko sučelje i podržavati konfiguraciju navigiranjem kroz razne izbornike i odabirom željenih postavki. Windows poslužitelje bez grafičkog sučelja moguće je konfigurirati kroz grafičko sučelje administratorske radne stanice alatima koji to podržavaju.

U velikim organizacijama koje imaju tisuće poslužitelja OS se najčešće konfigurira alatima za upravljanje konfiguracijom kao što su Ansible i Puppet, a ne izvršavanjem pojedinačnih naredbi. U svim organizacijama inženjeri pišu skripte koje omogućuju automatizaciju određenih procesa i smanjuju mogućnost greške. Bash je naredbena ljuska koja se koristi za konfiguraciju Linux sustava putem komandne linije. PowerShell je rješenje za automatizaciju za Windows OS, koje se sastoji od naredbene ljuske, skriptnog jezika i okvira za upravljanje konfiguracijom. Jedna od najvećih razlika PowerShella i Basha jest u tome što je PowerShell objektno orijentiran te može raditi s objektima .NET programskog jezika. Za dobivanje pomoći tijekom korištenja

PowerShella korisna je naredba **`Get-Help`** te za otkrivanje naredbi **`Get-Command`** i **`Show-Command`**. Prilikom korištenja basha pomoć je dostupna korištenjem man naredbe. Nakon same naredbe često se koriste dodatne zastavice i argumenti kako bi se promijenilo zadano ponašanje naredbe. Zastavica predstavlja prekidač koji je omogućen ili nije, dok argument sadrži neku od vrijednosti. Primjer naredbe u bashu bio bi **`chmod -R 777 /tmp.`** chmod predstavlja naredbu za mijenjanje prava pristupa datoteci, zastavica **`-R`** omogućuje rekurzivnost, argument **`777`** predstavlja koja prava želimo postaviti, a **`/tmp`** je putanja na koju se naredba primjenjuje.

PITANJA ZA PONAVLJANJE:

1. Poveži OS i alat za instalaciju paketa.

CentOS 8	yum
Ubuntu	apt
CentOS 7	dnf

2. Koja je razlika između SELinuxa i AppArmora?
3. Pomoću kojeg bi alata ograničio mogućnost mrežnog pristupa poslužitelju?
4. U kojoj se mapi najčešće nalazi konfiguracija paketa?
5. Kroz grafičko sučelje i pomoću konzole na Windows Server OS-u prikaži omogućene mogućnosti.
6. Po čemu se razlikuje proces instalirana sigurnosnih zakrpa na Windows i Linux poslužiteljima?
7. Kako konfigurirati Windows OS poslužitelj bez grafičkog sučelja?
8. Što je RDP?
9. Koji je ugrađeni mehanizam za upravljanje konfiguracijom Win dows poslužitelja dodanih u domenu?
10. Kako na Windows OS-u ograničiti mogućnosti mrežnog pristupa poslužitelju?

2.13. Infrastruktura u oblaku

Današnji pristup postavljanju aplikacija u rad bio bi nezamisliv bez infrastrukture u oblaku. Organizacije mogu svoje aplikacije rasporediti po cijelom svijetu i skoro odmah dobiti zatražene resurse. Upravo brzina dobivanja zatraženih resursa privlači velike i male organizacije infrastrukturi u oblaku. Infrastruktura u oblaku nezamisliva je bez Amazon Web Servicesa (AWS). AWS je nastao 2006. tako što je Amazon primijetio da koristi mali postotak ukupnog kapaciteta svoje IT infrastrukture. Kako bi ga iskoristio, odlučio ga je ponuditi u zakup. Prva usluga koju je nudio bio je S3 (*Simple Storage Service*). S3 omogućuje objektni pristup pohrani podataka putem interneta. Ubrzo je ponudio i drugu uslugu EC2 (*Elastic Compute*). EC2 omogućuje korisnicima da kreiraju svoj VM smješten u AWS-ovoj infrastrukturi putem interneta. AWS je nastavio rasti i trenutno nudi svojim korisnicima preko 100 različitih usluga. Amazonovu AWS-u u natjecanju za tržište pružanja usluga u oblaku 2008. pridružio se Googleov GCP i 2010. Microsoftova Azure platforma. Uz navedene glavne pružatelje usluga u oblaku pojavili su se i manji pružatelji koje ne treba zanemariti.

Kako bi organizacije imale uvid u potrebe za skaliranjem aplikacije, moraju pratiti njen rad. Za praćenje rada aplikacija i opterećenosti poslužitelja iznimno su korisne metrike.

Metrike mogu biti izložene za skupljanje na razini infrastrukture od strane pružatelja infrastrukture, na razini virtualne mašine ili kontejnera ili na razini aplikacije. Za skupljanje metrika brinu se rješenja kao što su prometheus, InfluxDB ili usluga pružatelja usluga u oblaku.

Skupljanje metrika na razini aplikacije zahtijeva da aplikacija izloži metrike ili uz nju bude pokrenut agent. Na temelju skupljenih metrika mogu se kreirati kontrolne ploče ili alarmi za praćenje stanja aplikacija i infrastrukture. Bez skupljenih metrika organizacije ne bi imale dovoljno podataka za praćenje rada aplikacija i analizu potrebe za skaliranjem. Pružatelji infrastrukture u oblaku nude i usluge automatskog skaliranja prema odabranim metrikama. Svaka usluga u oblaku ima svoj SLA (engl. *Service Level Agreement*), koji definira njenu dostupnost. U slučaju kršenja SLA-a pružatelji usluga u oblaku najčešće svojim korisnicima dodijele dodatne kredite za korištenje usluga.

2.14. Glavni pružatelji usluga u oblaku

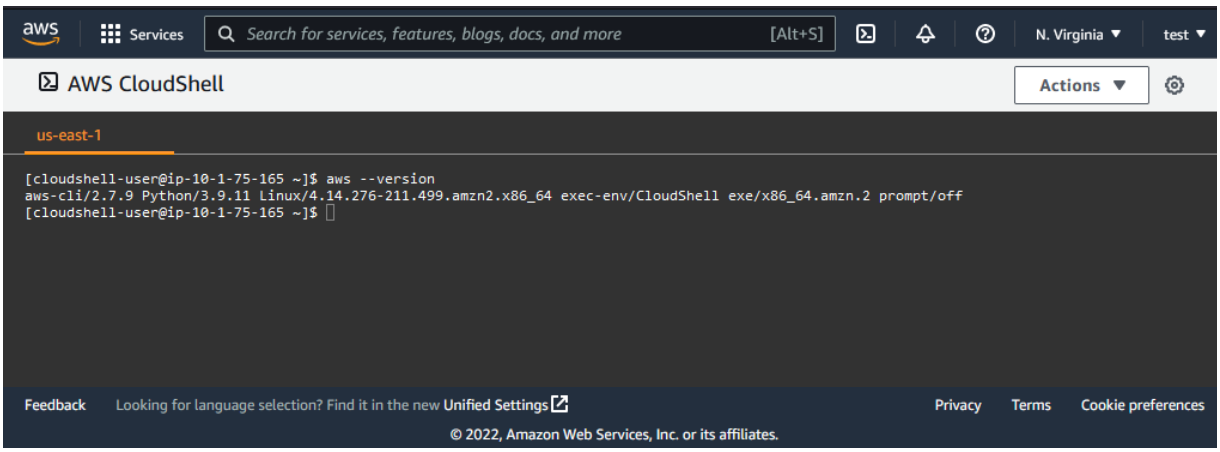
Interakcija s pružateljima usluga u oblaku može se odvijati na dva načina: pomoću internetskog portala ili pomoću API-ja. Alati za upravljanje infrastrukturom i alati za upravljanje infrastrukturom u obliku kôda koriste API pozive u pozadini kako bi izvršili zadani zadatak. Internetske adrese glavnih portala za upravljanje infrastrukturom u oblaku jesu:

- https://aws.amazon.com
- https://portal.azure.com
- https://cloud.google.com.

Svaki pružatelj usluga u oblaku nudi različite mogućnosti autentifikacije, dokumentirane u njihovoj službenoj dokumentaciji. Također, u službenoj dokumentaciji moguće je pronaći upute za korištenje API-ja ili direktno ili pomoći alata za upravljanjem infrastrukturom u oblaku. Dokumentacija često sadrži i primjere konfiguracije za razne usluge. Dokumentaciju je moguće pronaći na linkovima:

- https://docs.aws.amazon.com/
- https://docs.microsoft.com/en-us/azure/
- https://cloud.google.com/docs.

Proces instalacije alata za upravljanje infrastrukturom u oblaku ovisit će i o OS-u radne stanice. Alatima za upravljanje infrastrukturom moguće je pristupiti i kroz internetski portal bez instalacije dodatnih alata na radnu stanicu, kao što je vidljivo na slici 2.14.1.



Slika 2.14.1. AWS CloudShell

Cijene i način obračuna korištenja usluga u oblaku razlikuju se između pružatelja usluga te između različitih regija u kojima su resursi smješteni. Obvezivanjem na korištenje resursa na duži period, koji je najčešće minimalno 1 godinu, moguće je dobiti popust na određene usluge. Kako bi se usporedile cijene različitih pružatelja usluga, potrebno je prvo definirati željenu infrastrukturu kako bi se mogla kreirati procjena troška za svakog od pružatelja usluga u oblaku. U tu svrhu svaki od glavnih pružatelja usluga u oblaku nudi kalkulator troškova. U praksi kalkulatori troškova mogu zahtijevati razne unose koji nisu poznati dok aplikacija ne uđe u produkciju i najviše što mogu ponuditi jest okvirna procjena.

Nakon kreiranja same infrastrukture njezin trenutni i predviđeni trošak moguće je pronaći koristeći usluge za analizu troškova i pregled računa. Na računima za prošle mjesece moguće je pronaći detaljan opis usluga koje su plaćene.

2.15. Ostali pružatelji usluga u oblaku

AWS, GCP i Azure nisu jedini pružatelji usluga u oblaku. Uz njih postoje i manji pružatelji usluga kao što su Digital Ocean i Hetzner.

Digital Ocean je pružatelj usluga u oblaku i nudi usluge infrastrukture u oblaku. Osnovan je 2011. U trenutku pisanja ovog teksta ima podatkovne centre u Sjedinjenim Američkim Državama, Nizozemskoj, Singapuru, Engleskoj, Njemačkoj, Kanadi i Indiji. Nudi razne IaaS I PaaS usluge. Hetzner je pružatelj usluga infrastrukture u oblaku i kolokacije u podatkovnom centru. Kolokacija u podatkovnom centru predstavlja iznajmljivanje određenog dijela podatkovnog centra klijentima, gdje onda oni donose vlastite poslužitelje. Hetznerovi podatkovni centri u trenutku pisanja teksta smješteni su u Njemačkoj, Finskoj i Sjedinjenim Američkim Državama. Nudi IaaS usluge i usluge web-hostinga. Neki od hrvatskih pružatelja infrastrukture u oblaku su Hrvatski Telekom i SETCOR.

Ponuđači infrastrukture mogu svoje usluge nuditi globalno ili lokalno. Globalni ponuđači usluga nude svoje usluge u više zemalja i regija, dok lokalni nude svoje usluge u samo jednoj državi. Zbog svoje veličine globalni ponuđači usluga imaju puno veću ponu-

du usluga od onih lokalnih, mogu garantirati veću dostupnost i performanse. Lokalni ponuđači usluga ponekad mogu biti cjenovno jeftiniji, ali to ovisi o arhitekturi infrastrukture.

2.16. Tipovi usluga u oblaku

Svaki od pružatelja infrastrukture u oblaku svoje usluge kategorizira radi lakšeg snalaženja. Kategorije usluga glavnih pružatelja prikazane su na slici 2.16.1.

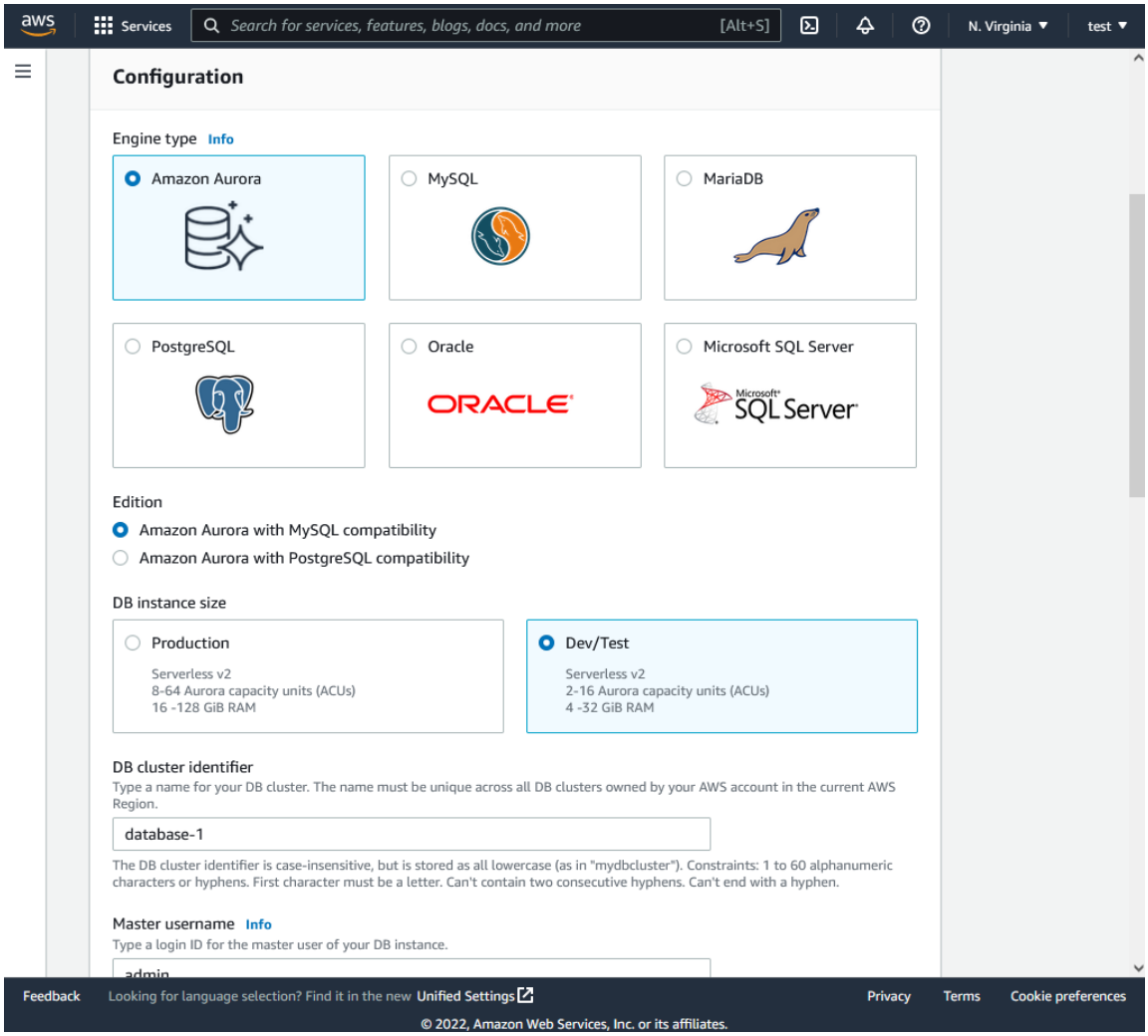
AWS	Azure	GCP
Analytics	AI + Machine Learning	Analytics
Application Integration	Analytics	Application Integration
AR & VR	Blockchain	Artificial Intelligence
AWS Cost Management	Compute	CI/CD
Blockchain	Containers	Compute
Business Applications	Databases	Databases
Compute	Developer Tools	Management
Containers	DevOps	Networking
Customer Enablement	Identity	Operations
Database	Integration	Other Google products
Developer Tools	Internet of Things	Security
End User Computing	IT & Management Tools	Serverless
Front-end Web & Mobile	Media	Storage
Game Development	Migration	Tools
Internet Of Things	Mixed Reality	
Machine Learning	Monitoring & Diagnostics	
Management & Governance	Networking	
Media Services	Security	
Migration & Transfer	Storage	
Networking & Content Delivery	Web	
Quantum Technologies		
Robotics		
Satellite		
Security, Identity & Compliance		
Storage		

Slika 2.16.1. Kategorizacija servisa prema pružateljima infrastrukture u oblaku

Svaki poslužitelj ima svoju kategorizaciju, iako su poprilično slične. Za svako aplikativno rješenje u oblaku potrebno je imati određenu *compute* uslugu, odnosno uslugu za procesiranje, određenu uslugu za pohranu podataka odnosno *storage* i/ili *database* uslugu te je potreban *networking*, odnosno mrežno povezati različite usluge. Potrebno je osigurati aplikaciju i infrastrukturu u oblaku koristeći *security* usluge i usluge za upravljanje identitetima i sigurnošću pristupa. Naposljetku, tijekom rada aplikacije potrebno je pratiti aplikaciju i infrastrukturu koristeći neku od monitoring usluga, odnosno usluga za nadzor. Za svaku uslugu odgovornost za različite komponente podijeljena je prema modelu korištenja infrastrukture u oblaku (IaaS, PaaS i SaaS) opisanom u jednom od prethodnih poglavlja. Upravljanje infrastrukturom u oblaku podijeljeno je na dvije razine: upravljanje samim oblakom i upravljanje podacima, takozvani data i management plane. Kreiranje virtualne mašine i upravljanje njenim postavkama vrši se na management razini, dok se podacima i konfiguracijom same mašine upravlja na podatkovnoj razini.

2.17. Mogućnosti usluga baza podataka u oblaku

Svaki pružatelj usluga računarstva u oblaku nudi vlastite usluge baza podataka u oblaku. Te usluge najčešću uključuju upravljanje SQL baze (MySQL, PostgreSQL i Microsoft SQL Server), NoSQL baze, Redis, Memcached i MongoDB kompatibilnu uslugu. Uz navedene baze pružatelji usluga u oblaku imaju rješenja za tzv. „big data" platforme i vlastite baze podataka kreirane na način da iskorištavaju puno više mogućnosti infrastrukture u oblaku u pozadini, ali su i dalje kompatibilne sa standardnim načinima pristupa. Primjeri vlastitih baza jesu AWS-ova Aurora, Azureov CosmosDB i GCP-ov AlloyDB. Svaka od baza ima svoj set mogućnosti, konfiguracijskih opcija i način naplaćivanja. Određene usluge u potpunosti su upravljane od strane pružatelja usluga u oblaku i visokom dostupnošću i redundancijom upravljaju oni, dok je za određene usluge korisnik sam odgovoran.



Slika 2.17.1. Prikaz kreiranja instance Amazon Aurora baze podataka

U AWS oblaku standardne baze podataka spadaju pod RDS (Relational Database Service) uslugu. RDS baze imaju sljedeće mogućnosti:

- implementacija u multizonalnoj konfiguraciji za visoku dostupnost
- automatsko prebacivanje u slučaju kvara u sekundarnu zonu
- kreiranje kopija za čitanje
- automatsko kreiranje sigurnosnih kopija
- automatska nadogradnja verzije baze
- automatsko skaliranje prostora za pohranu
- enkripcija u mirovanju
- enkripcija u tranzitu
- izolacija na razini računalne mreže.

2.18. Mogućnosti pohrane podataka u oblaku

Svaki pružatelj usluga računarstva u oblaku svojim korisnicima nudi razne načine pohrane podataka u oblaku. Postoje tri načina za pristup podacima:

- Pristup podacima na razini bloka podataka koji odgovara pristupu podacima kao fizički spojenom tvrdom disku gdje korisnik sam podešava datotečni sustav.
- Pristup podacima na razini datoteke gdje korisnik ne podešava datotečni sustav, već pristupa podacima nekim od *file-sharing* protokola.
- Pristup na razini objekta. Uslugama koje služe za pohranu podataka kao objektima najčešće se pristupa putem HTTP protokola.

Primjer pristupa podacima na razini bloka podatka jest AWS-ov EBS (*Elastic Block Storage*), gdje se virtualni diskovi dodaju virtualnim mašinama. Primjer pristupa podacima na razini datoteke jest pristupanje mrežnom dijeljenom direktoriju koji održava pružatelj usluga u oblaku kao što je Azure Files usluga. Primjer pristupa podacima na razini objekta jest AWS-ov S3. Svaka od usluga pohrane podataka ima mogućnosti za kreiranje sigurnosnih kopija podataka, njihovu replikaciju i upravljanje životnim ciklusom podataka.

2.19. Mogućnosti virtualnih računala u oblaku

Ono što razdvaja pružatelje VPS usluge i pružatelje virtualnih mašina u oblaku jest međusobna integracija s ostalim servisima, koju nude pružatelji infrastrukture u oblaku. Svi pružatelji usluga u oblaku nude mogućnost odabira veličine i tipa virtualne mašine prema zadanim tipovima kako bi se bolje prilagodili zahtjevima raznih aplikacija. Diskovi dodani virtualnim mašinama mogu biti smješteni direktno na poslužiteljima koji ih pokreću uz visoke performanse, ali nepostojanje otpornosti na pad poslužitelja ili u sustavima za pohranu podataka pružatelja infrastrukture u oblaku uz mogućnost ponovnog dodavanja diska na drugu virtualnu mašinu. Disk može u jednom trenutku biti dodan isključivo na jedan VM. Diskovi smješteni na poslužitelj izgubljeni su nakon brisanja VM-a. Pružatelji usluga nude mogućnost upravljanja smještajem VM-a kroz

grupe za smještanje za osiguranje od ispada pojedinačnog poslužitelja, mrežnog ormara ili zone dostupnosti. Također, zbog integracija VM-a s ostalim uslugama VM-ovi mogu biti dodani u setove za posebne namjene. U slučaju kvara jednog od VM-ova u setu on može biti automatski zamijenjen. VM-ovi su jedan od servisa koji podržavaju rezervaciju za smanjenje cijene uz dugoročnu obavezu.

Spot instance predstavljaju virtualne mašine kratkog životnog vijeka s niskom cijenom. Idealne su za poslove koji se mogu prekinuti. Imaju nisku cijenu jer ih pružatelji usluga u oblaku prodaju kako bi prodali nezakupljeni kapacitet. Ukoliko se stanje promijeni i pružatelju zatreba taj prostor, uz kratkoročnu najavu može obrisati korisnikovu spot instancu.

2.20. Mogućnosti statičnog hostinga u oblaku

Za statički hosting *web*-stranica AWS korisnicima nudi mogućnost korištenja S3 usluge za smještaj kôda internetske stranice. Za implementaciju https protokola pristupa statičkoj internet stranici smještenoj u S3 usluzi potrebno je koristiti Amazon CloudFront. Slično kao i AWS, GCP nudi korisnicima mogućnost statičkog hostinga pomoću njihove Cloud Storage usluge. Također, pristup putem https protokola zahtijeva dodatnu konfiguraciju putem usmjerivača prometa. Azure svojim korisnicima nudi statički hosting kroz uslugu Azure Static Web Apps i kroz Azure Storage Account uslugu. Azure Static Web Apps usluga nudi dodatne mogućnosti za integraciju s Azure DevOps uslugom i besplatne SSL certifikate. Glavni pružatelji usluga u oblaku pružaju sličan set osnovnih usluga. Nije neuobičajeno da jednostavne aplikacije mogu biti smještene kod bilo kojeg pružatelja usluga infrastrukture u oblaku. Uz pomoć zahtjeva aplikacije i projekta, dokumentacije pružatelja usluga i kalkulatora cijene stručnjaci moraju odrediti najbolje rješenje. Ponekad će se iskoristiti manje optimalno rješenje koje se bolje uklapa u cjelokupnu infrastrukturu organizacije.

PITANJA ZA PONAVLJANJE:

1. Kako je nastao AWS?
2. Navedi glavne pružatelje usluga u oblaku.
3. Kako organizacije mogu prepoznati potrebu za skaliranjem aplikacija u oblaku?
4. Koja je razlika između globalnih i lokalnih ponuđača infrastrukture u oblaku?
5. Koje su usluge potrebne za svako aplikativno rješenje u oblaku?
6. Koje su razine podjele upravljanja infrastrukturom u oblaku?
7. Navedi primjere baza pružatelja usluga u oblaku.
8. Objasni tri načina pristupa podacima.
9. Što su to spot instance?
10. Koja usluga omogućuje statički internet hosting na AWS-u?

2.21. Rješenja bez poslužitelja

Na početku računarstvo aplikacije su postavljane u rad direktno na fizičke poslužitelje. Izumom virtualizacije aplikacije su postavljane u rad na virtualni poslužitelj, izolirane od ostalih aplikacija na poslužitelju. Sljedeći tehnološki iskorak jest kontejnerizacija, pomoću koje su aplikacija i skup procesa i resursa izolirani od ostatka OS-a. Kontejnerizacija je detaljno objašnjena u kasnijem poglavlju. Rješenja bez poslužitelja započela su 2008., kada je Google izdao svoju uslugu Google App Engine. Često se prilikom korištenja izraza serverless zapravo podrazumijevaju funkcije kao usluga. Rješenja bez poslužitelja (engl. *Serveless*) nisu isto što i funkcije kao usluga (engl. *Function as a Service*). Funkcije su samo jedna od usluga koja se može smatrati uslugom bez poslužitelja. Razvojem izvođenja bez poslužitelja pružatelji usluga stvorili su usluge za pohranu podataka bez poslužitelja i integraciju međusobnih komponenti arhitekture aplikacije bez poslužitelja. Usluge bez poslužitelja mogu se kategorizirati u tri kategorije:

- usluge za izvođenje kôda (engl. Compute)
- usluge za integraciju aplikacija (engl. Application integration)
- usluge za pohranu podataka (engl. Storage).

Funkcije su način izvođenja kôda u oblaku pri čemu korisnici usluga u oblaku nisu svjesni poslužitelja. O tome gdje se kôd izvodi brine se pružatelj usluga, korisnici ga moraju samo pohraniti. Programeri i timovi za održavanje infrastrukture ne moraju brinuti o planiranju kapaciteta, konfiguracije, upravljanja infrastrukturom, visokom dostupnošću ili o skaliranju kao ni kod ostalih usluga bez poslužitelja. Kod arhitekture aplikacije bez poslužitelja instanca izvođenja kôda kreira se za svaki pristigli zahtjev. Usluge arhitekture bez poslužitelja naplaćuju se prema vremenu korištenja i alociranim resursima u tom vremenu. Usluge za integraciju aplikacija služe za međusobnu integraciju komponenti jedne aplikacije ili više aplikacija zajedno. U arhitekturi bez poslužitelja komponente aplikacije često komuniciraju putem sustava za razmjenu događaja ili sustava za redove poruka (engl. *message queue*). Rješenja za razmjenu događaja sastoje se od elemenata koji objavljuju i čitaju događaje ili poruke, a koji su raspoređeni u razne teme. Rješenja za redove poruka sastoje se od članova koji zapisuju i čitaju poruke objavljene u redu. Usluge za pohranu podataka omogućuju aplikaciji da sačuva svoje stanje ili pohrani određene resurse. Usluge za pohranu podataka

mogu se podijeliti na usluge za objektnu pohranu podataka te baze podataka. Iako je to možda neintuitivno, baze podataka bez poslužitelja također se mogu naplaćivati prema vremenu korištenja. Usluga AWS Lambda omogućuje pokretanje instance koda kao odgovor na određeni događaj. Događaji koji mogu izazvati pokretanje instance izvođenja koda razni su, neki od primjera takvih događaja jesu dolazni http zahtjev, spremanje novog objekta u S3 ili ručno pokretanje. Svaka instanca izvođenja odvojena je od ostalih izvođenja. Lambda funkcija izvodi se u izvedbenom okruženju. Svaki dodatan zahtjev, odnosno događaja koji uzrokuje pokretanje instance izvođenja funkcije, kreira novu instancu izvođenja. Na taj način lambda funkcija skalira ovisno o potražnji. Funkcija je ograničena brojem istovremenih instanci. Broj dozvoljenih istovremenih instanci promjenjiv je, za njegovu promjenu potrebno je kontaktirati AWS podršku. Svaki poziv funkcije mora završiti unutar 15 minuta – inače je terminirana.

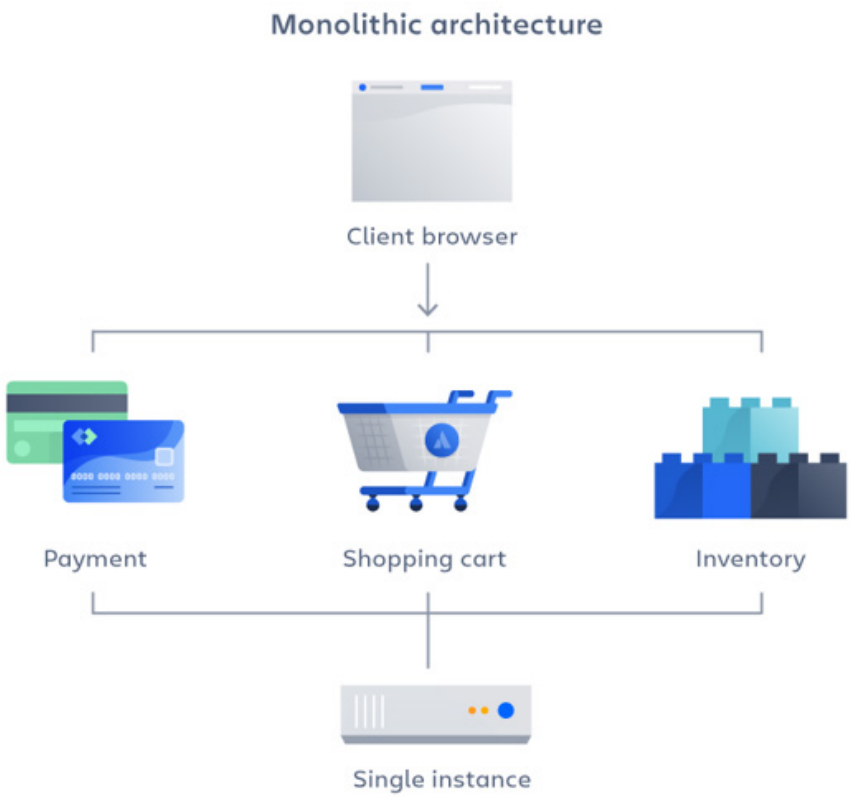
Usluga Azure Functions kao i AWS Lambda omogućuje pokretanje zadanog koda kao odgovor na određene događaje. Function app pruža kontekst u kojemu se Azure funkcija izvodi. Moguće je povezati više funkcija zajedno, gdje rezultat izvršavanja jedne funkcije pokreće drugu funkciju. Azure Functions kao i AWS Lambda integrirane su u ekosustav vlastitih pružatelja usluga u oblaku te je olakšano njihovo integriranje s ostalim uslugama iz portfelja pružatelja usluga. Kako bi funkcije mogle izvršiti određene zadatke i pristupiti podacima u oblaku, mogu imati dodijeljenu rolu ili identitet koji određuju njihova prava.

Google Firebase je platforma za razvoj mobilnih i internetskih aplikacija. Svojim korisnicima nudi razne usluge koje omogućuju brži razvoj aplikacija. Skup usluga koje nudi svojim korisnicima može se nazvati pozadina aplikacije kao usluga. Firebase u portfelju usluga nudi i izvođenje funkcija u usluzi Cloud Functions.

Primjer arhitekture aplikacije bazirane na uslugama bez poslužitelja jest internetska aplikacija u kojoj korisnik od videa kreira GIF. Korisnik putem internetske stranice pohrani video u S3 spremnik. Pohranjivanje videa u S3 spremnik pokreće izvođenje AWS Lambda funkcije, koja od pohranjenog videa kreira GIF. AWS Lambda na kraju svog izvođenja pokreće slanje e-maila s GIF-om u privitku pomoću usluge Amazon Simple Email Services.

2.22. Uvod u koncept mikroservisa

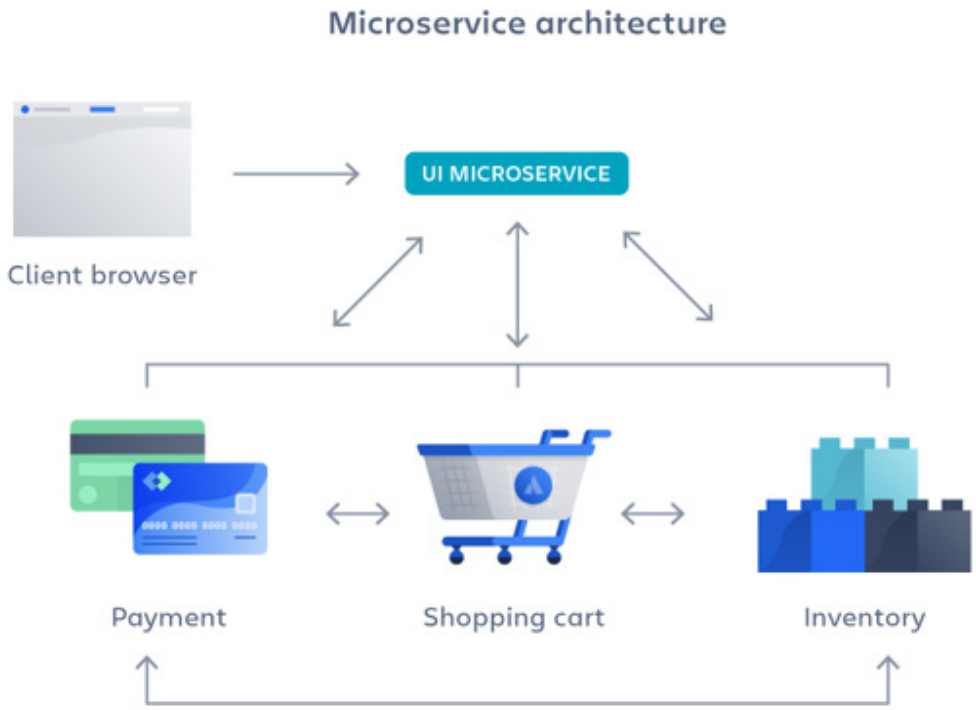
Monolitna arhitektura tradicionalna je arhitektura u kojoj aplikacija sve svoje komponente ima međusobno integrirane kao jedan program. U jednom programu sadržana je sva logika aplikacije. Monolitne aplikacije lakše su za programiranje jer različite komponente komuniciraju kroz memoriju. Traženje grešaka i kodiranje također su lakši jer se radi samo o jednoj grupi koda (engl. *code base*). Postavljanje aplikacija u rad lakše je jer se postavljaju u rad kao jedna komponenta. Rastom broja komponenti aplikacije ona postaje kompliciranija za razumijevanje. Izmjenjivanje koda postaje kompliciranije zbog bliske povezanosti raznih komponenti. Skaliranje je teže jer je nemoguće skalirati samo jednu komponentu aplikacije, već samo cijelu aplikaciju. Primjer monolitne arhitekture vidljiv je na slici ispod. Ako internetska trgovina ima navalu klijenata koji žele pogledati je li dostupan novi proizvod, a ne žele ga kupiti, nije moguće samo skalirati komponentu koda zaduženu za inventar.



Slika 2.22.1. Prikaz monolitne strukture

Preuzeto s https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith - 3.7.2022.

Mikroservisna arhitektura dijeli jednu aplikaciju ili jedan program u više manjih zasebnih komponenti. Svaka od tih komponenti može zasebno biti skalirana, ažurirana, testirana ili postavljena u rad. Aplikacija u mikroservisnoj arhitekturi nije manje kompleksna, već je jasnija podjela odgovornosti za pojedinu funkciju, što olakšava razumijevanje i održavanje koda. Pojedinačno skaliranje komponenti aplikacije višestruko je efikasnije. Mikroservisna arhitektura omogućuje slabiju vezu (engl. *decoupling*) između komponenti u smislu manjeg oslanjanja na točan način kako je nešto implementirano. Primjer mikroservisne arhitekture vidljiv je na slici ispod. U ovom slučaju komponenta za inventar može biti slobodno skalirana neovisno o ostalim elementima.



Slika 2.22.2. Prikaz mikroservisne arhitekture

Odluku hoće li aplikaciji biti monolitna ili mikroservisna arhitektura najbolje je donijeti na početku njezina dizajna. Tada ne postoji trošak prepravljanja koda i mijenjanja arhitekture usred razvoja.

Kako bi omogućili korisnicima da isprobaju njihove usluge i privuku ih na korištenje njihove platforme za usluge u oblaku, ponuđači usluga nude planove besplatnog korištenja (engl. *free tier*). Svaki od ponuđača usluga nudi različitu količinu resurse na

besplatno korištenje. Ponudu besplatnih usluga moguće je istražiti na poveznicama ispod:

- **AWS** - https://aws.amazon.com/free
- **Azure** - https://azure.microsoft.com/en-us/pricing/free-services
- **GCP** - https://cloud.google.com/free.

2.23. Hladni start pojedinih usluga

Funkcije se zapravo izvode u vlastitom kontejneru. Ukoliko se usluga ne pokreće često, pružatelj usluga u oblaku neće imati spreman kontejner koji će sadržavati kôd funkcije za pokretanje. Prilikom hladnog pokretanja kôd same funkcije mora biti dohvaćen iz pohrane, kontejner pokrenut, kôd učitan u memoriju i na kraju funkcija pokrenuta. To može izazvati posljedicu da prvo pokretanje funkcije ili pokretanje funkcije nakon dugog vremena potraje nekoliko sekundi, što ovisno o namjeni funkcije može biti predugo. AWS kao rješenje problema hladnog starta nudi uslugu „Provisioned Concurrency", koja omogućuje korisniku da uvijek ima određen broj instanci funkcije spreman za izvršavanje koda. Rizik hladnog starta u tome je što će se funkcija nakon predugog vremena neizvođenja sporo pokrenuti, što će dovesti do većih latencija za korisnika. Rješenja bez poslužitelja i mikroservisi omogućuju novi pristup arhitekturi aplikacija. Prije implementacija takve arhitekture potrebno je razmisliti koje su prednosti, a koji nedostaci korištenja te arhitekture za pojedinačnu aplikaciju i ima li razvojni tim dovoljno znanja za kreiranje takve aplikacije. Ispravnim korištenjem rješenja bez poslužitelja mogu biti višestruko jeftinija i boljih performansi u odnosu na tradicionalna, ali i skuplja i lošijih performansi ako nisu korištena na ispravan način.

PITANJA ZA PONAVLJANJE:

1. Što je funkcija?
2. Navedi tri kategorije usluga bez poslužitelja.
3. Koja je razlika između FaaS modela i računarstva bez poslužitelja?
4. Što je Google Firebase?
5. Koja je razlika između monolitne aplikacije i mikroservisa?
6. Skiciraj monolitnu aplikaciju.
7. Skiciraj mikroservis.
8. Što je „Provisioned Conccurency”?
9. Objasni hladni start pojedinih usluga.

2.24. Pristup infrastrukturi

Današnja infrastruktura uglavnom se ne nalazi kraj administratora, već je smještena u podatkovnim centrima po cijelom svijetu. Administracija poslužitelja i ostalih elemenata infrastrukture nezamisliva je i neprovediva bez protokola za udaljeni pristup infrastrukturi. Tijekom upravljanja konfiguracijom poslužitelja administratori često moraju kopirati različite datoteke i mape. Kako ne bi morali nositi USB do fizičkog poslužitelja, kreirani su protokoli za prijenos podataka.

Protokoli pristupa infrastrukturi i protokoli za prijenos podataka moraju podržavati enkripciju kako njihova komunikacija ne bi bila ranjiva na mrežno presretanje. Ukoliko određeni napadač presretne komunikaciju koja nije kriptirana, može iz nje saznati lozinke kojima se administrator prijavio na udaljeni poslužitelj, kopirane datoteke, pokrenute naredbe i njihov rezultat.

Enkripcija može biti simetrična i asimetrična. Simetrična enkripcija vrsta je enkripcije u kojoj se pomoću jednog ključa kriptira sva komunikacija. Svi sudionici u enkripciji moraju na siguran način saznati taj ključ. Asimetrična enkripcija vrsta je enkripcije u kojoj svaki sudionik komunikacije ima svoj privatni i javni ključ. Privatni i javni ključ različiti su, ali matematički povezani. Sudionici mogu svoj javni ključ podijeliti s bilo kim, ali svoj privatni ključ moraju zaštititi jer svatko tko ima privatni ključ može oponašati sudionika čiji ključ ima. Prilikom slanje poruke drugom sudioniku poruka se prvo kriptira njegovim javnim ključem te je zbog toga nitko osim željenog primatelja ne može dekriptirati ukoliko nema njegov privatni ključ.

FTP (engl. *File Transfer Protocol*) protokol je za prijenos datoteka između poslužitelja i klijenta. FTP koristi TCP port 21 za upravljanje i TCP port 20 za prijenos datoteka. FTP je poprilično star protokol i nije prikladan za korištenje u današnje doba zbog svojih ranjivosti i nepodržavanja enkripcije.

SFTP (engl. *SSH File Transfer Protocol*) može se shvatiti kao nasljednik FTP protokola, ali zapravo se radi o proširenju SSH protokola kako bi se pomoću njega mogle sigurno prenositi datoteke. SFTP protokol ostvaruje sigurnu, odnosno kriptiranu vezu kroz SSH protokol. Koristi TCP port 22, kao i SSH protokol.

SCP (engl. *Secure Copy Protocol*) protokol je za prijenos podataka. Koristi TCP port 22. Razvojni inženjeri preporučuju da se SCP protokol prestane koristiti i zamijeni alternativnim protokolom kao što je SCP ili rsync protokol.2

SSH (engl. *Secure Shell Protocol*) protokol omogućuje sigurnu komunikaciju između dvaju računala preko nesigurne mreže. Podržava autentifikaciju putem lozinke i putem javnih i privatnih ključeva. Najveća opasnost kod SSH protokola nesigurno je postupanje s privatnim ključevima. U sljedećem poglavlju nalazit će se upute za sigurno dijeljenje ključeva.

Prvi korak svakog korisnika koji želi koristiti autentifikaciju putem ključeva jest generirati svoj par ključeva. Postoje mnogi alati za generiranje javnih i privatnih ključeva, a jedan od najpopularnijih alata jest ssh-keygen. Pomoću njega korisnici mogu lako kreirati svoj par ključeva. Prilikom generiranje para ključeva privatni ključ može se dodatno zaštititi pomoću lozinke. Ukoliko se izgubi javni ključ, može se ponovno generirati iz privatnog ključa zbog njihove matematičke povezanosti. Nakon što je par ključeva generiran, potrebno je kopirati javni ključ u authorized_keys datoteku koja se nalazi u mapi .ssh. .ssh mapa nalazi se u home direktoriju svakog korisnika. Datoteke i mape čije ime započinje točkom označavaju skrivene datoteke i mape koje nisu vidljive bez korištenja odgovarajućih zastavica. Pomoću privatnog ključa korisnik će se moći spojiti na poslužitelj kao korisnik koji u svojoj authorized_keys datoteci ima njegov javni ključ. Ukoliko .ssh mapa ne postoji, potrebno ju je ručno kreirati. **`ssh-copy-id`** naredba može se koristiti za kopiranje javnih ključeva na Linux i Unix poslužitelje. Administratori i korisnici mogu svoj javni ključ kopirati na različite poslužitelje i postupiti mu pomoću jednog privatnog ključa.

Puna putanja konfiguracijske datoteka za ssh servis kod Linux i Unix OS-ova najčešće je **`/etc/ssh/sshd_config`**. Konfiguracijska datoteka sadrži mnoge mogućnosti čije objašnjenje može biti pronađeno u manual stranicama koje sadrže upute. Neke od najvažnijih opcija jesu **`PasswordAuthentication`**, **`PubkeyAuthentication`** i **`PermitRootLogin`**. Ukoliko je vrijednost **`PasswordAuthentication`** opcije **`Yes`**, autentifikacija putem lozinki dozvoljena je, inače nije. Zadana vrijednost **`PasswordAuthentication`** opcije je **`Yes`**. Ukoliko je vrijednost **`PubkeyAuthentication`** opcije **`Yes`**, autentifikacija putem ključeva dozvoljena je, inače nije. Zadana vrijednost **`Pubkey-`**

`Authentication` opcije je **`Yes`**. Opcija **`PermitRootLogin`** kontrolira ima li root korisnik mogućnost spajanje putem ssh protokola. Zadana vrijednost **`PermitRootLogin`** opcije je **`prohibit-password`**, što ga ograničava na autentifikaciju putem ključeva. Sigurnosne preporuke su onemogućiti spajanje root korisniku nakon što je podešen korisnik sa sudo pravima te onemogućiti spajanje putem lozinki nakon što je konfigurirano spajanje putem ključeva. Zadane vrijednosti mogu ovisiti o OS-u. SSH protokol često se koristi za upravljanje Linux operativnim sustavima i mrežnim uređajima. S Windows 10 i Windows Server 2019 uvedena je podrška za spajanje putem SSH i na Windows OS.3

WinRM (engl. *Windows Remote Management*) Microsoftova je implementacija WS-Management protokola. WS-Management protokol služi za udaljeno upravljanje uređajima koji ga podržavaju. PowerShell koristi WinRM protokol kako bi pokretao naredbe na udaljenim računalima. WinRM koristi TCP port 5985 za veze putem HTTP-a i TCP port 5986 za veze putem HTTPS-a. Veza putem HTTPS-a sigurnija je, ali zahtijeva više konfiguracije.

Protokol	Port
FTP	21/tcp - upravljanje 20/tcp - podatkovni prijenos
SFTP	22/tcp
SCP	22/tcp
SSH	22/tcp
WinRM	5985/tcp - HTTP prijenos 5986 tcp - HTTPS prijenos
RDP	3389/tcp 3389/udp

Slika 2.24.1. Protokoli pristupa i zadani portovi

RDP (engl. *Remote Desktop Protocol*) protokol je razvijen od strane Microsofta koji omogućuje povezivanje s drugim računalom preko mreže putem grafičkog sučelja. Najčešće se koristi za upravljanje Windows OS-om. RDP protokol koristi TCP port 3389 i UDP port 3389.

FTP, SFTP i SCP protokoli služe za prijenos podataka, dok SSH, WinRM i RDP pro-

tokoli služe za udaljeni pristup infrastrukturi.

Za sam pristup infrastrukturi najčešće se koriste lozinke, ali one su i najnesigurniji način autentifikacije. Ukoliko adekvatne sigurnosne mjere nisu konfigurirane, napadači naprosto mogu isprobati sve moguće kombinacije dok ne pogode točnu lozinku. Ovisno o dužini lozinka može biti iznimno brzo pogođena. Takav tip napada naziva se „brute force" napadom. Pogađanje može biti i brže kreiranjem rječnika koji se kreira na temelju informacija poznatih o tvrtki ili osobi čija se lozinka pogađa. Preporuča se korištenje lozinki od 16 ili više znakova i ne ponavljanje lozinki za različite račune. Za lakše pamćenje lozinki one mogu biti formulirane kao fraze na primjer *SvakiDanKoristimL1nux!.*

Autentifikacija putem para ključeva višestruko je sigurnija od autentifikacije putem lozinki. Samo za usporedbu, lozinka od 20 znakova ima 160 bitova, dok je uobičajena veličina SSH ključa 2048 ili 4096 bitova ako se koristi RSA algoritam. Uz RSA algoritam često se koristi i ECDSA algoritam, koji ima kraće ključeve uz jednaku razinu sigurnosti. RSA i ECDSA primjeri su algoritama za asimetričnu enkripciju.

Autentifikacija je proces provjere identiteta, odnosno je li osoba koja se prijavljuje stvarno osoba kojom koja se predstavlja. Autentifikacija putem više faktora omogućuje veću razinu sigurnosti kroz dodatnu provjeru radi li se o korisniku koji ima mogućnost prijave na sustav. Postoje tri glavne mogućnosti autentifikacije, nešto što osoba zna (npr. lozinka), nešto što osoba posjeduje (npr. token uređaj) i nešto što osoba jest (npr. otisak prsta). Prilikom dvofaktorske autentifikacije osoba koja se autentificira mora proći dvije provjere kako bi se utvrdio njen identitet.

Pristup bez lozinki ili autentifikacija bez lozinke vrsta je autentifikacije u kojoj je primarni način autentifikacije nešto što korisnik posjeduje. Na primjer, korisnik može posjedovati određeni broj mobitela, određeni e-mail račun ili token uređaj. Korisnik prilikom autentifikacije mora unijeti SMS kod, e-mail adresu, potvrditi zahtjev u aplikaciji ili trenutni PIN s tokena, a koji su pristigli u vrijeme autentifikacije.

Sigurno dijeljenje lozinki i ključeva iznimno je bitno za svaku organizaciju kako ne bi došlo do neovlaštenog pristupa. Lozinke, ključevi i ostale tajne spremaju se u sefove

(engl. *vault*) te im je pristup ograničen na temelju kontrola pristupa. Pružatelji usluga u oblaku nude usluge sefa, ali postoje i samostalne aplikacije koje nude istu funkcionalnost. Primjeri usluge sefa u oblaku jesu AWS Secrets Manager, Azure Keyvault i GCP Secret Manager. Primjeri samostalnih aplikacija koje nude istu funkcionalnost jesu HashiCorp vault i Bitwarden. Korisnici sefova mogu biti ljudi i/ili aplikacije. Svi sefovi ne podržavaju obje vrste korisnika, kao ni sve vrste tajni. Sefovi kao usluga ponuditelja usluga u oblaku najčešće podržavaju ljude i aplikacije kao korisnike. Lozinke i ostale tajne ne bi se trebale zapisivati u čistom tekstu na disk, već bi trebale biti sigurno pohranjene u sefu i dohvaćene u trenutku kada su potrebne. Uobičajena je praksa da se u samom kodu aplikacije ne pohranjuju lozinke, već da se lozinke ubace kao varijable okoliša. Infrastruktura svake organizacije sadrži informacije bitne za njeno poslovanje te zbog toga mora biti osigurana na odgovarajući način. Sprečavanje napadača da dođu do tajni samo je jedan od načina osiguranja. Naravno, mjere za sprečavanje napadača moraju biti implementirane na način da validni korisnici mogu neometano pristupiti infrastrukturi.

PITANJA ZA PONAVLJANJE:

1. Navedi protokole pristupa i njihove zadane portove.
2. Navedi alate za sigurno dijeljenje lozinki i SSH ključeva.
3. Koja je razlika između simetrične i asimetrične enkripcije?
4. Kojem se operacijskom sustavu često pristupa putem RDP i WinRM protokola?
5. Kojem se operacijskom sustavu često pristupa putem SSH protokola?
6. Koje načine autentifikacije podržava SSH protokol?
7. Koja je razlika između privatnog i javnog ključa?
8. Koji protokol odabrati za kopiranje podataka?
9. Što je dvofaktorska autentifikacija?
10. Koji je protokol sigurniji, SFTP ili FTP?

3. Integracija kôda

U OVOM POGLAVLJU NAUČIT ĆETE:

- razlikovati Git protokol od Git repozitorija
- kreirati Git repozitorije
- zašto se radi verzioniranje koda
- strukturu Git repozitorija
- održavati Git repozitorij.

3.1. Git

Git (engl. *Global Information Tracker*) sustav je otvorenog kôda koji služi za kontrolu verzija računalnih datoteka. Sustav je to kojem je fokus bilježenje i praćenje promjena kroz vrijeme nad datotekama, što omogućuje da više ljudi nesmetano radi istovremeno nad istim datotekama. Spomenute funkcionalnosti omogućuju nam da vidimo što je točno tko radio i u slučaju određene pogreške koju sam sustav ne zna razriješiti možemo stanje vratiti na prijašnju ili neku određenu željenu verziju. Git kao pojam pojavljuje se 2005. godine u trenutku kada je Linus Torvalds razvijao Linux Kernel5. Projekt je nastao s ciljem olakšavanja vođenja velikih i kompleksnih projekata. U samim počecima Git je imao potpunu drugačiju zamisao. Trebao je biti osnova drugim sustavima za razvijanje kôda. Međutim, rastom informatičke zajednice i rastom tehnologija Git je dobio potpuno novu namjenu. Postao je sustav za verzioniranje. Takav

sustav u programerskoj je zajednici bio najbrže prihvaćen. Zajednica je korištenjem Gita dobila izrazito povećanje efikasnosti suradnja timova. Svaki od korisnika sustava za verzioniranje može u svakom trenutku vidjeti rad svojih kolega te njihove povijesne promjene, što omogućuje visoku stabilnost u razvoju aplikacija. Git je trenutno daleko najpoznatiji i najpopularniji sustav za verzioniranje kôda te je postao standardom, iako je generalno kompliciraniji od konkurenata.

Za razumijevanje kroz praktičan primjer moramo biti svjesni postojanja strategija za verzioniranje programskog koda. Osim što Git verzionira izmjene, potrebno je popratiti izmjene određenim oznakama za prepoznavanje svakog artefakta izrade. Najpoznatija strategija naziva se semantičkim verzioniranjem.
Primjer takvog verzioniranja izgledao bio ovako: **MAJOR.MINOR.PATCH.**

MAJOR, odnosno glavna verzija, mora se povećati ako postoje bilo kakve unazad nekompatibilne neispravne promjene uključene u izdanje ili ako stvaramo izrazito veliku novu funkcionalnost koja je nekompatibilna s prethodnim verzijama. To ima prednost jer svima olakšava brzu identifikaciju hoće li nova verzija raditi drugačije od prethodne.

MINOR, odnosno druga verzija, mora se povećati ako postoji funkcionalnost kompatibilna s prethodnim verzijama, a napravili smo izmjenu nad njom ili dodali novu funkcionalnost. U najstrožem smislu to znači da bismo trebali moći nadograditi na novu manju verziju bez ikakvih prijelomnih promjena.

PATCH je verzija koja je namijenjena ispravcima grešaka, kompatibilna s prethodnim verzijama. Ne biste trebali očekivati nove funkcije s novom verzijom zakrpe, samo poboljšanja.

Konkretnu primjenu Gita u razvoju aplikacija možemo opisati sljedećim praktičnim primjerom. U početku razvoja aplikacije trebamo biti svjesni da svaka aplikacija ima stvarnog korisnika za kojeg rješavamo određeni problem. Problemi su definirani pomoću korisničkih zahtjeva, odnosno korisnik će opisati problem koji moramo riješiti. Vrlo često u trenutku definiranja korisničkih zahtjeva sam korisnik nije svjestan svih problema koje moramo rješavati. Kada korisnik pogleda prvu isporučenu verziju aplikacije, čak i ako je pisana točno prema njegovim zahtjevima, on će pronaći nešto što bi

trebalo promijeniti. U tom trenutku kreće stvaranje nove verzije aplikacije. U trenutku prezentacije korisniku se pokazuje verzija 1.0, a nakon što se sve tražene izmjene implementiraju, označili bismo novu verziju aplikacije kao verziju 1.1.

Nekoliko dana kasnije, s malo više iskustva u radu s aplikacijom, klijent zaključuje kako sada ima bolju ideju i ne sviđaju mu se izmjene u verziji 1.1. Na tradicionalan način ručno bismo obrisali sve nove značajke i obrisali bismo ih zauvijek. Pomoću Gita, vratili bismo se na verziju 1.0, napravili izmjene i to pohranili u 1.0.X, i ako se ponovno pojavi želja za određenim značajkama iz verzije 1.1, jednostavno bismo se preusmjerili na spomenutu verziju, uzeli potrebne značajke i dodali ih u verziju 1.0.X. Oznaka X označava da bismo mijenjali onoliki broj koliko smo zakrpa napravili.

3.2. Podešavanje Gita

Podešavanje Gita jednako je na svim operativnim sustavima. Bitno je da imate instaliranu Git programsku podršku. Ako koristite Linux operativni sustav, velika je vjerojatnost da će Git programska podrška biti predinstalirana, ali u slučaju operativnog sustava Windows sami ćete morati instalirati programsku podršku. Za rad i podešavanje Gita preporuka je koristiti naredbeni redak. Neovisno o operativnom sustavu i okruženju u kojem radite koraci za podešavanje Gita uvijek su jednaki. Osnovnu konfiguraciju trebali biste napraviti samo jednom na bilo kojem računalu. Bitno je imati na umu da se u slučaju ažuriranja Gita osnovna konfiguracija neće promijeniti. Osnovna konfiguracija pokreće se pomoću naredbe **`git config`**. Bitno je razumjeti da upisane postavke konfiguracije mogu vrijediti za lokalno okruženje, odnosno samo za specifičan projekt ili globalno za sve projekte.

Globalne postavke postavljaju se pomoću naredbe:

```
git config --global <naziv> <vrijednost>
```

Nakon pokretanja naredbe potpuna konfiguracija spremit će se u skrivenu datoteku **`.gitconfig`** u vašoj korisničkoj mapi na Windows operativnom sustavu **C:\Users\<vaš_korisnik>** ili u **/home/<korisničko_ime>** na Linux operativnom sustavu.

Lokalne postavke spremaju se u sakrivenu mapu .git unutar vašeg projekta, a tad je potrebno navesti naredbu:

```
git config <naziv> <vrijednost>
```

Za rad na projektima potrebno je postaviti ime i adresu e-pošte kako bi drugi korisnici znali tko je radio izmjene.

```
git config –-global user.name „Pero Perić“
git config –-global user.email „pero.peric@algebra.hr“
```

Ako želite provjeriti postavljene postavke, potrebno je pokrenuti naredbu

```
git config –-list
```

3.3. Lokalni repozitorij

Nakon uspješnog postavljanja Git postavki spremni smo krenuti na Git repozitorije. Prednost Gita u odnosu na druge sustave za verzioniranje kôda jest to što apsolutno svaka mapa može postati Git repozitorij. Kada pričamo o Git repozitoriju, pričamo o mapi u kojoj smo aktivirali Git funkcionalnosti. Inicijalizacijom Git repozitorija u mapi dobivamo mogućnost praćenja i spremanja povijesti svih promijenjenih datoteka unutar mape. Kada inicijaliziramo Git unutar mape, on će kreirati skrivenu mapu pod nazivom **`.git`**. U slučaju potrebe sigurnosne kopije cijelog repozitorija, sve što je potrebno jest spremiti **`.git`** mapu u arhivu. Prvo što želimo jest doći u mapu u kojoj želimo aktivirati .git te desnim klikom ući u naredbeni redak. Zatim pokrećemo:

```
Git init
Initialized empty Git repository in <lokacija_mape>
```

Nakon uspješno inicijalizirane Git mape zaključit ćemo da smo sada napravili lokalni Git repozitorij. Nazivamo ga lokalnim zato što se nalazi na našem računalu.

3.4. Udaljeni repozitorij

Bitno je naglasiti da je Git distribuirani sustav za verzioniranje kôda i u ovoj cjelini dotaknut ćemo se na koji način naš lokalni repozitorij može postati distribuiran. Cilj je zapravo postići komunikaciju između lokalnog i udaljenog repozitorija zato što udaljenom repozitoriju mogu pristupiti ostali koji rade na razvoju neke aplikacije. Postoje različiti načini za postizanje spomenute interakcije. Moguće je da smo tek stvorili mapu na lokalnom računalu pomoću **`git init`** i onda možemo poslati sadržaj mape na udaljenu lokaciju i na taj način ponudili smo naš Git repozitorij drugim korisnicima. Moguće je i da smo pronašli postojeći repozitorij negdje na internetu i sada mi želimo preuzeti taj repozitorij. Spajanje na udaljeni repozitorij objasnit ćemo u sekciji upravljanje udaljenim repozitorijima.

Za kreiranje ili preuzimanje postojećeg repozitorija potreban nam je udaljeni poslužitelj. Postoje razne hosting usluge, odnosno platforme, koje nam olakšavaju da ne moramo sami brinuti o repozitoriju. Jedan od najpoznatijih takvih jest GitHub. U primjerima koje ćete vidjeti u priručniku koristit ćemo GitHub. Tehnički termin za preuzimanje repozitorija je kloniranje. Kako bismo klonirali Git repozitorij, moramo biti svjesni da svaki udaljeni Git repozitorij mora sadržavati adresu.

Primjeri adrese repozitorija:

```
git://github.com/algebra/devops.git
HTTPS://algebra@github.com/algebra/devops.git
```

Platforma GitHub također nam omogućuje da istom repozitoriju pristupimo putem internetskog preglednika: HTTPS://github.com/algebra/devops. Svaki udaljeni repozitorij ima i svoje kratko ime, odnosno alias. Zadani alias za udaljeni repozitorij je **`origin`**.

3.5. Protokoli koje koristi Git

U prethodnom poglavlju, kada smo spominjali primjere adresa repozitorija, primijetili smo da postoje određeni protokoli pomoću kojih možemo komunicirati s udaljenim Git repozitorijem. Kada postavljamo komunikaciju s udaljenim Git repozitorijem, moramo najprije odabrati protokol pomoću kojeg ćemo komunicirati. Git može koristiti četiri protokola za prijenos podataka: lokalni, HTTP, Secure Shell (SSH) i Git. Kada pričamo o lokalnom protokolu, mislimo na osnovni protokol. Lokalni protokol koristi se kada imamo dodatno računalo koje se nalazi u istoj mreži u kojoj se nalazi i računalo s Git repozitorijem. U spomenutom slučaju potrebno je imati NFS (engl. Network File System) sustav na kojem se nalazi Git repozitorij. Ovakav primjer izrazito je rijedak u praksi. Izrazito čest primjer komunikacije jest onaj putem HTTP protokola. Komunikacija putem HTTP-a naziva se još i Smart HTTP komunikacija jer komunikacija zapravo ide putem HTTPS-a, odnosno komunikacija s udaljenim Git poslužiteljem kriptirana je. HTTP mehanizam komunikacije izrazito je koristan jer možemo koristiti dodatne metode provjere autentičnosti. Prilikom komunikacije s Git repozitorijem trebali bismo ponuditi korisničko ime i lozinku, koji će nam omogućiti pristup. Komunikacija putem HTTP-a izrazito je česta zato što je to najjednostavniji, a istovremeno i siguran model komunikacije s repozitorijem. Također se koristi i komunikacija putem SSH protokola jer na udaljenom serveru s kojim komuniciramo vrlo često imamo već podešen SSH ključ. SSH protokol siguran je, a prednost mu je i činjenica da je sveprisutan mrežni protokol koji je jednostavno podesiti. Na posljetku imamo Git protokol. Git protokol dolazi u paketu s instalacijom Gita. Protokol predefiniran na TCP portu 9418 koji pruža sličnu uslugu kao i SSH protokol, ali bez provjere autentičnosti.

3.6. Kloniranje repozitorija

Kloniranje je proces kojim kopiramo cijeli repozitorij s udaljene lokacije na naše lokalno računalo. S kloniranim repozitorijem možemo nastaviti raditi kao i s onim lokalnim. Potrebno je imati na umu da klonirani repozitorij čuva informacije o repozitoriju iz kojeg je kloniran, a te informacije koristit će se kako bismo kasnije ponovno mogli poslati sve izmjene ili preuzeti nove izmjene s udaljenog repozitorija.

Kako bismo napravili postupak kloniranja, potrebno je izvršiti Git naredbu s oznakom adrese gdje se nalazi udaljeni repozitorij:

```
git clone git://<udaljeni_repozitorij>
Cloning into uvod-u-git...
remote: Counting objects: 643, done.
remote: Compressing objects: 100% (346/346), done.
remote: Total 643 (delta 384), reused 530 (delta 271)
Receiving objects: 100% (643/643), 337.00 KiB | 56 KiB/s, done.
Resolving deltas: 100% (384/384), done.
```

Nakon izvršene naredbe kopirao se projekt, zajedno s cijelom povijesti izmjena i nadopuna, na naše računalo. Sada u tom projektu možemo gledati povijest izmjena, raditi izmjene i slično.

Od trenutka kada smo klonirali repozitorij trebamo biti svjesni njegove strukture. Prvo trebamo biti svjesni da postoje dva repozitorija u ovom trenutku. Jedan je udaljeni koji smo klonirali, a drugi je lokalni, koji se nalazi na našem računalu. Svaki repozitorij građen je od grana, odnosno onih različitih verzija s početka teme Git. Kako bismo mogli vidjeti strukturu našeg repozitorija, Git sadrži naredbu **`git branch -a`**. Pomoću spomenute naredbe dobivamo ispis svih grana koje su nam trenutno dostupne u lokalnom i udaljenom repozitoriju. Moramo se voditi informacijom da su nam nakon kloniranja repozitorija postale dostupne i grane s udaljenog repozitorija.

```
git branch -a
* main
remotes/origin/main
```

Iz ispisa linije broj dva možemo iščitati da se naš udaljeni repozitorij naziva **`origin`**. **`Remote/`** oznaka označava da je to udaljeni repozitorij, **`origin/`** označava naziv repozitorija, a **`/main`** označava granu na udaljenom repozitoriju. **`Remotes/origin/main`** u ovom je slučaju kopija grane main u udaljenom repozitoriju **`origin`**, što bi značilo da svaka izmjena neće utjecati na stvaran sadržaj na udaljenom repozitoriju. Zvjezdica ispred grane **`main`** označava našu trenutku lokaciju, a to je zapravo naš

lokalni repozitorij. **Remotes/origin/main** koristit će se za osvježavanje lokalnog repozitorija.

3.7. Upravljanje udaljenim repozitorijem

Nakon što smo kreirali lokalni i udaljeni repozitorij, vrijeme je za njihovo povezivanje. Administraciju udaljenog repozitorija radimo pomoću naredbe **git remote**. Ako nismo napravili kloniranje repozitorija naredbom **git clone**, a želimo povezati lokalni repozitorij s udaljenim, potrebno je izvršiti naredbu:

```
git remote add <naziv> <adresa>
```

Ako želimo obrisati udaljeni repozitorij iz određenog razloga, to radimo pomoću naredbe:

```
git remote rm <naziv>
```

3.8. Naredba: git fetch

Naredbu **git fetch** koristimo ako je netko napravio izmjene u repozitoriju, a mi želimo dohvatiti te izmjene. Potrebno je imati na umu da je naš lokalni repozitorij main radna verzija unutar koje mi sami stvaramo izmjene. U trenutku dodavanja udaljenog repozitorija stvaramo lokalnu kopiju udaljenog main repozitorija, međutim udaljeni repozitorij ne ažurira se automatski. Ako netko napravi izmjene na udaljenom repozitoriju, naš lokalni repozitorij neće biti svjestan izmjena. Kao što smo inicirali kloniranje, tako moramo inicirati i ažuriranje. Ažuriranja se rade na razini grana i svode se na dvije razine. U prvom procesu želimo dohvatiti stanje izmjena. Pomoću stanja izmjena možemo vidjeti koje su se datoteke izmijenile.

```
git fetch <naziv> <naziv_grane>
remote: Counting objects: 5678, done.
remote: Compressing objects: 100% (1967/1967), done.
remote: Total 5434 (delta 3883), reused 4967 (delta 3465)
```

```
Receiving objects: 100% (5434/5434), 1.86 MiB | 561 KiB/s, done.
Resolving deltas: 100% (3883/3883)
From git@github.com:Darwin0id/uvod-u-git
```

U ovom trenutku **origin/main** u našem lokalnom repozitoriju osvježen je tako da mu je stanje isto kao i **main** udaljenog repozitorija. S **origin/main** možemo raditi skoro sve kao i s ostalim lokalnim granama. Možemo, na primjer, pogledati njegovu povijest:

```
git log origin/main
```

Možemo pogledati razlike između udaljene i trenutne grane

```
git diff origin/main
```

3.9. Naredba: git pull

U sljedećem koraku izvršit ćemo naredbu koja će uzeti izmjene i staviti ih u naš repozitorij.

```
git fetch
git merge origin/main
```

Opisane dvije naredbe skoro uvijek izvršavamo u istom redoslijedu te je zbog učestalosti ponavljanja Git ponudio ekvivalent toj kombinaciji:

```
git pull
```

Opisana naredba kombinacija je **git fetch** i **git merge origin/main**.

3.10. Naredba: git push

Svaka naša dosadašnja radnja odnosila se na rad u lokalnom repozitoriju. Nakon postavljanja projekta i dohvaćanja svih izmjena krenimo s radom na udaljenom repozitoriju. Odlučili smo izmijeniti određenu datoteku i sada je želimo poslati na udaljeni

repozitorij. Kako bismo provjerili postoji li nešto za slanje u udaljeni repozitorij, prvo je potrebno provjeriti to naredbom: **`git status`**. Ako dobijemo poruku koja ukazuje na to da ne postoji nijedna datoteka s izmjenom, a znamo da smo dodali datoteku:

```
git status
# On branch main
nothing to commit (working directory clean)
```

U tom slučaju moramo biti svjesni da se naša datoteka ne nalazi u indeksu gita. Git sadrži vlastiti indeks unutar kojeg bilježi stanja datoteka. Ako smo naknadno kreirali datoteku, Git neće biti svjestan toga, pa zato moramo napraviti sljedeću radnju:

```
git add naziv_datoteke.ekstenzija
```

Sada, kada bismo ponovno pokrenuli **`git status`**, vidjeli bismo na popisu našu novu datoteku. Sada nam preostaje spremiti izmjenu.

```
git commit –m „Poruka koju želimo upisati“
```

U znakovnom nizu nakon zastavice -m moramo unijeti komentar koji se odnosi na izmjenu za određenu datoteku. Git nam neće dopustiti spremanje bilo kakvih izmjena bez komentara. Kada napravimo spremanje izmjena i odlučimo provjeriti status korištenjem naredbe **`git status`**, nećemo više vidjeti popis izmjena. Sada smo spremni napraviti pohranu izmjena na udaljeni repozitorij. Ono što moramo znati jest da prebacivanje naših lokalnih izmjena na udaljeni repozitorij ovisi o tome imamo li potrebne ovlasti za stvaranje izmjena u udaljenom repozitoriju. Ako nemamo ovlasti, dobit ćemo poruku pogreške koja će tražiti da od administratora udaljenog repozitorija zamolimo za ovlasti pisanja.

Ako imamo ovlasti, onda je potrebno pokrenuti naredbu:

```
git push <naziv> <naziv_grane>
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 1.45 KiB, done.
```

```
Total 9 (delta 6), reused 0 (delta 0)
To git@github.com:Darwin0id/uvod-u-git.git
0335d78..63ced90 main -> main
```

Ovo je uspješan primjer slanja kôda na udaljeni repozitorij. No što ako je netko drugi napravio izmjene prije nas? Nažalost, nećemo moći poslati izmjene sve dok ih mi ne preuzmemo. U tom slučaju, ako napravimo **`git push <naziv> <naziv_grane>`**, dobit ćemo poruku sličnu ovome:

```
git push <naziv> <naziv_grane>
To git@github.com:Darwin0id/uvod-u-git.git!
[rejected] main -> main(non-fast-forward)
error: failed to push some refs to git@github.com:Darwin0id/uvod-u-git.git
```

Git u ovom slučaju ne zna što napraviti s obzirom na to da stanje našeg lokalnog repozitorija nije jednako onom udaljenom. U tom slučaju koristit ćemo se naredbom koju znamo iz prethodne cjeline.

```
git pull <naziv> <naziv_grane>
```

Kad preuzmemo posljednje izmjene, tek nakon toga možemo izvršiti naredbu **`push`**.

3.11. Datoteka .gitignore

Tijekom rada unutar mape pojavit će se datoteke koje ne želimo spremiti u povijest projekta. U praksi to su najčešće konfiguracijske datoteke za različite uređivače programskog kôda ili datoteke koje nastaju u procesu prevođenja (npr. .class za javu, .pyc za python, .o za C i slično). U tom slučaju dužni smo definirati Gitu što smije pohranjivati, a što ne. Da bismo ograničili pohranjivanje datoteka, moramo u glavnoj mapi (ne u nekoj podmapi) kreirati datoteku s nazivom .gitignore. Unutar datoteke moramo navesti sve datoteke odvojene novim retkom koje ne želimo pratiti. Uz navođenje datoteka možemo navesti i mape, no bitno je da na kraju naziva mape dodamo kosu crtu u desno.

```
# Java prevedene klase:

*.class
# Mape s rezultatima prevođenja i izgradnje
target/
```

Linije koje započinju oznakom **#** komentari su i Git će se ponašati kao da ne postoje.

3.12. Grane u Git strukturi

Jedna od velikih prednosti Gita jest to što nam omogućuje jednostavan i brz rad s višestrukim granama. Grane predstavljaju odvojene i samostalne dijelove razvoja projekta. Cilj je da odvojimo glavnu granu u zasebnu, unutar koje ćemo testirati ili raditi posebne izmjene, a da ne utječemo na glavni smisao projekta. Pomoću grana možemo raditi na različitim dijelovima projekta bez da utječemo na glavnu granu, a na kraju možemo našu odvojenu granu spojiti s glavnom ako smo zadovoljni rezultatom rada. U prethodnim koracima opisali smo naredbu **`git branch –a`**. Pomoću zastavice -a dobivamo ispis i onih udaljenih grana. Ako maknemo -a, vidjet ćemo samo lokalne grane. Možemo primijetiti da se trenutno nalazimo u main grani. Ako želimo kreirati novu granu, moramo pokrenuti sljedeću naredbu:

```
git branch <naziv_grane>
```

Nakon kreiranja grane želimo se prebaciti u novokreiranu granu. To radimo pomoću naredbe:

```
git checkout <naziv_grane>
```

Nakon ovog koraka spremni smo raditi sve izmjene. Sve napravljene izmjene bit će dostupne samo na toj novokreiranoj grani.

3.13. Naredba: git merge

Nakon završetka rada u novoj grani često želimo te izmjene proslijediti u glavnu main granu. To možemo napraviti pomoću naredbe **`git merge`**. U tom bismo slučaju pokrenuli naredbu **`git checkout main`**, a nakon toga unutar main grane pokrenuli:

`Git merge <naziv_grane>`

3.14. Označavanje koda

U trenutku kreiranja nove funkcionalnosti stvorili bismo sporednu granu unutar koje bismo radili sve izmjene. Kada zaključimo da su funkcionalnosti dovršene, tada bismo u glavnu granu preuzeli sve izmjene iz sporedne. Na taj način imat ćemo ne samo dvije grane, već nekoliko njih. U procesu razvoja imali bismo grane za eksperimente, razvoj funkcionalnosti, ispravljanje grešaka i slično. Nakon završetka svake veće cjeline spojili bismo te grane te iz toga kreirali oznaku kôda. Prisjetimo se priče s početka cjeline, gdje želimo označiti radimo li na verziji programa 1.0 ili 1.0.1. U Gitu za takav proces koristimo tagove. Tag, odnosno oznaka ili ključna riječ, koristi se za hijerarhijsku klasifikaciju. Cilj implementacije tagova u Gitu jest označavanje specifične verzije kôda u vremenu.

3.15. Naredba: git tag

Do sada smo primijetili da kad god upišemo **`git commit`**, Git će dodijeliti jedinstveni identifikator. Međutim, taj niz znakova nije nam od prevelikog značaja. Tag nije ništa drugo nego neki drugačiji naziv za određeni **`git commit`**. U računalnom svijetu to još nazivamo alias. Popis svih trenutno postojećih tagova možemo dobiti pomoću naredbe:

`git tag`

`1.0`

`1.0.1`

Pomoću naredbe **`git tag <naziv_taga>`** dodajemo nove tagove. Nakon što kreiramo tag naš zadatak je i poslati taj tag na udaljeni repozitorij. Cijeli proces bismo napravili ovako:

```
Git tag <naziv_taga>
git push origin --tags
```

Ako bismo u jednom trenutku željeli vratiti kôd na stanje kakvo je bilo u trenutku verzije 1.0, tada bismo se prebacili jednako kao i što bismo se prebacivali između grana.

```
git checkout 1.0
```

Sustav za verzioniranje kôda Git izrazito je bitan u IT zajednici. Vrlo često zahtjevni projekti zahtijevaju veći broj razvojnih timova i u takvom procesu želimo izbjeći konflikte u pisanju kôda. Naposljetku, želimo napraviti oznaku određenog kôda kako bismo znali što šaljemo klijentu. Sve spomenute probleme rješavamo koristeći Git.

PITANJA ZA PONAVLJANJE:

1. Što je Git?
2. Zašto koristiti Git?
3. Koja je razlika između lokalnog i udaljenog Gita?
4. Nabrojite protokole koje koristi Git.
5. Što znači klonirati Git repozitorij?
6. Postoji li razlika između naredbe Git Fetch i Git Pull?
7. Zašto koristimo .gitignore?
8. Zašto je potrebno označavati kôd?

3.16. Povijest Git stanja

U prethodnom poglavlju spomenuli smo izrazito bitnu naredbu **`git log`** – naredbu pomoću koje možemo vidjeti povijest spremljenih datoteka u grani gdje se trenutno nalazimo. No ona nije dovoljna za proučavanje povijesti projekta. Što više budemo koristili Git, primijetit ćemo da povijest promjena može biti izrazito kompleksna. Česti primjer prakse jest taj da želimo saznati promjene koje su se desile prije posljednjeg *commita*. Ili primjerice, želimo vratiti izmjene na određenoj liniji programa na verziju kada je to funkcioniralo. U ovom poglavlju proći ćemo često korištene naredbe za proučavanje povijesti projekta.

3.17. Naredba: git reflog

Pomoću naredbe **`git reflog`** možemo pratiti svaku pojedinu promjenu u određenoj grani ili tagu. Snaga *refloga* dolazi u dijelu gdje zadržava i prati povijest svih grana ili tagova bez obzira na to jesu li izmjene rađene lokalno ili su došle s udaljenog repozitorija. Ovakva značajka izrazito je bitna jer Git može pratiti svaki naš korak, pa ako npr. zabunom obrišemo određenu granu, a kasnije shvatimo da to nismo željeli, pomoću *refloga* možemo vratiti obrisane dijelove koji su nam sada potrebni.

```
git reflog
e28ab5d (HEAD -> main, origin/main) HEAD@{0}: commit: v1.0
024ed91 HEAD@{1}: commit: Feature: Uređivanje korisnika
4627312 HEAD@{2}: commit: Feature: Autorizacija
```

Možemo primijetiti i zaključiti da *reflog* prikazuje sve *commitove*, odnosno datoteke koje su spremne za slanje na udaljeni repozitorij, na koje je pokazivao HEAD. HEAD je zapravo referenca na trenutni *commit*. Ako pričamo o predzadnjem *commitu*, on će dobiti oznaku **`HEAD@{1}`**. Što dalje idemo u povijest, broj, tj. indeks unutar vitičastih zagrada povećava se. U prethodnom poglavlju pričali smo o mijenjanju grana, to možemo postići i *commitovima*. Ukoliko se prebacimo na *commit* koji nije posljednji u grani, tada kažemo da je repozitoriji u *detached* HEAD stanju.

3.18. Naredba: git rebase

U timskom radu može doći do situacije kada mi razvijamo određenu značajku aplikacije u zasebnoj grani, a ostali kolege rade na glavnoj grani. U tom trenutku shvaćamo da su oni možda otišli predaleko s razvojem i da su nam potrebni dijelovi koje su oni ubacili. Često je paralelan razvoj projekta teško pratiti i organizirati. Zato je organizacija repozitorija izrazito zahtjevan postupak i zahtijeva pažnju. Ako bismo željeli preuzeti sve izmjene i pomaknuti mjesto otkud smo granali određenu granu, trebat ćemo koristiti naredbu **`git rebase`**. Prvi zadatak bit će nam prebaciti se u granu koju želimo pomaknuti i pokrenuti naredbu:

```
git rebase <grana>
```

Oznaka **`<grana>`** ona je grana na koju se želimo pomaknuti. U idealnoj situaciju Git će znati riješiti sve probleme i rezultat naredbe izgledat će ovako:

```
git rebase master
First, rewinding head to replay your work on top of it...
Applying: Uređivanje korisnika
Applying: Autorizacija
```

Vrlo se često u ovom koraku znaju pojaviti problemi povezani s konfliktima. Konflikti se pojavljuju ako su drugi autori uređivali jednake datoteke kao i mi. U tom slučaju Git ne zna koje izmjene treba sačuvati i proglasit ih glavnima. Kad se isprave konflikti, nikako ne smijemo napraviti **`commit`**, već samo spremiti izmjene u indeks s naredbom **`git add`**. A nakon spremanja pokrećemo naredbu:

```
git rebase --continue
```

Nakon izvršavanja naredbe **`rebase`** dužni smo odraditi naredbu **`merge`**. Ako imamo previše konflikata, možemo se odlučiti na uzimanje samo dijela izmjena, a ne svih, ili čak i na odustajanje od postupka. U slučaju odabira dijela izmjena to nazivamo *interactive rebase*. Kako bismo saznali koje sve izmjene želimo preuzeti, prvo pokrećemo **`git reflog`** da bismo saznali što sve možemo preuzeti. Nakon odlučivanja što nam sve treba možemo pokrenuti naredbu:

```
git rebase –interactive HEAD~<indeks>
```

<indeks> predstavlja koliki broj *commitova* želimo uzeti. Odustajanje radimo pomoću naredbe:

```
git rebase ––abort
```

„Higijena" i razumijevanje tko je što napravio na repozitoriju u svakom su trenutku izrazito bitni. Jako je važno razumjeti kojim naredbama možemo pratiti što se događa i što se događalo u repozitoriju. Kroz obrađene cjeline naučili smo kako pratiti tko je što napravio i kako u trenutku nastavka razvoja preusmjeriti stanje određene grane.

PITANJA ZA PONAVLJANJE:

1. Koja je razlika između Git Log i Git Reflog?
2. Što predstavlja ključna riječ: HEAD?
3. Kad vidimo detached HEAD prilikom korištenja Gita, što to znači?
4. Koju vrijednost dobivamo korištenjem naredbe Git Rebase?
5. Što napraviti ako postoje promjene u glavnoj grani, a potrebne su nam u našoj odvojenoj grani?
6. Što je potrebno napraviti ako se pojave konflikti prilikom Git Rebasea?
7. Kako zaustaviti Git Rebase?

3.19. Git metodologije integracije promjena različitih autora u finalno rješenje

Kako bismo razvili kvalitetan softver, moramo biti u mogućnosti pratiti sve što se događa tijekom razvoja softvera. Git ispunjava tu ulogu praćenjem povijesti projekta i omogućuje spajanje promjena koje je napravilo više autora. Takav pristup uvelike ubrzava rad i daje nam mogućnost da lakše pronađemo greške. Štoviše, rad u distribuiranim timovima moguć je uglavnom zahvaljujući ovim alatima. Omogućuju da nekoliko ljudi istovremeno radi na različitim dijelovima projekta i kasnije svoje rezultate spoje u jedan proizvod. U procesu spajanja kôda događaju se izrazito bitni događaji. Prvo bismo pisali funkcionalnost aplikacije, testirali je i nakon potvrde od strane tima da je sve uredu izmjene bismo poslali na udaljeni repozitorij. Na taj način napravili bismo zahtjev za povlačenjem kôda. Nakon zahtjeva za povlačenjem koda radi se recenzija naših izmjena. Recenzent može potvrditi naše izmjene ili umetnuti komentare na svaki dio izmjene za koji vjeruje da ga možemo poboljšati ili ga smatra nepotrebnim. U slučaju postojanja komentara kreator može odgovoriti na njih i na taj način pokreće se rasprava, ili može pratiti komentar i u skladu s njim izmijeniti kôd i ponoviti cijeli proces. Razvijanjem zajednice razvilo se više različitih metodologija kolaboracije. Postoje dvije izrazito poznate metodologije, koje sadrže skup pravila koja moramo pratiti.

3.20. Metodologija: Git Flow

U metodologiji Git Flow mora postojati jedna glavna razvojna grana s ograničenim pristupom istoj. Često se takva grana zove: develop. Autori na projektu stvaraju zasebne grane za svaku značajku aplikacije. Nakon što završe proces razvoja stvaraju zahtjev za povlačenje kôda. U zahtjevima za povlačenje kôda ostali dionici komentiraju promjene i vode rasprave na temu izmjena koje su napravljene. Potrebno je neko vrijeme za dogovor oko konačne verzije promjena. Nakon što je to dogovoreno, zahtjev za povlačenjem prihvaća se i spaja s granom develop. Nakon određenog vremena i razvijanja značajki, vrijeme je da se odluči je li develop grana dostigla dovoljan stupanj razvoja za puštanje kôda u produkciju, stvara se zasebna grana s nazivom verzije za pripremu konačne verzije. Aplikacija se iz grane s nazivom verzije testira i primjenjuju se ispravci pogrešaka do trenutka kad je aplikacija spremna za objavljivanje krajn-

jim korisnicima. Nakon što je to učinjeno, spajamo konačni proizvod s main granom i označavamo ga izdanom verzijom. Na develop grani u međuvremenu se razvijaju nove značajke. Jedna od prednosti Git flowa jest stroga kontrola. Samo ovlašteni programeri mogu odobriti promjene nakon što ih pomno pregledaju. Takvim pristupom osiguravamo kvalitetu kôda i pomažemo u ranoj eliminaciji grešaka. Međutim, moramo imati na umu da to također može biti veliki nedostatak. Takav pristup može uvelike usporiti razvoj softvera. Ako je brzina primarna briga, onda bi to mogao biti ozbiljan problem jer projekt zahtijeva održavanje, a s obzirom na velik broj grana, ponekad revizija onoga što smo napravili može dosta dugo potrajati.

3.21. Metodologija: Trunk-based

U ovom slučaju svi programeri rade na jednoj grani s otvorenim pristupom, odnosno ne postoje ograničenja kao u GitFlowu. Često je to jednostavno main grana. Svi suradnici šalju izmjene na glavnu granu bez posebnih ograničenja. U slučajevima kada je izmjena izrazito velika i ovdje se stvaraju zasebne grane. Također, stvaramo grane u slučaju integracija s alatima koji testiraju kôd. U takvim slučajevima, ako radimo izrazito velike značajke, ovakav pristup olakšava posao jer se provjeravaju točno određene značajke iz kojih možemo dobiti izvještaje, a nakon toga te se značajke spajaju u main granu. Ovakav pristup osigurava da razvoj bude kontinuiran i sprečava programere da stvaraju konflikte u kôdu, koje je ponekad teško riješiti. Jedini način pregleda kôda u takvom pristupu jest pregled cijelog izvornog kôda, a ne samo određenih izmjena. Duge rasprave obično su ograničene. Nitko nema strogu kontrolu nad onim što se mijenja u repozitoriju i zato je važno imati jednak stil pisanja kôda. Programeri koji rade u takvom stilu trebali bi imati iskustva kako ne bi narušili kvalitetu izvornog kôda. Ovakav pristup izrazito je otežan u neiskusnim timovima.

U izrazito brzom razvoju koji prati mogućnosti brzih izmjena postoje različiti pristupi problemima; vodeći se tom mišlju razvojni timovi stvorili su metodologije vođenja udaljenih repozitorija. U ovom poglavlju možemo zaključiti da ovisno o veličini tima i agilnosti trebamo odabrati ispravan pristup vođenja repozitorija.

PITANJA ZA PONAVLJANJE:

1. Zašto je bitna implementacija različitih metodologija u vođenju repozitorija?
2. Kako funkcionira metodologija Git Flow?
3. Koliko različitih grana postoji u metodologiji Git Flow?
4. U kojem smo trenutku sigurni da je grana spremna za produkciju u metodologiji Git Flow?
5. Kako funkcionira metodologija Trunk-based?
6. Koliko različitih grana postoji u metodologiji Trunk-based?
7. U kojem smo trenutku sigurni da je grana spremna za produkciju u metodologiji Trunked-based?
8. Koja je metodologija u kojem slučaju bolja?

4. Kontejner aplikacije

U OVOM POGLAVLJU NAUČIT ĆETE:

- razlikovati virtualizirano i fizičko okruženje
- što su to kontejner aplikacije i kada se koriste
- kako se aplikacije kreiraju u Docker obliku
- koncepte Docker sustava
- kada se koristi Kubernetes
- koji se tipovi servisa u Kubernetes sustavu najviše koriste
- razlikovati Kubernetes resurse.

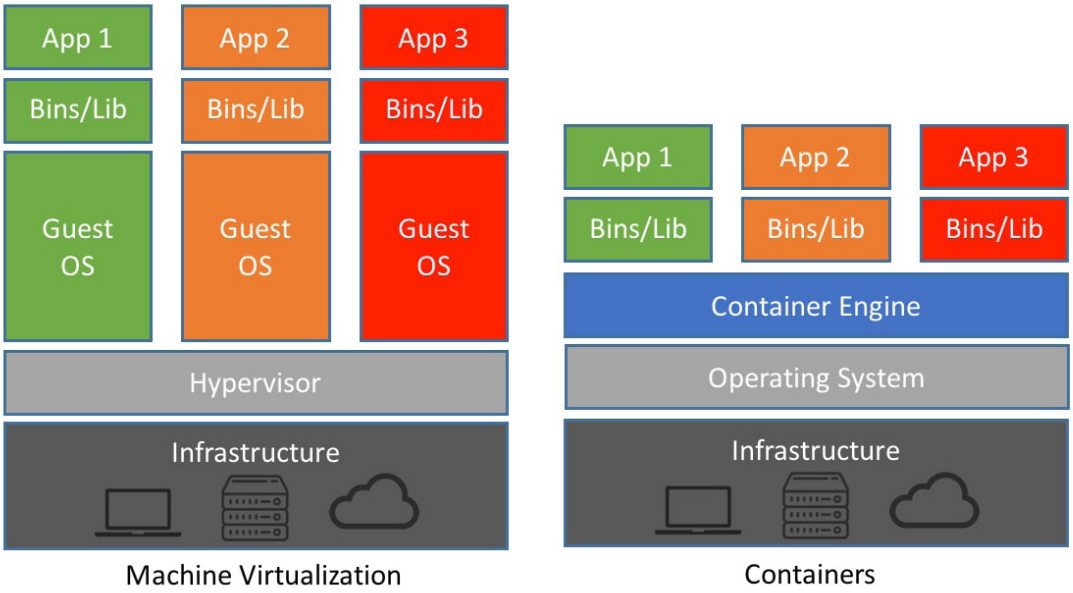
4.1. Osnove virtualizacije i oblikovanja aplikacija i usluga kroz kontejnerizaciju

Virtualizacija zapravo omogućuje dijeljenje resursa jednog fizičkog poslužitelja između više virtualnih poslužitelja. Virtualna mašina dobit će određen broj virtualnih procesora, virtualnu memoriju i virtualni disk. Fizički poslužitelji mogu imati više od jednog fizičkog procesora; često imaju dva fizička procesora, ali mogu ih imati i više. Svaki procesor ima određen broj jezgri. Broj jezgri procesora ovisi o modelu procesora. Jedna fizička jezgra može sadržavati i više od 30 jezgri. Svaki poslužitelj zauzima određeno mjesto u ormaru za računalnu opremu te se svaki poslužitelj sastoji od više komponenti. Poslužitelji uz procesor i memoriju imaju matičnu ploču koja omogućuje međusobnu interakciju komponenti, napajanje koje omogućuje rad poslužitelja,

mrežne kartice za komunikaciju s ostatkom infrastrukture, diskove za pohranu podataka i sustav za hlađenje. S rastom broja poslužitelja raste i broj popratnih komponenti, koje nisu nužno u potpunosti iskorištene. Poslužitelji s procesorima s manjim brojem jezgri i dalje moraju imati sve navedene komponente, a broj aplikacija i virtualnih mašina koje mogu opsluživati manji je. Zbog toga je optimalno imati manje poslužitelj s više resursa. Iz pogleda dostupnosti, poželjno je imati više poslužitelja s manje resursa kako kvar jednog poslužitelja ne bi utjecao na rad ostalih. Ukoliko tvrtke i dalje žele imati pristup jedan poslužitelj jedan OS i ignorirati virtualizaciju, njihova infrastruktura sastojala bi se od puno više poslužitelja s puno manjim opterećenjem. Optimalno opterećenje poslužitelja je oko 70 % kako bi imao zalihu resursa u slučaju dodatnog opterećenja. S pristupom jedan poslužitelj jedan OS bilo bi teško dizajnirati sustave toliko precizno. Ukoliko virtualna mašina nema dovoljno resursa ili ima prenisku iskorištenost, njezina veličina može se lako promijeniti, što nije jednostavno kod fizičkih poslužitelja. Ako dođe do nedostupnosti fizičkog poslužitelja, sve virtualne mašine također će biti nedostupne. Osnovne mogućnosti virtualizacijskih rješenja slične su, ali napredne mogućnosti mogu se uvelike razlikovati između proizvođača softvera za virtualizaciju.

Operativni sustav koji se izvršava u virtualnom okruženju, tj. u virtualnoj mašini, u potpunosti je odvojen od OS-a virtualizatora. Ukoliko izmijenimo bilo koju postavku unutar OS-a virtualne mašine, ta postavka neće biti promijenjena na virtualizatoru. Zbog toga virtualizator može, na primjer, imati instaliran Windows Server OS, a virtualna mašina Ubuntu OS. Svaka virtualna mašina zauzimat će određenu količinu resursa za potrebe samog operativnog sustava virtualne mašine neovisno o instaliranoj aplikaciji. Primjeri virtualizatora jesu Microsoft Hyper-V, VMware ESXi, Xen i qemu. Pružatelji usluga u oblaku pružaju uslugu izvođenja virtualnih mašina, ali konfiguracijom njihovih virtualizatora nije moguće upravljati.

Virtualizacija omogućuje kreiranje virtualnih mašina, dok kontejnerizacija omogućuje kreiranje kontejnera (engl. *container*) za potrebe posluživanja aplikacije. Razvoj virtualizacije započeo je šezdesetih godina dvadesetog stoljeća u IBM-ovu znanstvenom centru u Cambridgeu s CP-40 sustavom. Virtualizacija sustava x86, odnosno 32-bitne i kasnije 64-bitne arhitekture razvila se tek kasnih 90-ih godina dvadesetog stoljeća. Kako je rasla važnost virtualizacije za velike tvrtke, proizvođači OS-a i procesora sve su više ulagali u razvoj tehnologije.



Slika 4.1.1. Usporedba virtualizacije i kontejnerizacije

Prvi kontejneri bili su Solaris kontejneri, koji su izašli u beta verziji 2004. godine. Daljnjim razvojem tehnologije kreirana je virtualizacijska metoda LXC (Linux Containers). LXC omogućuje pokretanje više izoliranih kontejnera na jednom poslužitelju koristeći samo jednu jezgru OS-a. LXC koristi mogućnosti kontrolnih grupa i imenskih prostora kako bi izolirao kontejnere od OS-a. Na LXC-u kreiran je Docker 2013. godine. Kontrolne grupe (engl. *cgroups*) omogućuju praćenje i izoliranje korištenja resursa kao što je CPU i memorija. Imenski prostori (engl. *namespaces*) omogućuju apstrakciju globalnog resursa na način da procesima unutar imenskog prostora izgleda da oni imaju vlastitu izoliranu instancu globalnog resursa. Kao što je vidljivo na slici 4.1.1., svaka virtualna mašina ima vlastiti OS, dok kontejneri dijele OS.

4.2. Kontejnerski operativni sustavi

Kontejneri za razliku od virtualnih mašina dijele jezgru OS-a s virtualizatorom. U praksi to znači da dokle god jezgra OS-a virtualizatora zna ispravno odgovoriti na zahtjev kontejnera, moguće je pokrenuti kontejner baziran na jednom OS-u na virtualizatoru drugog OS-a. Nije moguće na Windows Serveru pokrenuti Linux kontejnere direktno. Primjeri rješenja za kontejnerizaciju jesu Docker i Podman. Pružatelji usluga u oblaku također imaju svoja rješenja za pokretanje kontejnera.

Određeni operativni sustavi prilagođeni su tako da se temelju njih kreiraju slike kontejnera. Od njih se kreiraju bazne slike na koje se nadograđuju druge slike. Karakterizira ih mala veličina i vrlo mali skup uključenih biblioteka. Alpine Linux jedna je od Linux distribucija na temelju koje se kreiraju slike kontejnera. Ima samo 5 MB. Operativni sustav koji bi nakon instalacije zauzimao npr. 5 GB svakako ne bi bio primjeren za upotrebu kao kontejner. Windows Server također nudi slike za kontejnere bazirane na Windows Server OS-u. Zbog male veličine kontejnera idealni su za mikroservise. Nove instance kontejnera kreiraju se iznimno brzo i samim time skaliranje je puno brže.

Samim kontejnerskim sustavima moguće je upravljati, ali zbog prirode kontejnera koja će biti objašnjena u sljedećim poglavljima najbolje je upravljati samim slikama kako bi sve promjene bile spremljene kod kreiranja novog kontejnera od najnovije slike. U produkcijskim okruženjima nije uobičajeno upravljati OS-om kontejnera direktno, već se to radi najčešće za potrebe istraživanja grešaka.

4.3. Virtualizacija aplikacija

Većina aplikacija može biti pretvorena u kontejnerski oblik. Ovisno o programskom kodu aplikacije aplikacija može zahtijevati izmjene kako bi bila optimalna za izvođenje u kontejneru. Ukoliko aplikacija koristi sistemske pozive koji nisu podržani u svim verzijama jezgre OS-a, kontejner će se moći izvršavati isključivo na virtualizatorima čija jezgra OS-a podržava te sistemske pozive. Ako je aplikacija napisana za određenu platformu kao što su Tomcat i Joomla ili u programskom jeziku koji nije sadržan u baznoj slici, potrebno je kreirati novu sliku kontejnera koja će sadržavati sve potrebne ovisnosti za uspješno izvršavanje aplikacije. Aplikacije koje čuvaju stanje (engl. *stateful*) zahtijevaju više rada kako bi se uspješno kontejnerizirale jer je potrebno osigurati pohranjivanje stanja kod zamjene kontejnera novijom verzijom. Kontejneri imaju puno kraći životni ciklus i zamišljeni su na način da se lako zamijene drugim kontejnerom. Baze predstavljaju tip aplikacija koji se ne koristi često u kontejnerskom obliku u produkcijskim okolinama zbog potrebe očuvanja podataka. U razvojnim okolinama ili za potrebe testova na laptopu razvojnog inženjera baze se često koriste u obliku kontejnera.

Standardna je praksa kreirati sliku za aplikaciju prema Dockerfileu, datoteci koja specificira kako kreirati sliku kontejnera. U CI (engl. *Continuous Integration*) procesu jedan od testova može biti kreiranje slike kontejnera. Nakon kreiranja slike aplikacija se može i automatski testirati u testnoj okolini. Automatizacijom procesa kreiranja slike kontejnera – postavljanja kontejnera u razvojnu i produkcijsku okolinu – uvelike se umanjuje rizik različite konfiguracije usluge na razvojnim i produkcijskim sustavima. Korištenjem konfiguracije u obliku koda lako je pronaći razlike u samoj konfiguraciji.

Kontejnerski operativni sustav zapisan je u slici kontejnera. Zbog toga se ažuriranja instaliraju u sliku kontejnera. No sliku kontejnera moguće je izmijeniti samo tijekom njene kreacije te je zbog toga potrebno kreirati novu sliku kontejnera koja će sadržavati instalirana ažuriranja OS-a i instaliranih paketa. Moguće je koristiti bazne slika čiji održavatelj osigurava instalaciju najnovijih ažuriranja OS-a. Instalaciju najnovijih ažuriranja moguće je dodatno osigurati dodavanjem koraka instalacije ažuriranja u konfiguracijsku datoteku za kreiranje slike kontejnera. Ovisno o korištenom sustavu za repozitorije moguće je automatizirati proces kreiranja novih slika aplikacije nakon što se kreira nova bazna slika. Virtualizacija je omogućila iskorištavanje većeg postotka resursa dostupnog na poslužiteljima i samim time stvorila potrebu za poslužiteljima s više dostupnih resursa. Virtualizacija uz sve svoje prednosti dolazi s nedostatkom da virtualne mašine ipak trebaju par minuta za pokretanje i određenu količinu resursa za operativni sustav. Kontejneri su manje izolirani od operativnog sustava virtualizatora zbog dijeljenja jezgre OS-a, ali se zbog toga pokreću brže i zahtijevaju manje resursa.

4.4. Repozitorij

Slike kontejnera pohranjene su u repozitorijima slika. Registar se sastoji od više repozitorija. Repozitorij se sastoji od više slika s istim imenom i različitom oznakom (engl. tag). Uz oznaku slike jedinstveno su identificirane prepoznate svojim sažetkom (engl. *hash*). Ukoliko se slika s istom oznakom nalazi u više repozitorija, potrebno je navesti ime repozitorija prilikom refenciranja u obliku **`<registar>/<imenski prostor>/<slika>:<oznaka>`**, npr. **`dockerhub/httpd:latest`**. Određeni registri ne zahtijevaju navođenje imenskog prostora.

Registri mogu biti privatni i javni. Privatni registri zahtijevaju autentifikaciju prilikom pristupa. Neki od repozitorija jesu Docker Hub, JFrog Artifactory i Red Hat Quay. Pružatelji usluga u oblaku također nude repozitorije kao uslugu. JFrog Artifactory često se koristi u velikim okruženjima kao lokalni registar. Registri se međusobno razlikuju prema cijeni, načinu naplaćivanja, mogućnostima autentifikacije, mogućnostima skeniranja slika za ranjivosti i mogućnosti brisanja nepotrebnih datoteka.

Open Container Initiative (skraćeno OCI) standard je za format slika kontejnera i okolina za izvođenje. Omogućuje međusobnu kompatibilnost različitih alata za rad s kontejnerima.

PITANJA ZA PONAVLJANJE:

1. Što omogućuje virtualizacija?
2. Koja je razlika između virtualizacije i pristupa jedan OS jedan poslužitelj?
3. Navedi nekoliko rješenja za virtualizaciju.
4. Navedi prednosti podešavanja pojedinih dijelova sustava u virtualnom obliku.
5. Što je kontejner?
6. Mogu li se sve aplikacije pretvoriti u kontejnerski oblik?
7. Navedi nekoliko rješenja za kontejnerizaciju?
8. Koliko je kontejnera moguće imati na jednom poslužitelju?
9. Za što se koriste repozitoriji za kontejnere?
10. Može li RHEL poslužitelj pokretati kontejnere bazirane na Ubuntu OS-u? Ako da, zašto?

4.5. Sustav za kontejniziranje „Docker”

Docker projekt započeo je kao interni projekt unutar tvrtke dotCloud, koja svojim korisnicima nudi platformu kao servis. Kreirao ga je Solomon Hykes 2010. godine. Solomon je s Kamelom Founadijem i Sebastienom Pahlom osnovao tvrtku Docker Incorporated i nastavio njegov razvoj. Docker se izdvojio od ostalih natjecatelja na području kontejnerizacije svojim kompletnim ekosustavom za upravljanje kontejnerima. Kroz partnerstva i suradnje s Red Hatom, Microsoftom i AWS-om Docker je postao jedno od glavnih rješenja za kontejnerizaciju.

Kontejneri se izvršavaju pomoću sustava za izvršavanje kontejnera (engl. *container engine*). Tipični zadaci o kojima se brine sustav za izvršavanje kontejnera zaprimanje je korisničkih zahtjeva, odnosno naredbi, dohvaćanje slika te pretvaranje slika u kontejnere. Za samo puštanje kontejnera u rad brine se sustav izvršnog okruženja kontejnera (engl. container runtime).

Docker je nastao kao proširenje mogućnosti Linux kontejnera. Do verzije 0.9 koristio je LXC za izvršavanje kontejnera. Od verzije 0.9 prestao je koristiti LXC i počeo koristiti vlastitu pomoćnu datoteku za izvršavanje kontejnera libcontainer . libcontainer je razvijen kako bi zamijenio LXC zbog toga što je glavna svrha LXC-a bila kreiranje više izoliranih virtualnih Linux okolina. Virtualne Linux okoline omogućile bi izvršavanje više Linux OS-ova na jednom Linux poslužitelju. Glavna svrha Docker kontejnera izvršavanje je jedne aplikacije u izoliranoj okolini. dotCloud tvrtka izdala je Docker pod licencom otvorenog koda 2013. godine. Svojom arhitekturom Docker kontejneri idealni su za mikroservise. Zbog iznimno male veličine kontejnera i dijeljenja jezgre OS-a s OS-om poslužitelja iznimno se brzo pokreću i skaliraju.

Docker u trenutku pisanja za svoj rad zahtijeva:

- 64-bitni OS
- Linux kernel verzije 3.10 ili više
- iptables verzije 1.4 ili više
- git verzije 1.7 ili više
- ps alat
- XZ Utils verzije 4.9 ili više
- ispravno mapiranu cgroups hijerarhiju.

Za popularne distribucije kao što su CentOS, Debian, Ubuntu ili RHEL dovoljno je koristiti podržanu verziju OS-a i instalirati Docker putem paketa te će svi zahtjevi biti zadovoljeni, odnosno alat za upravljanje paketima instalirat će potrebne pakete ukoliko neki nedostaje.

Komponente Dockera jesu:

- Docker nadzorni proces (engl. daemon)
- Docker klijent
- Docker Desktop
- Docker registri
- Docker objekti.

Nadzorni proces Docker sluša za API zahtjeve i prema zahtjevima upravlja Docker objektima. Docker klijent, odnosno Docker konzolni alat upućuje API zahtjeve nadzornom procesu ovisno o zadanoj naredbi i njenim argumentima. Docker registri služe za pohranu Docker slika (engl. *images*), odnosno slika kontejnera. Docker repozitoriji mogu biti javni ili privatni. Javni repozitoriji sadrže javno dostupne slike, dok privatni repozitoriji sadrže slike kojima je pristup ograničen samo na određene korisnike i zahtijeva autentifikaciju. Pod Docker objekte spadaju slike, kontejneri, mreže, volumeni, dodaci i ostali objekti koje Docker kreira i s kojima radi prilikom izvršavanja kontejnera. Docker Desktop aplikacija je koja se sastoji od Docker nadzornog procesa, Docker klijenta, alata Docker Compose, Kubernetes klastera od jednog člana i pomoćnog alata za rad s vjerodajnicama. Docker Compose alat je za rad s tzv. „dockeriziranim" aplikacijama, koje se sastoje od više kontejnera. Kubernetes sustav bit će objašnjen u jednom od sljedećih poglavlja. Docker Desktop predstavlja rješenje koje omogućuje razvojnim inženjerima jednostavan rad s kontejnerima i kontejnerizaciju aplikacija.

Nadzorni proces Docker može se konfigurirati na dva načina: editiranjem konfiguracijske datoteke ili korištenjem zastavica i argumenata prilikom pokretanja samog procesa. Ukoliko je ista opcija konfigurirana putem argumenata i konfiguracijske datoteke, proces se neće pokrenuti. Nadzorni proces Docker najčešće se konfigurira za automatsko pokretanje prilikom pokretanja računala.

Jedna od najpopularnijih alternativa Dockeru jest Podman. Podman je alat bez nadzornog procesa otvorenog koda sukladan s OCI standardom za pronalaženje, pokretanje, kreiranje i puštanje u rad kontejnera i kontejnerskih slika. Sintaksa mu je gotovo identična Dockerovoj. Podman podržava i rad s kapsulama (engl. *pod*). Kapsula je grupa jednog ili više kontejnera s dijeljenom pohranom i mrežnim resursima, odnosno kontejneri u kapsuli dijele određene imenske prostore. CRI-O je kao i Podman alternativa Dockeru, ali njegova primarna namjena nije da ga koriste administratori i razvojni inženjeri za razvoj na vlastitom računalu, već da bude sustav za izvršavanje kontejnera u Kubernetes klasteru.

4.8. Izrada Docker slika

Docker slika predložak je od kojeg se kreira kontejner. Sastoji se od više slojeva. Slojevi i sama slika nepromjenjivi su. Nije moguće editirati sloj slike, već je potrebno kreirati novi sloj. Slojevi su međusobno povezani na način da se jedan nastavlja na drugi i svaki sloj sadrži samo promjenu od prošlog sloja. Slojevi sadrže datoteke i mape potrebne kontejneru za rad. Slika kontejnera kreira se prema uputama zapisanima u datoteci Dockerfile. Svaka instrukcija u uputi kreira novi sloj. Kontejner se kreira od slike njezinim izvršavanjem. Nakon što je kontejner pokrenut dodaje mu se privremeni sloj na koji je moguće zapisivati. Podaci pohranjeni u privremenom sloju izgubljeni su ukoliko se kontejner obriše ili zamijeni drugom kontejnerom.

Preživljavanje podataka moguće je osigurati mapiranjem mapa na samom poslužitelju na određenu mapu u kontejneru (engl. *bind-mount*) ili korištenjem zapremnina (engl. *volume*). Preporučeno je koristiti zapremnine. Zapremnine omogućuju pohranjivanje podataka na samom poslužitelju, dijeljenom direktoriju ili u oblaku korištenjem upravljačkih programa za zapremnine. Kako bi kontejnerizirali aplikaciju, servise od kojih se ona sastoji potrebno je izdvojiti u zasebne kontejnere. Na primjer, aplikaciju koja se sastoji od frontenda u Reactu, backenda u Javi i koristi bazu MySql za pohranu podataka potrebno je podijeliti u više kontejnera. Baza može, ali i ne mora biti kontejnerizirana, ali *frontend* i *backend* moraju se podijeliti u zasebne kontejnere.

Docker slike mogu biti referencirane putem imena i oznake ili jedinstvenog sažetka.

Slika se sastoji od više slojeva, od kojih svaki sloj ima jedinstveni sažetak, no određene slike mogu se koristiti s različitim arhitekturama procesora. Različite arhitekture procesora zahtijevaju različitu programsku podršku. Kako bi se prilikom preuzimanja slike odabrali točni slojevi za određenu arhitekturu i za željenu verziju slike, svaka verzija slike ima manifest. Manifest može definirati druge manifeste za određenu arhitekturu. Manifest sadrži konfiguracijsku datoteku za sliku i jedinstvene sažetke slojeva slike. Konfiguracijsku datoteku slike koristi sustav izvršnog okruženja kontejnera kako bi kreirao kontejner. Drugi način refenciranja Docker slika jest putem imena, no sve verzije slika za istu aplikaciju imaju isto ime. Kako bi se razlikovalo različite verzije slike, slike su dodatno označene jednom ili više oznaka (engl. tag). Jedna od najpoznatijih oznaka jest oznaka latest, no nije nužno da ona uvijek upućuje na zadnju verziju slike. Oznaka latest može označavati i zadnje izdanje softvera.

Slike kontejnera kreiraju se prema uputama sadržanima u kompozicijskim datotekama. Dockerfile je naziv za kompozicijske datoteke namijenjene kreiranju Docker kontejnera. Svaka naredba u kompozicijskoj datoteci predstavlja kreiranje jednog sloja, dok je svaki sloj razlika od prošlog sloja. Važno je napomenuti da prilikom pohrane slika, ukoliko se viša slika sastoji od identičnih slojeva, Docker neće dva puta preuzeti i pohraniti isti sloj, već će prepoznati da se radi o istim slojevima i uštedjeti na prostoru. Jedna od najčešće korištenih instrukcija u kompozicijskim datotekama jest instrukcija **RUN**. Instrukcija **RUN** izvršit će sve naredbe koje su joj predane. Za izvršavanje više uzastopnih naredbi moguće ih je povezati korištenjem operatora **&&** za spajanje naredbi te smanjiti ukupan broj korištenih **RUN** instrukcija. Smanjivanjem broja korištenih RUN instrukcija smanjen je i ukupan broj kreiranih slojeva.

Dockerov mrežni sustav modularan je i podržava korištenje više upravljačkih programa za ostvarivanje različitih mrežnih konfiguracija. Most (engl. *bridge*) je zadani mrežni upravljački program. Moguće je kreirati više mostova. Svaki most omogućuje međusobnu komunikaciju samo kontejnerima koji su povezani na taj most. Host mrežni upravljački program omogućuje kontejneru da koristi mrežna sučelja poslužitelja tijekom komunikacije. Primjer korištenja host vrste mrežnog upravljačkog programa pokretanje je nginx kontejnera koji sluša na TCP portu 80. nginx je web-poslužitelj otvorenog koda koji uz funkciju web-poslužitelja može pružiti funkcionalnost obrnutog proxyja, raspoređivača prometa (engl. *load balancer*) te za potrebe pohranjivanje HTTP privremene memorije (engl. *HTTP cache*). Ukoliko nginx kontejner koristi host

mrežni upravljački program, na sve mrežne zahtjeve koji pristignu na poslužitelj koristeći TCP port 80 odgovorit će kontejner. Uz navedene vrste mrežnih upravljačkih programa postoje i drugi, svaki sa svojom funkcijom. Docker podržava i korištenje dodataka trećih strana za proširenje mrežnih funkcionalnosti.

Varijable okoline dinamične su vrijednosti kojima OS i druge aplikacije mogu pristupiti. Često se koriste za privremenu pohranu različitih tajni kako one ne bi bile trajno spremljene na disk. Docker okruženje podržava dvije vrsti varijabli okoline. ARG varijable okoline dostupne su prilikom procesiranja Dockerfilea i kreiranja slike. Nakon što je sama slika kreirana i kasnije kontejner kreiran, one više nisu dostupne. **`ENV`** varijable okoline dostupne su prilikom izgradnje slike kontejnera te prilikom pokretanja kontejnera od njegove slike. **`ARG`** i **`ENV`** ujedno su i instrukcije kojima se te varijable podešavaju. Prilikom izgradnje slike vrijednost **`ENV`** varijabli nije moguće promijeniti, a vrijednost **`ARG`** varijabli moguće je promijeniti prosljeđivanjem argumenta **`--build-arg`** te nazivom i vrijednosti **`ARG`** varijable prilikom pozivanja docker **`build`** naredbe.

Kako bi se optimizirao proces kreiranja Docker slika jedna od preporuka svakako je korištenje zajedničke bazne slike koja će sadržavati instalirana ažuriranja. Nakon instalacije ažuriranje čišćenje privremeno pohranjenih paketa smanjit će veličinu krajnje slike. Kombiniranje više naredbi **`&&`** operatorom omogućit će manji broj slojeva. Prilikom kreiranja slike od svakog sloja kreira se novi kontejner, izvrši se zadana instrukcija te se od promjene na datotečnom sustavu kreira novi sloj. Određene instrukcije ne uzrokuju kreiranje novog kontejnera, već samo kreiraju novi sloj ako nema promjena na datotečnom sustavu. Manji broj slojeva i korištenje zajedničke bazne slike omogućuje i brže kreiranje same slike.

Kreiranje Docker slike u više razina (engl. *multi-stage build*) omogućuje kreiranje više kontejnera, od kojih svaki ima različitu baznu sliku i kopiranje artefakata između njih. U praksi to omogućuje kreiranje prvog kontejnera u kojemu će se instalirati svi potrebni alati za kompiliranje i proizvesti izvršna datoteka samog koda aplikacije. Nakon toga kreirat će se drugi kontejner u koji će se kopirati izvršna datoteka iz prvog kontejnera. Ovim pristupom smanjena je veličina slike kontejnera koji će se koristiti u produkciji te sam kontejner sadrži puno manje nepotrebnih datoteka i postavki. Bez kreiranja Docker slike u više koraka Dockerfile bi sam po sebi morao biti puno kompleksniji

kako bi se uklonili podaci i alati potrebni za kompiliranje te ne bi bilo moguće koristiti različitu baznu sliku. Smanjenjem veličine krajnje slike kontejner će se brže pokrenuti, a prilikom pokretanja na poslužitelju gdje slika nije preuzeta brže će se preuzeti.

Docker sustav za kontejnerizaciju omogućio je mikroservisnu arhitekturu s manje potrebnih resursa, bržim skaliranjem i pokretanjem aplikacija te je uvelike olakšao razvoj softvera razvojnim inženjerima na vlastitim laptopima. Mikroservisna arhitektura nezamisliva je bez tehnoloških rješenja koje je donijela kontejnerizacija. Svojim mogućnostima Docker omogućuje efikasno kreiranje slika i dizajniranje arhitekture utemeljene na kontejnerima.

PITANJA ZA PONAVLJANJE:

1. Navedi nekoliko sustava za kontejniziranje aplikacija.
2. Objasni Docker slike.
3. Objasni oznake Docker slika.
4. Kako slika postaje kontejner?
5. Kako se kreiraju Docker slike?
6. Kako se zove kompozicijska datoteka za Docker?
7. Čemu služi Docker zapremnina?
8. Objasni varijable okoline dostupne prilikom izrade slike.
9. Objasni varijable okoline dostupne prilikom izvršavanja aplikacije.
10. Kako smanjiti broj kreiranih slojeva slike za naredbu RUN?

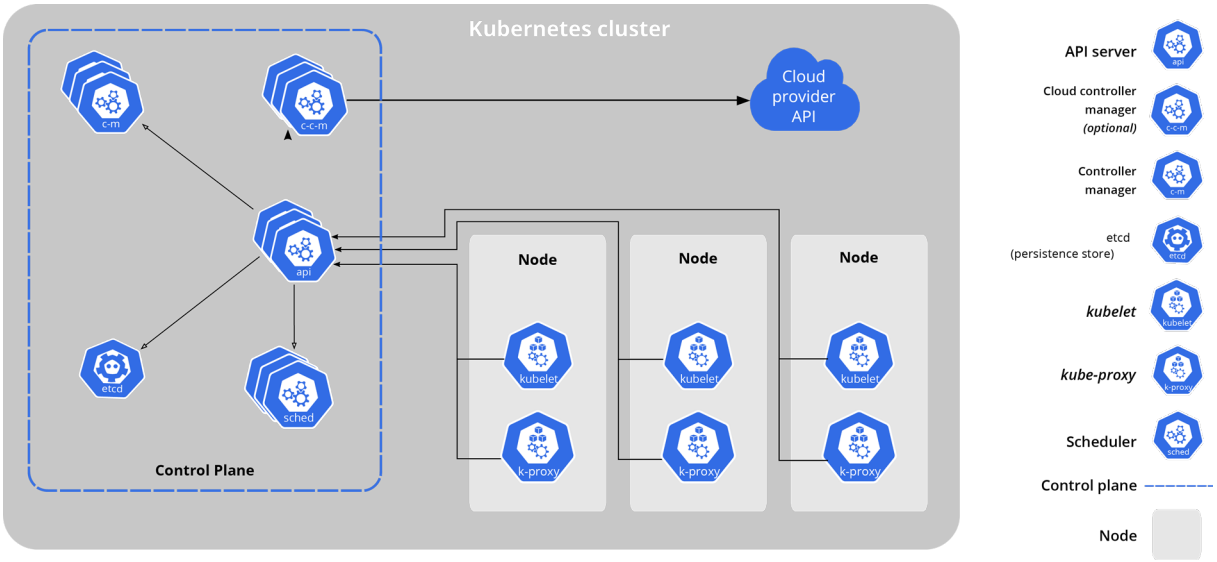
4.11. Sustav „Kubernetes”

Docker kao rješenje idealan je za upravljanje kontejnerima nekoliko aplikacija na jednom računalu ili poslužitelju. No što ako želimo upravljati kontejnerima na stotinama ili tisućama računala? Tijekom upravljanja većim brojem kontejnera želimo povezati više kontejnera i popratnih objekata u razumljiviju cjelinu koja bolje predstavlja određenu aplikaciju. To nam omogućuju orkestratori kontejnera.

Kubernetes (skraćeno k8s) je orkestrator kontejnera otvorenog koda. Inicijalno je razvijan u Googleu i utemeljen je na njihovu iskustvu prilikom kreiranja internog Borg upravitelja klastera. Verzija 1.0 izdana je 21.7.2015. godine.

Docker je sustav za izvršavanje kontejnera, a Kubernetes je sustav za orkestraciju kontejnera. Docker također ima svoj orkestrator Docker Swarm. Kubernetes nije rješenje za izvršavanje kontejnera, već će se za izvršavanje pobrinuti instaliran sustav za orkestraciju kontejnera. Od verzije 1.20 Kubernetes ne podržava Docker kao sustav za izvršavanje kontejnera zbog Dockerove nekompatibilnosti sa sučeljem za izvršavanje kontejnera (engl. *Container Runtime Interface*). Docker je idealan za pokretanje kontejnera na individualnom računalu prilikom razvoja, ali za upravljanje kontejnerima u produkcijskom okruženju potreban je sustav za orkestraciju kontejnera.

Slika kontejnera kreirana putem Dockera može biti korištena i u Kubernetesu zbog toga što Docker kreira slike kontejnera sukladno OCI standardu. Generalno nisu potrebne veće izmjene aplikacija prilagođenih za izvođenje u Dockeru prilikom prilagodbe za izvršavanje u Kubernetes okruženju. Veća količina rada potrebna je za samu konfiguraciju izvođenja aplikacije unutar Kubernetes sustava.



Slika 4.11.1. Arhitektura sustava Kubernetes

Pristupano: https://kubernetes.io/docs/concepts/overview/components

Na slici 4.11.1. prikazani su osnovni elementi Kubernetes sustava odnosno klastera. Osnovne komponente Kubernetes sustava jesu:

- API poslužitelj (engl. *API server*)
- upravitelj kontrolera (engl. *Controller Manager*)
- raspoređivač (engl. *Scheduler*)
- etcd
- Kubernetes član klastera (egl. *Node*)
- kubelet
- Kapsula (engl. *Pod*)
- sustav izvršnog okruženja kontejnera (engl. *container runtime*)
- kube-proxy
- dodana mreža (engl. *Plugin Network*).

API poslužitelj predstavlja sučelje za komunikaciju s Kubernetes klasterom putem REST-a. Svi zahtjevi za dohvatom ili promjenom stanja klastera moraju se uputiti API poslužitelju. Zahtjeve za dohvatom ili promjenom stanja klastera mogu uputiti administratori, korisnici i ostale komponente klastera. Klaster se sastoji od više različitih objekata koji predstavljaju njegove elemente i konfiguraciju.

Stanje klastera i svi podaci o klasteru pohranjeni su u bazi etcd te sve izmjene stanja u etcd-u vrši API poslužitelj. etcd je visoko dostupna, distribuirana, konzistentna baza podataka za pohranu podataka u obliku ključ – vrijednost.

Kontroleri dohvaćaju i mijenjanju stanje klastera kroz API poslužitelj kako bi trenutno stanje klastera doveli do željenog stanja. Administratori definiraju željeno stanje klastera. Upravitelj kontrolera komponenta je klastera zadužena za pokretanje svih kontrolera.

Većina kapsula kod kreiranja nema zadan član klastera na kojemu će se pokrenuti, već ima definiran niz uvjeta koji moraju biti zadovoljeni kako bi kapsula mogla biti pokrenuta na određenom članu klastera. Raspoređivač na temelju zahtjeva kapsule pronalazi odgovarajući član klastera. Kapsule mogu imati jednu ili više oznaka (engl. *label*) koje služe za logičko grupiranje kapsula.

Kubelet je agent koji se nalazi na svakom članu klastera. Prima definicije kapsula koje mora kreirati i nadzirati. Definiciju kapsule najčešće šalje API poslužitelj.

Kapsula je najmanja jedinica za puštanje u rad koju poznaje Kubernetes. Sastoji se od jednog ili više kontejnera koji dijele imenski prostor. Za pokretanje kontejnera odgovoran je sustav izvršnog okruženja kontejnera.

Kube-proxy održava mrežna pravila za ostvarivanje funkcije Kubernetes servisa. Uz kubelet instalira se na svakom članu klastera. Mrežnu komunikaciju kapsula unutar i izvan klastera omogućuju dodaci za mrežna sučelja kontejnera (engl. *Container Network Interface add-on*). Dodaci za mrežna sučelja kontejnera kreiraju sloj mreže nazvan nadmrežom (engl. *overlay network*) u kojemu kapsule dobivaju svoje IP adrese i međusobnu komuniciraju. Svaki dodatak za mrežna sučelja kontejnera realizira nadmrežu na svoj način.

Upravitelj kontrolera u oblaku (engl. *Cloud Controller Manager*) komponenta je prisutna kod Kubernetes klastera integriranih s infrastrukturom u oblaku. Pokreće kontrolere koji kontroliraju infrastrukturu u oblaku za potrebe klastera.

U sam klaster mogu biti pušteni u rad razni dodaci koji proširuju njegove mogućnosti. Primjeri dodataka jesu DNS, grafičko sučelje za upravljanje klasterom te razna rješenja za nadzor i bilježenje zapisa.

4.13. Tipovi servisa

Kubernetes servisi omogućuju logičko grupiranje više kapsula te njihovo izlaganje prema ostatku mreže pod jednim imenom. Kube-proxy kreira virtualnu IP adresu za većinu vrsti servisa. Ako se koristi dodatak za DNS, što i jest najbolja praksa, svaki servis imat će svoj DNS zapis. Ukoliko kapsula želi dohvatiti IP adresu za servis izvan njezina imenskog prostora, mora koristiti ime servisa i ime imenskog prostora u kojemu se nalazi. Podaci o kreiranim servisima također se dodaju u varijable okoline kapsule. Osnovni Kubernetes servisi jesu:

- ClusterIP
- NodePort
- LoadBalancer.

ClusterIP tip servisa dostupan je samo unutar klastera. Prilikom kreiranja novih kapsula i brisanja starih njihove IP adrese mogu se promijeniti. Ukoliko se kreira više kapsula iste namjene, potrebno ih je nekako grupirati. Servis tipa ClusterIP omogućuje grupiranje kapsula s istim oznakama te kreira IP adresu iza koje se promet usmjerava prema kapsulama.

NodePort servis dostupan je unutar i izvan klastera na svim članovima klastera na zadanom portu. Servis je dostupan na virtualnoj IP adresi unutar klastera i na svakom članu klastera na istom portu. Zbog korištenja portova na članovima klastera moguće je koristiti samo jedan servis po portu. To može predstavljati dodatan izazov kod aplikacija koje žele koristiti standardne portove, uključujući i internetske poslužitelje.

LoadBalancer servis kreira servis koristeći raspoređivač prometa (engl. *Load Balancer*) pružatelja usluge u oblaku. Servis je dostupan internim i vanjskim korisnicima. Bez instalacije dodataka servisi tipa LoadBalancer dostupni su isključivo u okruženjima u oblaku koji ih podržavaju.

4.14. Segmenti korištenja

Kubernetes klaster podijeljen je u dva segmenta: aplikacijski i kontrolni.

Kontrolni segment klastera zadužen je za rad samog klastera, reagiranja na događaje u klasteru, odnosno kontinuirano dovođenje klastera u željeno stanje. Komponente koje spadaju u kontrolni segment jesu:

- API poslužitelj
- etcd
- raspoređivač
- upravitelj kontrolera
- kubernetes član klastera.

Aplikacijski segment klastera zadužen je za izvršavanje zadataka koje definira aplikacijski segment klastera. Aplikacija, odnosno kapsule koje ju čine i popratni servisi logički su definirani unutar kontrolnog segmenta, ali se izvršavaju unutar aplikacijskog. Komponente koje čine aplikacijski segment klastera jesu:

- kubelet
- kapsula
- sustav izvršnog okruženja kontejnera
- kube-proxy.

Autorizacija korisnika unutar klastera može biti pomoću atributa, rola ili povratnih zahtjeva (engl. *webhook*). Autorizacija putem atributa konfigurira se pomoću datoteke koja sadrži jedan ili više redaka, a svaki redak definira samo jedan JSON objekt prema specifikaciji. Autorizacija putem povratnih zahtjeva omogućuju Kubernetes sustavu da sve zahtjeve za autorizaciju šalje vanjskoj aplikaciji. Aplikacija mora poštovati format koji sustavi Kubernetes prilikom slanja odgovora. Autorizacija putem rola (engl. *role-based access control*), skraćeno RBAC, omogućuje definiranje različitih grupa prava, odnosno rola, te njihovo dodjeljivanje. Servisni računi omogućuju kapsulama pristup resursima unutar klastera pod određenim identitetom. Rola je grupa prava nad određenim resursima. Role mogu biti kreirane na razini imenskog prostora i razini klastera. Identiteti se povezuju s rolama pomoću posebnog resursa povezivanja role

(engl. *role binding*). Role se mogu povezati s resursima na razini imenskog prostora i razini klastera. Rola ne zabranjuje određena prava. Prava dobivena pomoću više rola zbrajaju se.

Dodjeljivanjem role definirane na razini imenskog prostora identitet dobiva prava role unutar tog imenskog prostora. Dodjeljivanje role na razini klastera može omogućiti određene radnje nad resursima kontrolnog segmenta klastera i određene radnje nad svim resursima definiranim u roli u svim imenskim prostorima.

4.15. Kubernetes resursi

Kubernetes API definira više vrsta resursa. Neke od često korištenih vrsta resursa u Kubernetes sustavima jesu:

- imenski prostor
- kapsula
- deployment
- upravljači dolaznog prometa (engl. Ingress Controller)
- mrežne politike.

Moguće je i kreiranje vlastitih resursa i proširivanje Kubernetes API-ja.

Imenski prostori pružaju mogućnost logičkog grupiranja različitih resursa u odvojenu cjelinu. Imena resursa moraju biti jedinstvena unutar imenskog prostora. Resursi mogu biti na razini klastera ili na razini imenskog prostora. Imenski prostor razdvaja samo resurse koji se kreiraju unutar pojedinog imenskog prostora. Pomoću kvota za resurse moguće je ograničiti ukupne resurse imenskog prostora.

Dolazni resurs (engl. *Ingress*) omogućuje vanjski pristup servisima unutar klastera, najčešće putem HTTP i HTTPS protokola. Može pružiti različite funkcionalnosti kao što su balansiranje, odnosno upravljanje prometom, zatvaranje SSL konekcija i virtualni hosting na temelju imena. Točna funkcionalnost dolaznog resursa ovisi o upravljaču dolaznog prometa (engl. *Ingress Controller*) koji omogućuje njegov rad. Upravljač

dolaznog prometa posebna je vrsta upravitelja prometa koji je namijenjen za rad s Kubernetes sustavima. Prima promet koji dolazi izvan klastera te ga usmjeruje određenim kapsulama ili servisima. Može upravljati i odlazećim prometom.

ReplicaSet tip objekta osigurava da određen broj kapsula uvijek bude pokrenut.

Deployment tip objekta služi za definiranje željenog stanja kapsula i ReplicaSet objekata. Ukoliko se definira drugačije stanje ReplicaSet objekta, odnosno promijeni broj željenih kapsula, kontroler deploymenta postepeno mijenja broj kapsula. Korištenjem Deployment objekta može se kontrolirati način puštanja aplikacije u rad. Kako bi kontroler Deploymenta mogao prepoznati koje kapsule spadaju pod koji Deployment, svaki resurs tipa Deployment definira selektor (engl. *selector*) koji se sastoji od jedne ili više oznaka i uvjeta koje te oznake moraju zadovoljiti. Zadatak kontrolera Deploymenta jest da stanje unutar klastera dovede u stanje definirano u samom Deploymentu.

Postoji više načina za određivanje na kojem će članu klastera kapsula biti pokrenuta. Specifikacijom nodeSelector atributa kapsula će biti pokrenuta samo na članovima klastera koji zadovoljavaju sve uvjete. Pomoću afiniteta resursa moguće je koristeći različite operatore definirati niz uvjeta vezanih uz oznake članova klastera ili postojećih kapsula na određenom članu klastera te tako odrediti gdje će se kapsula pokrenuti.

Mrlje (engl. *taint*) resursa pružaju funkciju obrnutu od afiniteta resursa. One omogućuju članu klastera da odbije pokrenuti određene kapsule. Određene kapsule mogu tolerirati mrlje te se tako svejedno pokrenuti na određenom članu klastera unatoč mrlji. Kontroler članova klastera automatski primijeni određene mrlje ukoliko se stanje člana klastera promijeni. Mrlje mogu spriječiti pokretanje novih kapsula ili izbaciti sve kapsule koje nemaju toleranciju s određenog člana klastera.

Mrežne politike implementirane su od strane dodataka za mrežna sučelja kontejnera. Mrežne politike ograničavaju kako kapsule mogu komunicirati s ostatkom mreže. Politike se mogu definirati u dva smjera: dolazni (engl. *ingress*) i odlazni (engl. *egress*). Smjer se gleda od same kapsule, znači da dolazna pravila definiraju promet koji može doći do kapsule, a odlazna koji promet kapsula može uputiti dalje.

Kubernetes je iznimno moćan alat za orkestraciju kontejnera. Sa svojim mnogim mogućnostima može predstavljati izazov razvojnim inženjerima i administratorima. Kubernetes svakako nije rješenje za sve probleme aplikacija i nije optimalno rješenje za sve slučajeve korištenja. No nove tehnologije također znače i nove sigurnosne ranjivosti te je zbog toga potrebno obratiti dodatnu pažnju na sigurnosne ranjivosti Kubernetes sustava.

PITANJA ZA PONAVLJANJE:

1. Kako Kubernetes proširuje mogućnosti Docker sustava?
2. Navedi vrste servisa u Kubernetes klasteru.
3. Kada je LoadBalancer servis prikladan za korištenje?
4. Što je rola?
5. Koji još mehanizmi raspoređivanja kapsula u klasteru postoje uz deployment?
6. Može li Kubernetes klaster raditi bez Dockera?
7. Koja se vrsta servisa koristi za komunikaciju između aplikacija unutar klastera?
8. Kako omogućiti kapsuli određena prava unutar samog klastera?
9. Navedi vrste Kubernetes resursa koje poznaješ.
10. Može li kapsula sadržavati više od jednog kontejnera?

5. Sustavi za distribuciju aplikativnih rješenja

U OVOM POGLAVLJU NAUČIT ĆETE:

- koje vrste sustava distribucije aplikacija postoje
- koje su vrste konfiguracijskih datoteka u distribuciji koda
- kako izgleda proces izrade aplikacije
- kako se izvršava testiranje koda
- kako se izvodi distribucija aplikacija
- koja je razlika između sustava za distribuciju mobilnih i web-aplikacija.

5.1. Vrste sustava distribucije

Korištenje sustava za distribuciju u procesu razvoju koda vrlo je važno zbog automatizacije tijeka razvoja softvera, čime smanjujemo pojavu pogrešaka te na kraju dobivamo kvalitetniji kôd. U tom procesu na svaki zahtjev izmjene koda radimo prevođenje, testiranje ispravnosti, pakiranje, isporuku koda te izvješćivanje. Sustav distribucije građen je od linija (engl. *pipelines*), pri čemu svaka linija ima određene etape unutar kojih se izvodi spomenuto kompiliranje, izgradnja te implementacija koda. Vrlo je često zahtjevno pronaći alat za distribuciju zato što moramo pronaći alat koji odgovara svim našim zahtjevima.

DevOps timovi danas teško mogu funkcionirati bez sustava za distribuciju. Vrlo često takvi sustavi postanu izrazito vrijedni za organizaciju kako se povećava složenost razvojnog procesa. Sustav za distribuciju u jednom trenutku mora podržavati više korisnika, biti pouzdan i pouzdano raditi unatoč velikom opterećenju. U velikim organizacijama dešava se da jedan sustav za distribuciju preraste jedan tim i krene ga koristiti cijela organizacija za podržavanje svake poslovne linije.

	SELF-HOSTED	MANAGED (SAAS)
Koordinacija među timovima	Zahtijeva koordinaciju među timovima za primjenu, ažuriranja, nadogradnje sustava i popravke.	Koordinacija je smanjena, lakša podjela *pipelina* i artefakata između timova.
Skalabilnost	Sami smo odgovorni za nadzor sustava, planiranje nadogradnje i ulaganja u poslužiteljske resurse.	Povećanje često označava dodavanje dodatnih korisnika ili prelazak na višu razinu usluge. Zahtijeva više troškova.
Ažuriranje i održavanje softvera	Sami smo odgovorni za praćenje i primjenu ažuriranja softvera.	Upravitelj usluga odgovaran je za ažuriranje softvera.
Popravljanje i održavanje	Sami smo odgovorni za održavanje sustava u radu i za njihovo popravljanje kada se pokvare.	Upravitelj usluga odgovoran je za održavanje sustava.

Tablica 5.1.1. Usporedba Self-hosted i Managed sustava za distribuciju

Kada pričamo o upravljanju našom vrstom sustava za distribucijom, pričamo o (engl. *self-hosted*) verziji. U ovoj vrsti distribucijskog sustava sami smo zaduženi za instalaciju i pokretanje sustava na vlastitom poslužitelju, za ažuriranja, popravke, monitoriranje i vrijeme rada. S druge strane, vanjski su sustavi zapravo aplikacije u oblaku, tj. SaaS varijante. U SaaS varijanti prepuštamo upravljanje sustava davatelju usluga te na taj način odustajemo od kontrole nad poslužiteljem i aplikacijom. Ponekad to možemo gledati kao prednost jer se ne moramo brinuti o svakom detalju S druge strane, to znači i da se oslanjamo na nekog drugog da obavi posao za nas.

Danas većina sustava za distribuciju nudi obje mogućnosti: *self-hosted* i upravljane sustave. Kada pričamo o sustavim za distribuciju, jedan od najpoznatijih alata na

tržištu jest Jenkins. Jenkins, iako je najrašireniji alat, ne nudi *managed* varijantu, već su DevOps inženjeri zaduženi za postavljanje *self-hosted* verzije. Zbog spomenutog problema konkurenti su uvidjeli priliku napraviti sličan alat, uz mogućnost korištenja *managed* verzije. Među poznatijim *managed* alatima na tržištu jesu: Gitlab CI, Travis Ci, GitHub Actions.

5.2. Alat Jenkins

Jenkins, izvorno Hudson, najpoznatiji je sustav za distribuciju. Njegov primarni zadatak očitati je promjene u udaljenom repozitoriju i pokrenuti proces testiranja softvera i izgradnje aplikacije. U slučaju pogreške sustav će nas obavijestiti da bismo što brže ispravili pogrešku. Osim toga, sustav će proizvesti izvješća o kvaliteti koda, kao što je analiza izvršenih testova, statistika o vremenu izgradnje, učestalost pogrešaka i slično. Jenkins je postao iznimno popularan zbog jednostavnosti korištenja, pristupačnog i jednostavnog korisničkog sučelja i, ono najbitnije, zbog jednostavne mogućnosti proširenja pomoću dodataka (engl. *plugins*). Alat kao što je Jenkins nudi izrazito visoku skalabilnost i tvrtke ga često uvode postupno u rad.

Tipični hodogram uvođenja alata Jenkins prati sljedeći hodogram:

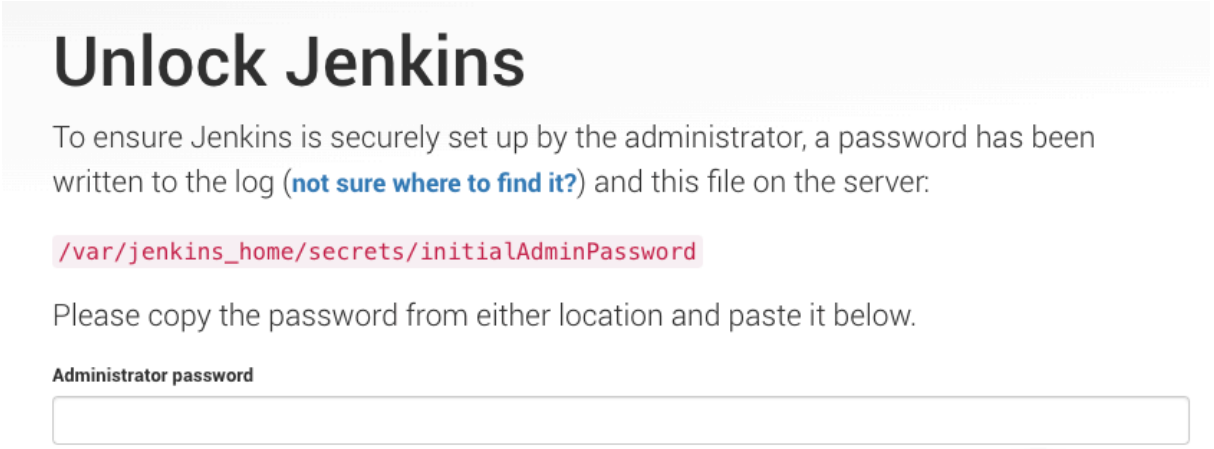
- U prvom koraku potrebno je uvesti sustav koji će na dnevnoj bazi izgrađivati aplikaciju s udaljenog repozitorija.
- Zatim u drugoj fazi želimo uvesti automatska testiranja prije izgrađivanja aplikacije, također na dnevnoj razini.
- Sljedeća je faza uvođenje metrika za kvalitetu koda te automatska izgradnja dokumentacije koda.
- Četvrta je faza dodatno testiranje aplikacije koje se odrađuje tek nakon što je aplikacija izgrađena.
- Posljednja je faza kontinuirano objavljivanje aplikacije krajnjim korisnicima.

5.3. Instalacija Jenkinsa

Podesit ćemo Jenkins pomoću Docker kontejnera, što označava da radimo pomoću *self-hosted* verzije. Za pokretanje koristimo naredbu sličnu sljedećem primjeru:

```
docker container run -d \
    -p [LOKALNI_PORT]:8080 \
    -v [PUTANJA_ZA_POHRANU]:/var/jenkins_home \
    --name jenkins-local jenkins/jenkins:lts
```

Nakon pokretanja naredbe pokrenut će se *docker* kontejner unutar kojeg će se nalaziti aplikacija. Naš sljedeći korak otići je na URL putanju: **`http://localhost:[LOKALNI_PORT]`**, gdje ćemo morati postaviti početne postavke kako bi Jenkins funkcionirao.

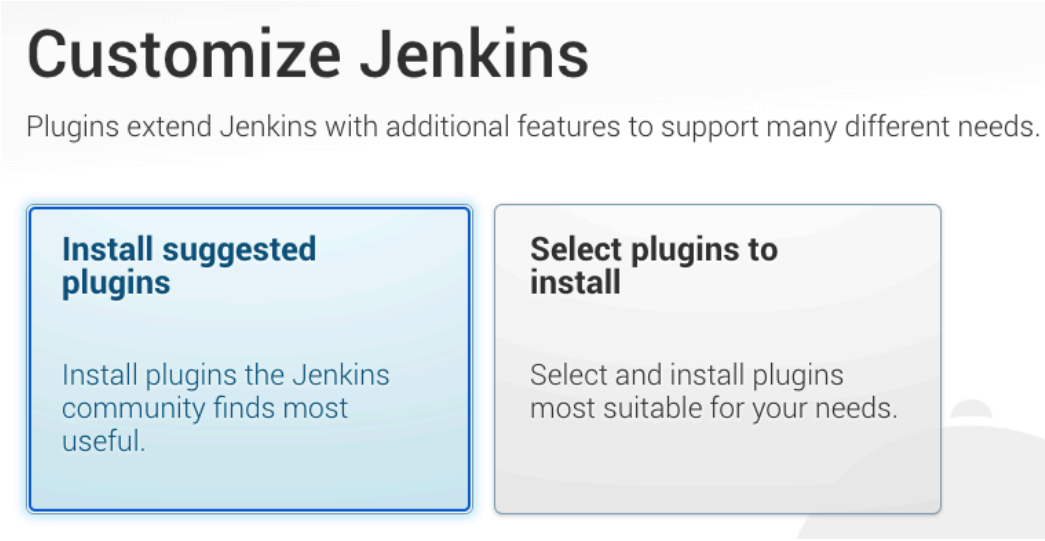


Slika 5.3.1. Postavljanje Jenkinsa

Prema slici 5.3.1. morat ćemo potražiti lozinku koju je generirao sam Jenkins. Kako bismo to odradili, moramo otići u naredbeni redak i pokrenuti naredbu:

```
docker container exec jenkins-local \
sh -c "cat /var/jenkins_home/secrets/initialAdminPassword"
```

Nakon pokretanje naredbe u naredbenom retku ispisat će se znakovni niz koji predstavlja lozinku administratora, nakon toga kopiramo tu lozinku i zalijepimo je u tekstualni okvir: Administrator password.



Slika 5.3.2. Instalacija dodataka za Jenkins

U sljedećem koraku Jenkins će tražiti da podesimo dodatke. Odabrat ćemo „Install suggested plugins". Nisu nam potrebni posebni dodaci prilikom prve instalacije. Nakon toga pokazat će se ekran gdje pratimo napredak instalacije Jenkinsa. To može potrajati i do nekoliko minuta. Nakon uspješne instalacije sustav će tražiti da napravimo račun administratora sustava, što znači da bi te informacije trebali unaprijed pripremiti.

5.4. Jenkins CI/CD

Kontinuirana integracija (engl. *CI*) praksa je u razvoju programskih rješenja koja pomaže programerima u pravovremenom testiranju i ranom uklanjanju pogrešaka nakon što su napravili izmjene u kodu. Čest primjer iz prakse jest taj da razvojni inženjeri razvijaju novu funkcionalnost u odvojenoj grani, pa bi prije nego što spoje odvojenu granu s glavnom htjeli saznati kako će se izmjene ponašati na stvarnom sustavu. Nakon svake promjene postavljene na udaljeni repozitorij na distribuiranom sustavu, u našem slučaju Jenkins, pokreće se izgradnja aplikacije (engl. *Build*). Izgradnja aplikacije u svojem procesu uzima programsko rješenje s udaljenog repozitorija i postavlja okolinu za generiranje artefakta. Ako rezultati testa nisu uspješni, Jenkins će obavijestiti programera o tome te on može napraviti potrebne izmjene kako bi njegov kôd uspješno prošao testiranje. Cijeli opisani proces daje mogućnost programerima da više vremena posvete kôdu, a manje izgradnji aplikacije i testiranju kôda. Pomoću ovakvog

pristupa također smo pristupačniji klijentu, s obzirom na to da je dostupnost posljednje verzije programskog rješenja uvijek dostupna. CD (engl. *Continuous Delivery*), odnosno kontinuirana isporuka aplikativnog rješenja koja je proizašla iz kontinuirane integracije proces je čiji je cilj na pojednostavljen način isporučiti klijentu rješenje uz što veću pouzdanost i uštedu vremena. U praksi želimo osigurati da krajnjem korisniku isporučujemo posljednje izmjene proizvoda u predefiniranom razdoblju bez intervencije razvojnih inženjera. *Continuous integration* (CI) / *Continuous delivery* (CD) linija predstavlja seriju koraka koji se moraju izvesti kako bi se programsko rješenje isporučilo krajnjem korisniku. U procesu izvedbe CI/CD-a Jenkis se sastoji od izrazito bitnih komponenata:

- poslovi (engl. *jobs*)
- izgradnja aplikacije (engl. *builds*)
- parametri
- linije (engl. *pipelines*).

Jenkins posao skup je pravila prema kojemu se izgrađuje aplikacija. Sastoji se od naziva posla, opisa, definiranih parametara za pokretanje izgradnje aplikacije, instrukcija za izgradnju aplikacije te liste akcija nakon što je posao završen.

5.5. Jenkins komponenta posao

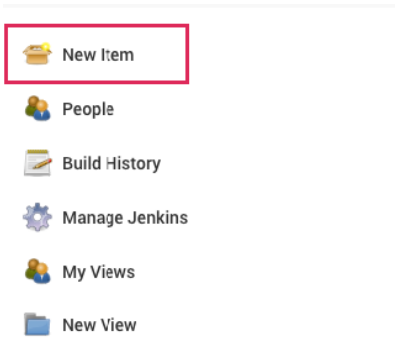
Izgradnju Jenkins posla, odnosno linije, možemo izvesti na dva različita načina. Linija se može izgraditi programiranjem, ali i uz pomoć grafičkog sučelja. Češća je praksa programiranje tako da stvorimo u udaljenom repozitoriju Jenkinsfile. Jenkinsfile je tekstualna datoteka koja sadrži definicije Jenkinsove linije i piše se u sintaksi Groovy programskog jezika. Uobičajena praksa jest ta da se Jenkinsfile nalazi u istoj mapi gdje i izvorni kôd. Svaka linija mora imati minimalno jednu fazu izvršavanja i potrebno je definirati poslužitelj na kojem se izvršava posao. Jenkins etapa jedinica je posla i može biti jednostavna Linux Bash ili Windows batch naredba, a preko dodataka mogu se izvršavati kompleksne stvari poput: Python skripta, Groovy, preuzimanje Android dodataka itd.

```
Linija {
   agent   any
   stages  {
     stage('Izgradnja')    {
        steps   {
           echo 'Gradim..'
        }
     }
   }
}
```

Najbitniji dio Jenkinsova posla jesu parametri prema kojima upravljamo tokom izvršavanja linije. Jenkins podržava velik broj tipova parametara, kao što su znakovni nizovi, brojevi, izbornici i slično. Parametri se predaju automatski ili ručno. Primjer automatskog predavanja parametara jest kada jedan Jenkins posao pokreće drugi te se predaju informacije onog prethodnog. Primjer ručnog predavanja parametara jest kada korisnik odabere verziju koju želi pokrenuti. Primjer nadogradnje našeg kôda:

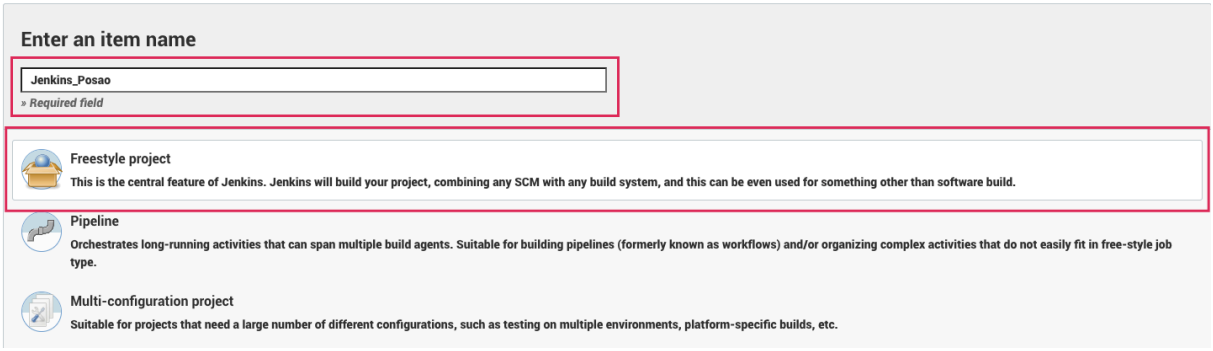
```
Linija {
   agent   any
   parameters   {
     string(
  name: "BUILD_VERZIJA",
  trim: true,
  description: "Ovdje upisujemo verziju aplikacije koju gradimo")
   }
   stages  {
     stage('Izgradnja')    {
        steps   {
           echo 'Gradim..'
        }
     }
   }
}
```

Sada kada imamo spreman *Jenkinsfile* želimo napraviti prvi *Jenkins posao*.



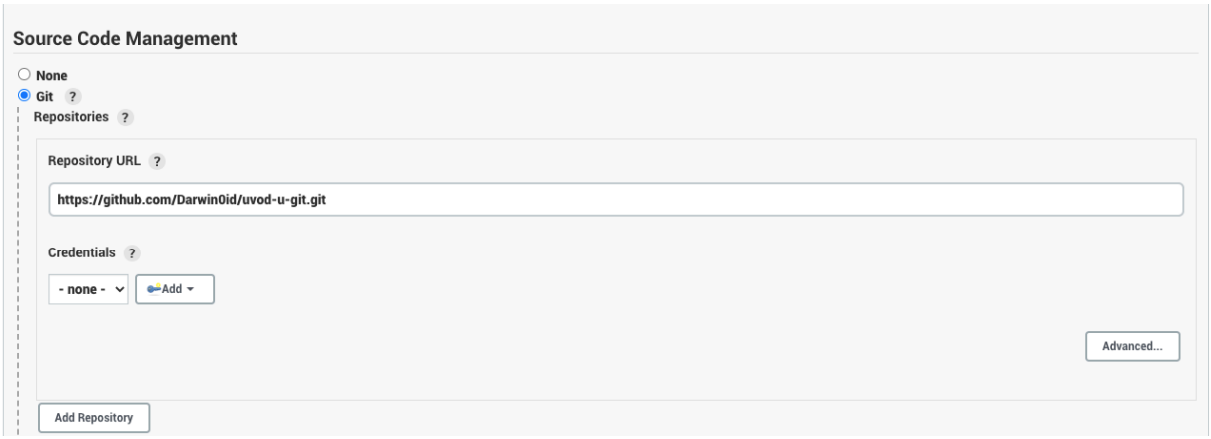
lika 5.5.1. Stvaranje prvog Jenkins posla

S lijeve strane u izborniku moramo odabrati opciju „New Item”. Nakon toga u sljedećem koraku moramo obavezno definirati naziv posla i odabrati tip projekta.



Slika 5.5.2. Definiranje naziva posla i odabir vrste posla

U sljedećem koraku odabiremo s koje lokacije uzimamo kôd. Mi ćemo koristiti udaljeni repozitoriju na GitHubu.



Slika 5.5.3. Lokacija Jenkinsfilea

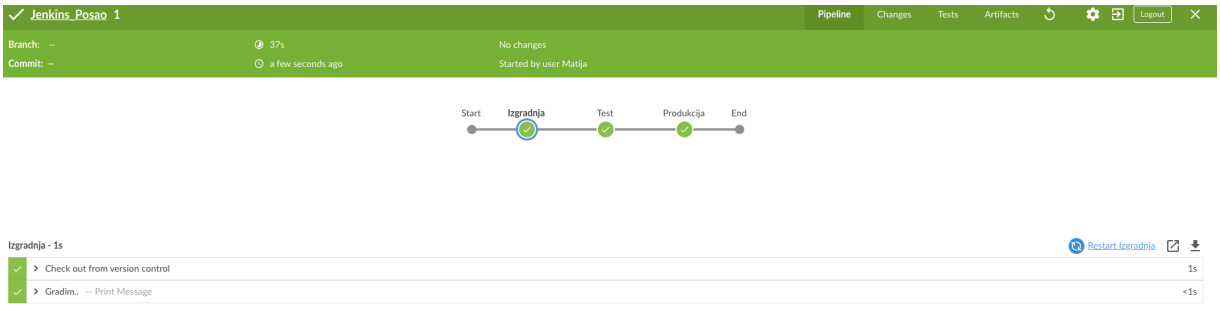
Nakon potvrde repozitorija možemo pritisnuti gumb „Build”.

Slika 5.5.4. Prikaza parametriziranog Jenkins posla

5.6. Sekvencijalni i paralelni Jenkins poslovi

Etapa u Jenkinsu predstavlja skup zadataka kroz koje Jenkins posao mora proći da bi dovršio liniju. Etapa može sadržavati jedan ili više koraka. Svaki korak može izvršavati definirane zadatke ili može pokrenuti i drugi Jenkins posao.

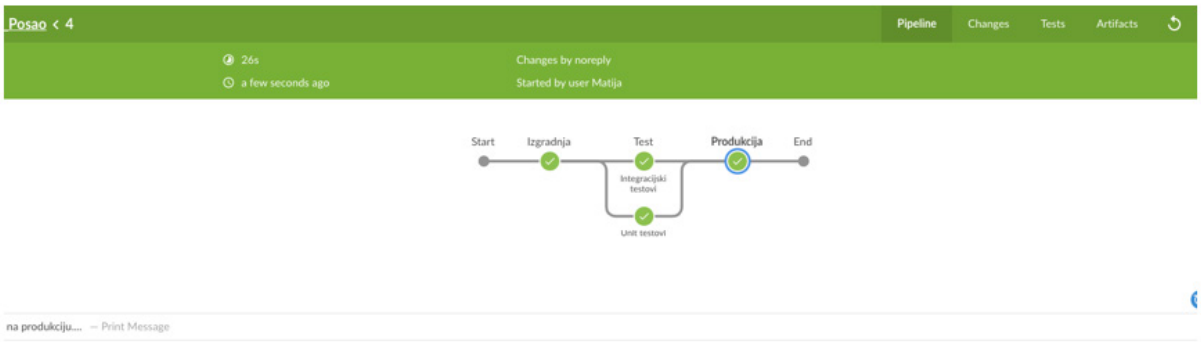
Sekvencijalna linija predstavlja izvršavanje etapu po etapu, baš kao što smo naveli u prethodnom primjeru. Nakon izvršavanja takvog posla dobit ćemo prikaz kao na slici 5.6.1.



Slika 5.6.1. Prikaz sekvencijalnog izvršavanja zadataka

Paralelne linije vrlo često dolaze do izražaja u etapama testiranja. To je trenutak kada želimo pokrenuti različite etape istovremeno. Za takvu funkcionalnost morat ćemo koristiti blok s nazivom parallel.

```
Linija  {
   stages  {
     stage('Izgradnja')    {
        steps    {    echo  'Gradim..' }
     }
     stage('Test')    {
        steps    {
            parallel  (
                "Unit testovi":{echo   'Testiram..'},
                "Integracijski   testovi":  {echo 'Testiram..'}
            )
         }
    }
}
```



Slika 5.6.2. Prikaz paralelnog izvršavanja zadataka

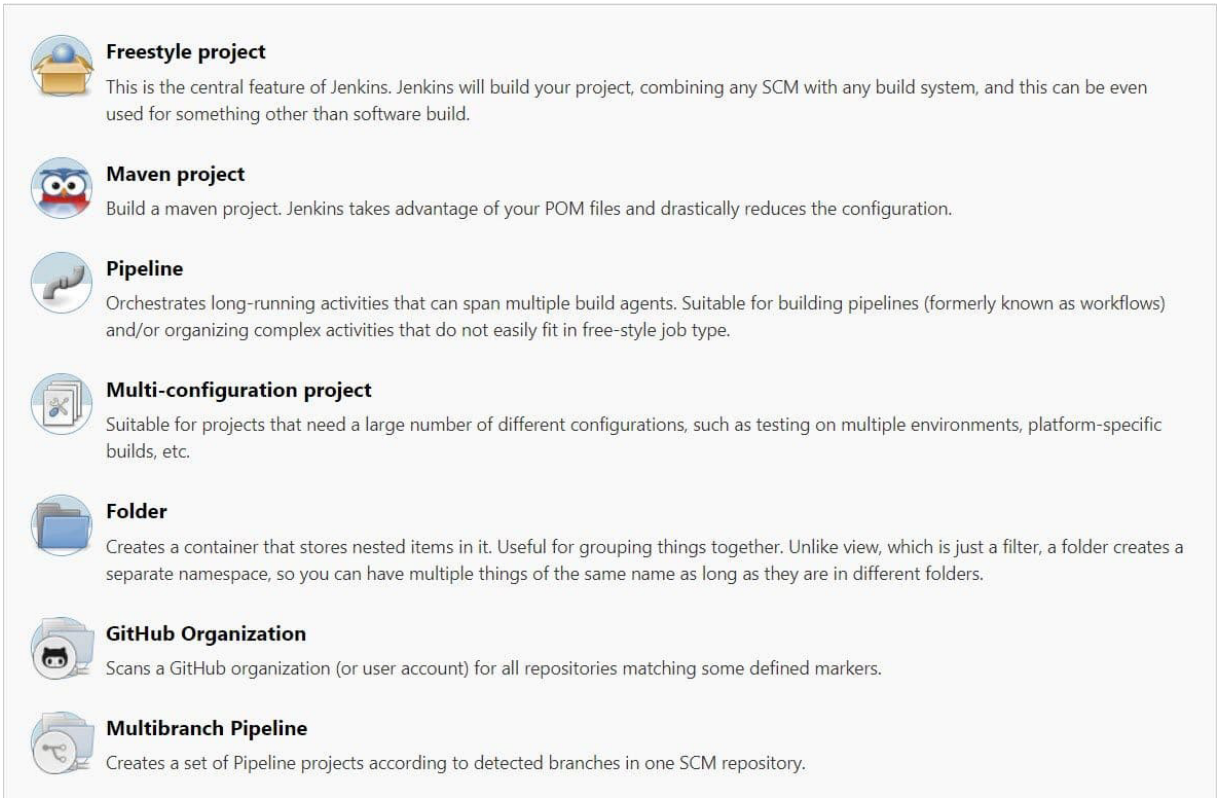
Paralelne linije zahtijevaju navođenje etapa unutar kôda koje će se izvršiti, dok etape razdvajamo zarezom. Na ovaj način možemo ubrzati dugačke linije koje u izvršavanju mogu potrajati izrazito dugo. *Parallel* također ima vlastitu težinu u trenutku kada imamo jako puno Jenkins poslova. Ako zaključimo da se određeni poslovi mogu izvršavati istovremeno jer pripremaju podatke za glavnu liniju, tada bismo u bloku *Parallel* naveli ostale poslove.

PITANJA ZA PONAVLJANJE:

1. Za što se koriste sustavi za distribuciju?
2. Nabrojite vrste sustava za distribuciju.
3. Koja je razlika između sustava za distribuciju koja se nalazi u oblaku i *self-hosted* verzije?
4. Što je Jenkins?
5. Što je proces izgradnje aplikacije?
6. Što označava CI? Što označava CD?
7. Osnovne komponente CI-a/CD-a.
8. Objasnite razliku između sekvencijalnog i paralelnog posla.

5.7. Konfiguracija sustava distribucije

Prije izgradnje Jenkins posla moramo biti svjesni zahtjeva za izgradnju naše aplikacije. Potrebno je razmisliti na kojem operativnom sustavu trebamo izvršavati posao, koji programski jezik koristi naš razvojni tim, koje sve biblioteke trebamo za uspješnu izgradnju, trebamo li to raditi na specifičnoj infrastrukturi.



Slika 5.7.1. Prikaz vrsta Jenkinsovih poslova

Poslovi su središnja točka Jenkinsova procesa izgradnje. Posao se može smatrati posebnim zadatkom za postizanje traženog cilja u Jenkinsu. Spomenuti razlozi prvo zahtijevaju od nas odabrati prikladnu vrstu Jenkins posla.

Vrsta posla	Opis
FreeStyle Project	Freestyle Project u Jenkinsu improvizirani je ili neograničeni posao za izgradnju aplikacije ili zadataka s više operacija. Operacije mogu biti građenje, izvođenje linija ili bilo koje pokretanje skripte.
Maven Project	Maven Project koristi se za upravljanje i izgradnju projekata koji sadrže POM datoteke (Java aplikacije). Jenkins će automatski odabrati POM datoteku, napraviti konfiguracije i pokrenuti izgradnju aplikacije.
Linija	Linija Project izrazito je koristan za dugotrajne aktivnosti koje sadrže različite agente, odnosno poslužitelje za izgradnju aplikacija. Pogodan je onda kada nije dovoljan FreeStyle Project te kada su nam potrebni različiti poslužitelji za različite etape.
Multi-configuration Project	Ova je opcija prikladna u onim uvjetima kada su potrebne različite konfiguracije, poput testiranja na različitim okruženjima, posebne konfiguracije za specifične platforme.
GitHub Organization	Ova opcija skenira GitHub račun i pretražuje udaljene repozitorije koji odgovaraju određenim definiranim oznakama.

Tablica 5.7.1. Prikaz vrsta Jenkinsovih poslova

U trenutku kada odaberemo vrstu posla koja nam je potrebna, potrebno je razmišljati o etapi pripreme. U etapi pripreme želimo pripremiti okruženje ovisno o potrebama. **Uobičajeno je razmišljanje opisano sljedećim hodogramom:**

1. Želimo dohvatiti kôd iz udaljenog repozitorija.
2. Želimo podesiti virtualno okruženje unutar kojeg ćemo pokretati izgradnju aplikacije kako bismo osigurali okruženje bez utjecaja prethodno instaliranih aplikacija.
3. Želimo instalirati u virtualno okruženje sve ovisnosti biblioteka za pokretanje aplikacijskog rješenja.
4. Potrebno je odrediti izlaznu datoteku procesa izgradnje aplikacije.
5. Potrebno je pripremiti testove koji će testirati izgradnju aplikacije.
6. Potrebno je odrediti lokaciju gdje ćemo to pohraniti.

Za definiranja događaja unutar etape linija možemo koristit deklarativnu ili skriptnu

sintaksu; obje vrste temelje se na jeziku Groovy (DSL) ili pak možemo koristiti YAML.

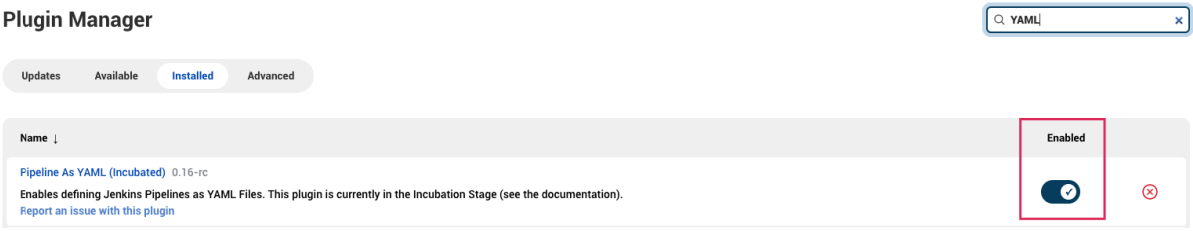
Deklarativna sintaksa izrazito je pojednostavljena i nepromjenjiva, a radimo je kroz korisničko sučelje Jenkinsa. Kada kreiramo i konfiguriramo deklarativne linije, tada zapravo ocrtavamo, odnosno definiramo alate i naredbe koje želimo da Jenkins izvrši. Takav pristup konfiguraciji pruža određenu razinu jednostavnosti u kreiranju i konfiguriranju linija, ali nedostaje mu fleksibilnost jer nemamo potpunu kontrolu nad sadržajem skripte i parametara.

Skriptna sintaksa svodi se na pisanje kôda kojim upravljamo etapama Jenkins posla. Jenkinsfile konfiguraciju na ovakav način koristimo kada nam je potrebno više koraka, ali moramo poznavati programski jezik Groovy. Ovakav pristup omogućuje nam pisanje Jenkins poslova visoke složenosti jer možemo koristiti uvjete, petlje, parametre i slične elemente programskog jezika. Ručno pisanje Jenkinsfilea također nam omogućuje stvaranje definicija varijabli, prilagođavanje vanjskih biblioteka i lakše generiranje izvještaja izvršavanja poslova.

5.8. Kako konfigurirati Jenkins posao pomoću YAML-a?

YAML je serijalizacijski standard za sve programske jezike. Baziran je na JSON-u i pruža sve njegove mogućnosti, a dodatno je nadograđen i novima. Najveća prednost YAML-a jest ta što podržava relacijska stabla, čime se može povećati čitljivost, kompaktnost i jasnoća sa smanjenom mogućnošću pogrešaka. YAML je danas postao standard u konfiguriranju distribucijskih sustava. Svi trenutno postojeći sustavi za distribuciju koriste YAML format za kreiranje linija. Ako se odlučite koristiti YAML, tada ćete pristupiti stvaranju posla kao i u prethodnoj sekciji, samo što nećete trebati znati Groovy već YAML.

Da bismo koristili YAML za konfiguraciju, morat ćemo omogućiti takvu postavku kroz *plugin* manager Jenkinsa.



Slika 5.8.1. Omogućivanje YAML-a unutar Jenkinsa

Nakon omogućivanja postavke kao što je prikazano na slici 5.8.1. naš je sljedeći korak napraviti Jenkins.yaml datoteku unutar koje ćemo zapisati etape i događaje.

```
linija:
  agent:
    any:
 stages:
    -stage:"Izgradnja"
      steps:
      - echo "Gradim.."
```

Kao što možemo primijetiti, osnove sintakse YAML zbirke su blokova i mapiranja koja sadrže parove ključ/vrijednost. Svaka stavka u zbirci započinje znakom **„-”**.

5.9. Izgradnja aplikacije iz grana i tagova

U radu s Jenkinsom, osim što može izgraditi aplikaciju iz grana repozitorija, mogu se koristiti i oznake. Biranje izgradnje aplikacije iz grane ili taga izrazito je korisno zato što možemo testirati aplikaciju prije nego što spojimo našu granu s glavnom granom. Unutar izgradnje pokrećemo testove koji će nam dodatno pomoći u shvaćanju jesmo li spremni za spajanje grana. Kako bismo podesili želimo li izgraditi aplikaciju iz određene grane ili taga, potrebno je konfigurirati Jenkins posao.

U odjeljku gdje podešavamo linije odabiremo da izvorni kôd uzimamo iz Git repozitorija. Osim adrese do repozitorija možemo birati iz koje grane želimo napraviti izgradnju aplikacije.

Slika 5.9.1. Odabir grana

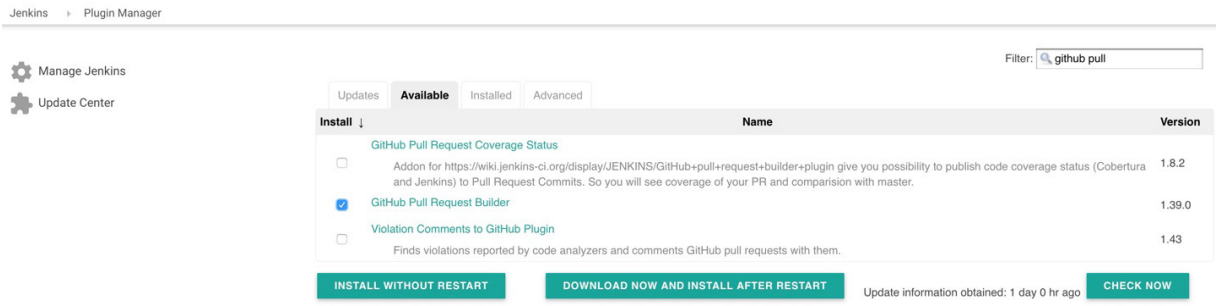
U tekstualno polje možemo navesti sve grane iz kojih želimo izgraditi aplikacije. Ako to polje ostavimo prazno, izgradit će se sve grane. Unutar tekstualnog polja na slici 5.9.1. možemo primijetiti unos **`*/main`**. Upisana vrijednost zadana je vrijednost za polje, no ovdje možemo upisivati različite vrijednosti. Najsigurniji je način korištenje sintakse **`refs/heads/<naziv_grane>`**. Na ovaj način očekivana grana za izgradnju nije dvosmislena. Osim grana možemo specificirati i tagove. Tagovi su nam izrazito korisni za testiranje i uspoređivanje sadašnje razvojne verzije s objavljenom verzijom. Za izgradnju aplikacije iz grane potrebno je koristiti oznaku: **`refs/tags/v1.0`**.

Slika 5.9.2. Odabir taga

5.10. Izgradnja aplikacije ovisne o događajima

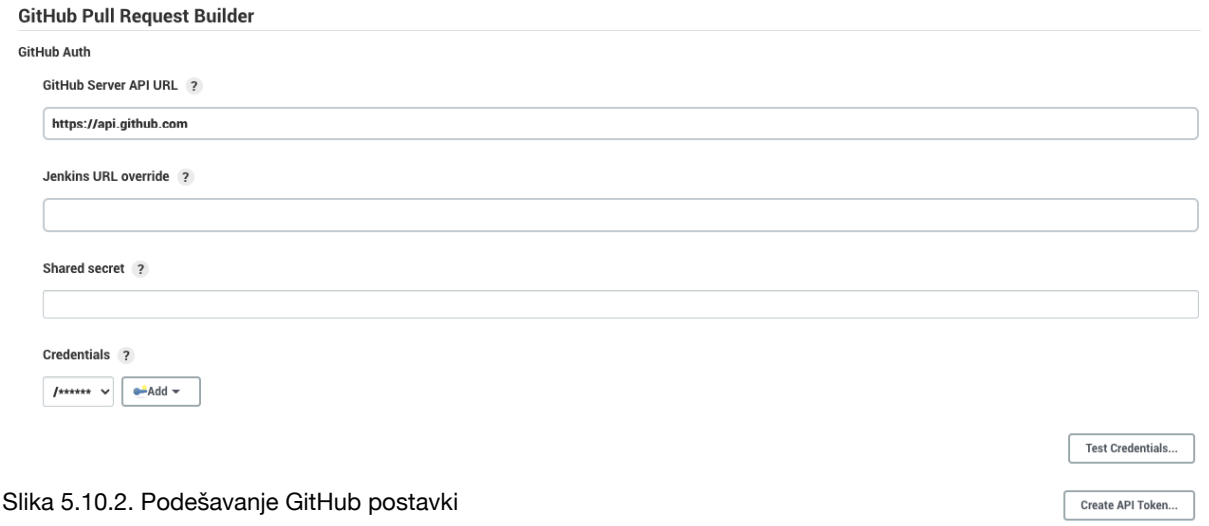
Izgradnja projekata na temelju zahtjeva za povlačenjem ili na objavi aplikacije nešto je što se ne može izbjeći u CI/CD linijama. Danas svaki tim radi nekoliko implementacija/operacija dnevno i u tom procesu mora se dogoditi puno izgradnji aplikacije. Također, timovi koji rade na istoj grani za suradnju zahtijevaju bržu integraciju kôda. Stoga je bolje imati automatizirani proces izrade koji pokreće CI/CD linija na zahtjev umjesto ručnog pokretanja poslova. Dosad smo vidjeli kako izgleda proces ručne izgradnje.

U prvom koraku prema postizanju spomenute fleksibilnosti morat ćemo instalirati dodatak unutar Jenkinsa. U odjeljku za upravljanje dodacima potrebno je instalirati dodatak s nazivom: GitHub Pull Request Builder.



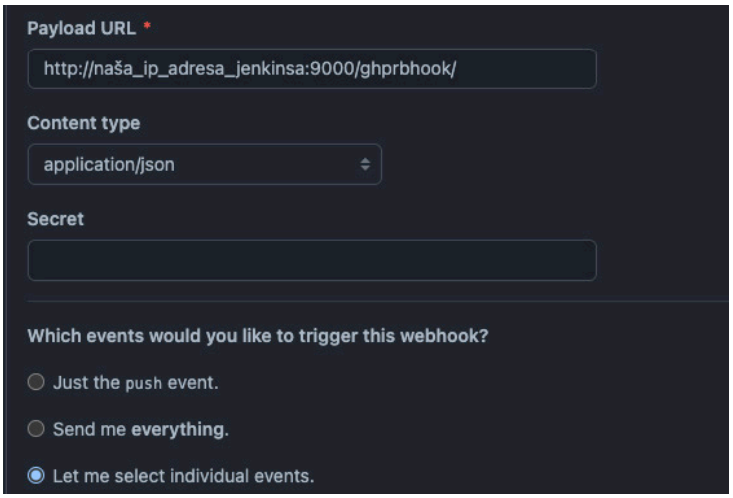
Slika 5.10.1. Instalacija dodatka: GitHub Pull Request Builder

Nakon instalacije zadatak nam je podesiti dodatak. Odlazimo u postavke Jenkinsa i tražimo gumb „System configuration”. U sekcijama tražimo naslov: „GitHub Pull Request Builder”. Sustav će tražiti da napravimo integraciju s našim GitHub računom.



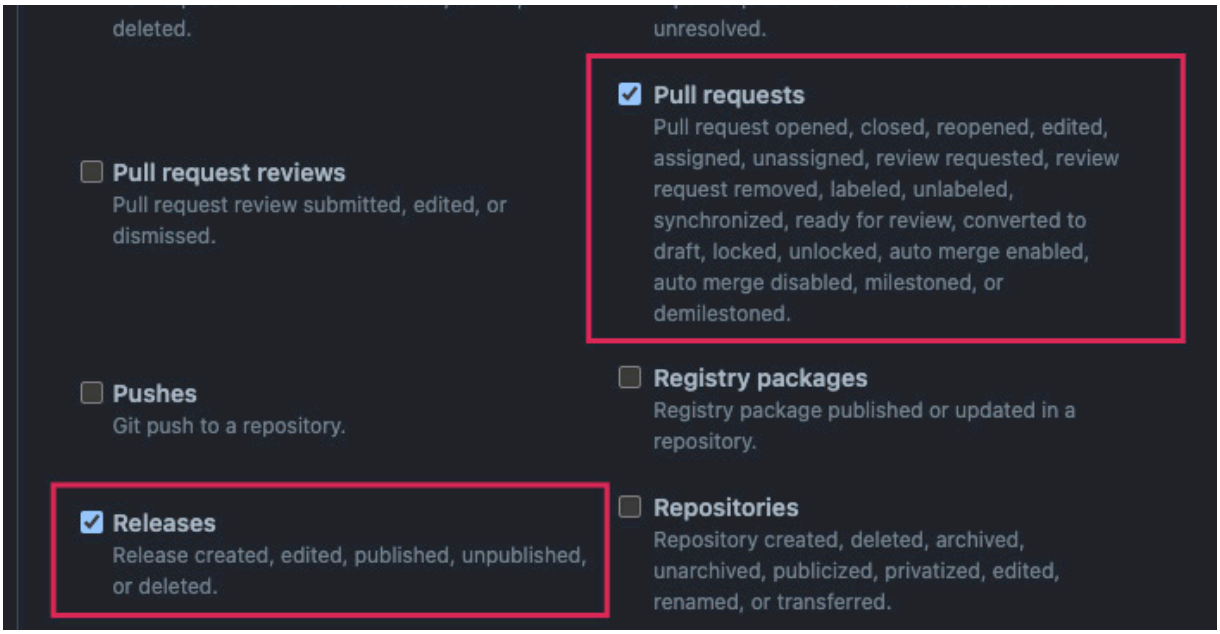
Slika 5.10.2. Podešavanje GitHub postavki

Nakon podešavanja Jenkinsa preostaje nam podesiti GitHub Povratni zahtjev (engl. *Webhook*).Povratni zahtjev u razvoju *web*-aplikacija metoda je za mijenjanje ponašanja *web*-aplikacije s prilagođenim povratnim pozivima. Povratne pozive mogu održavati, mijenjati i njima upravljati korisnici i programeri trećih strana koji ne moraju nužno biti povezani s izvornom web-aplikacijom. U našem slučaju GitHub će napraviti poziv prema našem Jenkins poslu i dobit će informaciju stanja izvršavanja Jenkins posla.



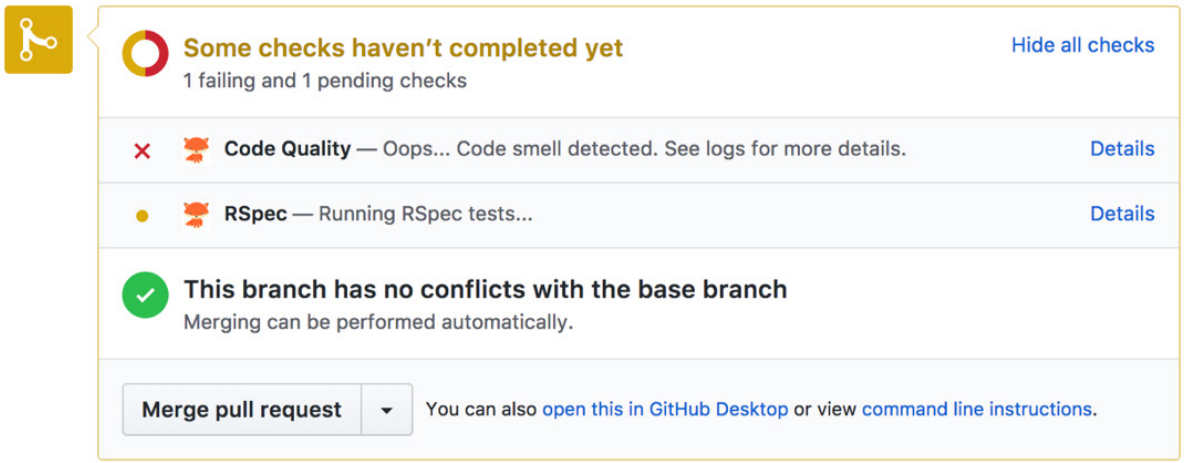
Slika 5.10.3. Podešavanje WebHooka

Nakon postavljanja IP adrese ili DNS-a za naš Jenkins server potrebno je odabrati na koje će se događaje izvršavati Jenkins posao.



Slika 5.10.4. Odabir na koje će se događaje izvršavati Jenkins posao

Nakon podešavanja WebHooka naš je zadatak podesiti Jenkins posao. Vrsta Jenkins posla bit će GitHub project, unutar kojeg ćemo morati upisati adresu našeg GitHub projekta. U trenutku kad se desi specificiran događaj na WebHooku, GitHub pokreće naš Jenkins posao. Na korisničkom sučelju GitHuba vidjet ćemo dodatak. Unutar njega možemo vidjeti status izvršavanja Jenkins posla. U ovom trenutku imamo prednost, razvojni inženjer ne mora imati pristup Jenkins poslu kako bi vidio proces izvršavanja.



Slika 5.10.5. Prikaz izvršavanja Jenkins posla u GitHubu

5.11. Koraci u sustavima distribucije

U ovom poglavlju opisat ćemo korake koji su predefinirani da bismo zadovoljili smisao linije. Da se podsjetimo, linija je zapravo CI/CD, što kombinira integraciju, isporuku i implementaciju četiri glavne etape:

- izvor (engl. *source*)
- izgradnja (engl. *build*)
- testiranje (engl. *test*)
- implementacija (engl. *build*).

Svaka etapa definirana je određenim standardima kako bismo mogli izgraditi i isporučiti traženo, baš kao i fizička proizvodnja u određenoj tvornici. U tvornicama moramo koristiti prilagođene strojeve kako bismo optimizirali i ubrzali proces izrade proizvoda. Isto tako, u softverskim linijama često prilagođavamo linije razvojnim inženjerima i njihovim potrebama.

Prva etapa linije preuzimanje je izvornog kôda, no da bismo došli do izvornog kôda, potrebno je definirati spremište kôda. Razvojni inženjeri u svojem razvoju koriste integrirano razvojno okruženje (IDE). Integrirano razvojno okruženje odlično je zbog značajki kao što su provjera kôda za rano otkrivanje pogrešaka, skeniranje ranjivosti, utvrđivanja standarda kôdiranja te objava kôda na server. Takav pristup želimo izbjegavati u projektima gdje više timova radi na značajkama aplikacije jer je teško odrediti u kojem smo trenutku spremni za objavu kôda na poslužitelj. Često se dešava u pristupu objave kôda da pokušavamo spojiti različite kôdove koje su nam razvojni inženjeri poslali i tu se dešavaju pogreške. Iz tog razloga ključno je da za sustave distribucije imamo definirano spremište kôda kao što je Git. Na taj način možemo definirati liniju po granama i izvršiti faze testiranja koje će ovisno o statusu objaviti kôd na poslužitelj. Ono što trebamo imati na umu jest da bismo često za svaku posebnu granu mogli imati drugačiju liniju.

Proces izgradnje uzima izvorni kôd iz spremišta, uspostavlja veze s knjižnicama kôda, različitim modulima i ovisnostima te se naposljetku radi izgradnja svih komponenti u izvršne datoteke. Alati koji se koriste u ovoj fazi također generiraju zapisnike procesa, označavaju pogreške koje treba istražiti i ispraviti te obavještavaju programere da je izgradnja dovršena. Biblioteke i alati koji se koriste prilikom izgradnje također generiraju zapisnike procesa, označavaju pogreške koje treba istražiti i ispraviti te obavještavaju o statusu izgradnje. Kao i kod stvaranja kôda, alati za izgradnju ovise o odabranom programskom jeziku. Vrlo se često ova etapa u liniji prilagođava onoj koju razvojni inženjeri koriste u trenutku lokalnog testiranja kôda. Stoga DevOps inženjer treba prikupiti zahtjeve za prevođenje kôda kako bi iste mogao implementirati u ovu fazu. Ono što moramo biti svjesni jest da ova etapa vrlo često treba implementaciju okruženja. Vrlo često želimo da se naš kôd izgrađuje na okruženju sličnom produkcijskom, a često dolazi do implementacije virtualnih računala ili Docker kontejnera unutar kojih se dešava izgradnja aplikacije.

Nakon što se izvršila izgradnja aplikacije, cilj nam je unutar linija podesiti okruženje na kojem ćemo izvršiti testiranje aplikacije. U ovom procesu pričamo o dinamičkom testiranju gdje će se svi testovi izvršiti automatski na temelju čega trebamo dobiti određeni izvještaj. Proces testiranja započinje pokretanjem testova koje su napisali programeri. Unutar tog testiranja radimo osnovna testiranja značajki; rade li one ispravno. Nakon toga potrebno je provesti regresijsko testiranje. Regresijsko testiranje uobičajeno

pišu inženjeri osiguranja kvalitete. Regresijsko testiranje proces je testiranja u kojem osiguravamo da nove promjene, odnosno značajke, nisu narušile rad prethodnih funkcionalnosti. Kod zahtjevnijih linija često se implementira dodatan niz testova kao što su integracijski, performanse i preglednički testovi. Ako se tijekom testiranja pojave pogreške, rezultati se vraćaju programerima radi analize i ispravaka u sljedećim verzijama. Tradicionalan pristup gdje su inženjeri osiguranja kvalitete testirali ručno sve korake bio je izrazito spor proces te podložan pogreškama jer su ta testiranja često bila subjektivna, odnosno testiranja su bila odrađena na način kako bi sam stručnjak koristio aplikaciju.

Posljednja etapa linije tipično je implementacijska etapa. Ako je etapa testiranja prošla bez pogrešaka, smatramo da je izvršna datoteka kandidat za implementaciju odnosno *release candidate* (RC). U posljednjem koraku odabiremo šalje li se izvršna datoteka ljudskim dionicima i trebaju li oni odobriti hoće li se desiti objava. U slučaju takvog pristupa takvu bismo liniju nazvali linija kontinuirane isporuke. Ako u etapi implementacije šaljemo kôd na određeno okruženje i na tom okruženju podignemo aplikaciju, tada takvu liniju nazivamo linijom kontinuirane isporuke. Ono što moramo imati na umu jest to da u liniji kontinuirane isporuke ne želimo odmah nakon izgradnje kôd postaviti na produkcijsko okruženje. Iako je sve uspješno pripremljeno, potrebne su dodate mjere opreza i korisničko testiranje kako bismo utvrdili funkcionira li stvarno aplikacija kako treba.

Do sada smo opisivali predefinirane korake odnosno korake koje bi svaka linija trebala imati. Često se u procesu izrade linija javljaju potrebe za implementacijom korisničkih odnosno specifičnih koraka. Kada pričamo o korisničkim koracima, moramo biti svjesni gdje ih postaviti kako ne bismo utjecali na kvalitetu izgradnje ili testiranja aplikacija. Česta etapa koja se dodaje kao poseban korak jest provjera razlika u paketima. Uzmimo na primjer da gradimo aplikaciju koja postoji duže vrijeme. Razvojni inženjeri morat će osigurati izvršnu datoteku koja se na određenom okruženju pokreće prvi put te izvršnu datoteku koja može ažurirati već postojeće okruženje. Često u paralelnom procesu izgradnje obje izvršne datoteke zbog različitih ovisnosti mogu uzrokovati razlike u paketima za izgradnju. Upravo je to razlog zašto bismo prije testiranja voljeli provjeriti koje su to razlike koje moramo nadomjestiti za obje izvršne datoteke. Predefinirani koraci određeni su standard koji bismo trebali imati, ali ne moramo ih se strogo držati.

U prethodnoj cjelini spominjali smo razliku između *Self-Hosted* i *Managed* okruženja. U slučaju odabira *Managed* okruženja postavlja se pitanje hoće li biti bolje instalirati sustav distribucije direktno na poslužitelj ili Docker. Na primjeru alata Jenkins brže je i jednostavnije implementirati/instalirati sustav na Docker okruženje. Iz primjera mogli smo vidjeti da pomoću jedne naredbe možemo podići cijeli radno okruženje. Takav pristup nije uvijek najbolji. Docker okruženju ima svoje mane, kao što su problemi s pravima pristupa i slučaj koji se često dešava: što ako unutar Docker okruženja moramo pokrenuti dodatno Docker okruženje? Često u etapi izgradnje razvojni inženjeri vole koristiti Docker kao pomoć u simulaciji okruženja koje je slično produkcijskom. Da bi to funkcioniralo, morali bismo se pomno potruditi. U tom slučaju voljeli bismo instalirati sustav distribucije direktno na poslužitelj.

U procesu konfiguracije sustava distribucije ključno je poznavati potrebe projekta, odnosno programskog kôda. Proces konfiguracije sustava distribucije izvršava se u samim počecima kada prikupljamo zahtjeve sustava. Tvrtke takav proces vole implementirati tijekom radionica otkrivanja (engl. *discovery workshops*). Cilj DevOpsa u tom procesu izrazito je dobro pratiti arhitekturu koju predlažu razvojni inženjeri te na temelju toga odrediti koju vrstu posla projekt zahtijeva, koje sve vanjske alate moramo koristiti i kako što jednostavnije pripremiti poslove za razvojne inženjere. Cilj konfiguracije sustava jest to da DevOps inženjer pripremi okruženje za razvojne inženjere i nakon toga radi minimalne dorade.

PITANJA ZA PONAVLJANJE:

1. Nabrojite i objasnite vrste linija projekata.
2. Objasnite razliku u deklarativnoj i skriptnoj sintaksi poslova.
3. Što je YAML?
4. Razlika u postavljanju poslova kroz sintaksu YAML-a i Groovya.
5. Zašto je bitno da linija komunicira s udaljenim repozitorijem?
6. Što je povratna veza (engl. *webhook*)?
7. Nabrojite zadane vrste koraka u liniji.
8. Zašto je korak testiranja izrazito bitan?

5.12. Proces izrade (konstrukcija) aplikacija

U procesu izgradnje, odnosno konstrukcije aplikacije, osim što se izvršava proces izgradnje kôda, automatizacija izgradnje uključuje korištenje alata za provjeru sigurnosti kôda i provjeru slijede li se najbolje prakse. Etapa izrade obično završava u koraku generiranja artefakata, gdje stvaramo paket spreman za objavu aplikacije. Proces izgradnje možemo podijeliti na četiri djela:

- prevođenje – izgradnja aplikacije
- provjera programskih i stilskih pogrešaka (engl. *linting*)
- analiza kôda – provjera kvalitete kôda
- generiranje artefakata – pakira aplikaciju za izdavanje ili implementaciju.

5.13. Prevođenje

Prvi korak u fazi izrade odnosi se na prevođenje aplikacije. Jezici koji zahtijevaju prevođenje generiraju binarne datoteke, dok na interpretiranim jezicima potvrđujemo da imamo potrebne ovisnosti i alate za uspješnu izgradnju aplikacije. Rezultat prevođenja može biti npr. binarna datoteka, zapakiran kôd, paket koji se može instalirati, mrežna stranica i slično. Glavni cilj dobiti je nešto što možemo pokrenuti i testirati. Za izgradnju aplikacije, neovisno o programskom jeziku, potrebna je konfiguracijska datoteka u kojoj se nalaze naredbe za izgradnju. Uz datoteku za izgradnju postoji datoteka koja opisuje ovisnosti drugih paketa potrebnih za izgradnju. Spomenute datoteke nastaju u procesu razvoja aplikacije. Datoteka za opisivanje izgradnje nastaje u procesu kreiranja projekta, a datoteku s ovisnostima generira upravitelj paketima. Svaki put kada razvojni inženjer odluči dodati specifičnu ovisnost, pokreće naredbu za instalaciju paketa pomoću upravitelja paketa. Obje će datoteke linija iskoristiti za instalaciju i postavljanje okruženja. Unutar etape izgradnje zaduženi smo napisati naredbe koje će inicirati proces. Postoje specijalizirane linije unutar kojih nismo zaduženi definirati naredbe, no prilikom stvaranja linija zaduženi smo odabrati za koji programski jezik radimo izgradnju.

Na primjeru prevođenja Jave – prvo ćemo morati na naš poslužitelj na kojem ćemo

izvršavati izvođenje instalirati Javu. Nakon toga naš je zadatak instalirati upravitelj paketa ovisno o tome što su razvojni inženjeri koristili za izgradnju aplikacije. Vodeći popularni alati za izradu i upravljanje paketima u Javi jesu Apache Maven i Gradle. U primjeru radit ćemo s Mavenom. Obrađivanje ovisnosti i prevođenje odrađuje se jednom naredbom: **`mvn compile`**.

```
git    checkout
cache  restore
mvn    compile
cache store
```

Naredba **`cache`** detektirat će prevedeni izlaz u ciljnom direktoriju i pohraniti ga za buduću upotrebu. To će nam pomoći u budućim gradnjama projekta.

5.14. Linting i analiza koda

Programiranje zahtijeva disciplinu. Moramo rješavati teške probleme slijedeći dobre prakse i pridržavajući se jednakog stila kôdiranja u timu. Automatizirani alati za analizu pomažu nam izbjeći kôd koji nije napisan prema dobrim preporukama i praksama. Općenito, postoje tri klase alata za analizu kôda:

1. **Linting**: linteri poboljšavaju kvalitetu kôda ukazujući na problematične i dijelove koji se teško održavaju. Linteri koriste skup pravila za provedbu jedinstvenog standarda koji poboljšava čitljivost za tim.

2. **Kvaliteta**: alati za kvalitetu koriste metriku za pronalaženje mjesta na kojima se kôd može poboljšati. Prikupljeni pokazatelji uključuju broj redaka, pokrivenost dokumentacije i razinu složenosti.

3. **Sigurnost**: alati za sigurnosnu analizu skeniraju kôd, označavaju dijelove koji mogu uzrokovati ranjivosti i provjeravaju imaju li ovisnosti poznate sigurnosne probleme.

Budući da nitko ne želi postaviti u rad nesiguran ili pogrešan kôd, moramo dizajnirati CI liniju koja zaustavlja izvođenje kôda kad se pronađu pogreške. Kada CI linija pokreće prave alate za analizu kôda, programeri koji pregledavaju kôd mogu se usredotočiti na bitna pitanja kao što je „Rješava li ovaj kôd pravi problem?"

Linting u Javi ponekad je zbunjujući s obzirom na to da Java programeri koriste IDE-ove koji lintiraju kôd u stvarnom vremenu. Bez obzira na to koliko su napredni linteri u IDE-u, to nije dovoljno. Za uključivanje lintera prilikom izgradnje potrebno je dodati ovisnost u pom.xml, datoteku u koju su pohranjene sve ovisnosti potrebne prilikom prevođenja.

```
<properties>
<maven.checkstyle.plugin.version>3.1.2</maven.checkstyle.plugin.version>
</properties>
```

Osim spomenutog dodavanja potrebno je aktivirati dodatak u procesu izgradnje Java kôda.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-checkstyle-plugin</artifactId>
            <version>${maven.checkstyle.plugin.version}</version>
            <configuration>
                <failsOnError>true</failsOnError>
                <failOnViolation>true</failOnViolation>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Sada možemo pokrenuti checkstyle s:

```
git     checkout
mvn checkstyle:checkstyle
```

Nakon analize kôda želimo provjeriti i pogreške koje smo potencijalno uzrokovali razvojem. Alat PMD alat je za statičku analizu kôda, koji će javiti sve potencijalne buduće probleme u razvoju. Možete skenirati kôd sa zadanim pravilima za Javu sa sljedećim:

```
git     checkout
wget

HTTPS://github.com/pmd/pmd/releases/download/pmd_releases%2F6.46.0/
pmd-bin-6.46.0.zip

unzip pmd-bin-6.46.0.zip

./pmd-bin-6.46.0/bin/run.sh pmd -d . -R rulesets/java/quickstart.xml
-f text
```

S obzirom na to da smo utvrdili da ne postoje potencijalni problemi koji bi mogli nastati daljnjim razvojem kôda, sljedeći korak sigurnosna je analiza kôda. Na tržištu postoji izrazito cijenjen alat naziva Infer. Alat Infer kreiran je od strane tvrtke Facebook i služi za pronalaženje pogrešaka koje bi stvorile sigurnosni problem na aplikaciji tijekom izvođenja, poput curenje memorije i slično. Facebook je napravio ovaj alat i koristi ga za detekciju problema u njihovim mobilnim aplikacijama.

```
git     checkout
cache   restore
mvn     clean
curl    -sSL    "HTTPS://github.com/facebook/infer/releases/download/
v1.1.0/infer-linux64-v1.1.0.tar.xz"    |     tar   -xJ
./infer-linux64-v1.1.0/bin/infer run -- mvn package
```

```
artifact push job --expire-in 1w infer-out

./infer-linux64-v1.1.0/bin/infer analyze --fail-on-issue
```

5.15. Generiranje artefakata

Objavljeni paket uključivat će sve što je potrebno za pokretanje ili instalaciju aplikacije, uključujući:

- kôd aplikacije
- instalacijske skripte
- ovisnosti i knjižnice
- metapodatke aplikacije
- dokumentaciju
- podatke o licenci.

Konačni paket može se objaviti na mrežnoj stranici, u bazi podataka paketa, u registru kontejnera ili se može izravno implementirati u QA ili produkcijsko okruženje. U slučaju Jave potrebna nam je .jar datoteka koju bismo poslali na okruženje. Nju možemo dobiti pomoću naredbe: **`mvn package`**.

```
git    checkout
cache  restore
mvn    package
cache store
```

Nakon izgradnje naš sustav za distribuciju sve će stvari koje smo pohranili predstaviti u artefaktima posla. Artefakti su sve izvršne datoteke koje smo gradili. Osim izvršnih datoteka unutar artefakata možemo pronaći izvještaje gradnje koje smo pohranjivali.

Kako bismo isporučili što kvalitetniji kôd, izrazito je bitno pratiti dobre prakse implementacije koraka u liniji. Bez provođenja koraka za provjeru programskih i stilskih

pogreška kao i analize kvalitete sigurnosti kôda moglo bi doći do problema – da se projekt ili produlji s izradom ili ne uspije. Često u praksi možemo vidjeti primjere gdje je tim za provođenje projekata vršio izrazito veliki pritisak na razvojne inženjere. Takav pristup rezultira time da se dosta implementacijskih stvari ne stigne provjeriti te ako uz to ne provjerimo kvalitetu kôda tijekom izgradnje aplikacije u liniji, krajnji korisnici koristit će alat koji nije u potpunosti funkcionalan.

PITANJA ZA PONAVLJANJE:

1. Nabrojite četiri obavezna djela u izgradnji aplikacija.
2. Kako funkcionira korak prevođenja u pipelienu?
3. Zašto je naredba cache izrazito bitna?
4. Što je linting?
5. Što je statička analiza kôda?
6. Nabrojite tri klase alata za analizu kôda.
7. Za što koristimo alat Infer?
8. Što su artefakti?

5.16. Izvršavanje testova

Kao i kod svakog drugog razvoja proizvoda, softver je potrebno testirati prije nego što dođe do naručitelja. Očigledan primjer testiranja nekog proizvoda korištenje je istog neko vrijeme kako biste bili sigurni da se ponaša prema očekivanjima. Ovo je način na koji je većina razvojnih inženjera u prošlosti provjeravala svoj rad prije isporuke. Razvojni inženjer uz svoj razvoj paralelno piše i testove koji će testirati funkcionalnosti. Često se u ručnom pokretanju testova dogodi problem da zaboravimo testirati određene dijelove kôda ili ne shvaćamo posljedice izmjena koje su se desile. Možda ne možemo biti sigurni da će naš kôd raditi u okruženju koje nije naše računalo. Čak i ako napravite sve kako treba, testiranje je uglavnom dosadna aktivnost koja oduzima vrijeme. U praksi često se odvaja proces testiranja u posebne odjele za osiguravanje kvalitete. Tako najčešće odjel za osiguravanje kvalitete testira funkcionalnosti i ponašanja aplikacije, a ne razvojni inženjeri. U tom procesu gubi se i ogromna količina vremena jer inženjeri kvalitete moraju redovito postavljati pitanja zašto je ovo ovako napravljeno? Što se mora desiti sljedeće? Oba opisana načina izrazito su naporni procesi koji oduzimaju veliku količinu vremena i zato taj proces koji se redovito ponavlja želimo automatizirati.

5.17. Izvršavanje implementiranih testova

Kada pričamo o testiranju, tema je izrazito široka i postoje deseci različitih vrsta testova, ovisno o potrebama razvojnih inženjera. Kako bi dvije strane mogle zajedno surađivati, razvojni inženjeri pišu testove koji će testirati funkcionalnost te je uz upravitelja projekta potrebno raspisati scenarije testiranja. Zamislimo da razvijamo značajku koja će omogućiti veću prodaju na određenoj *web*-trgovini. Scenarij bi trebalo definirati tako da ako klijent odluči kupiti tri proizvoda, potrebno je primijeniti popust od 20 %. Razvojni inženjeri napisat će testove koji će testirati programske funkcionalnost, a inženjeri kvalitete napravit će testove koji će potvrditi te funkcionalnosti. Zadatak je DevOps inženjera u liniji osigurati etapu koja će testirati funkcionalnosti razvojnog inženjera, a nakon toga testiranje funkcionalnosti koje je napravio tim za osiguravanje kvalitete. Uobičajena je praksa izrazito brzo izvršiti testiranja, pa se proces može ubrzati paralelnim etapama koje paralelno testiraju više testova. Kao DevOps inženjeri

moramo biti svjesni da je proces stvaranja testova izrazito dug, s obzirom na to da se ne mogu odmah razviti testovi za sve funkcionalnosti. U početku, kada nemamo sve testove, svaka promjena nepoznat je rizik. S vremenom, kako će se dodavati sve više testova i scenariji testiranja bit će bolje definirani, rizik će se smanjivati. Zato je bitno automatizaciju učiniti sastavnim procesom razvoja od samog početka. U procesu izgradnje aplikacije spominjali smo izvještaje koji se generiranju na temelju procesa linije. U procesu izvršavanja testova izvještaji koje će nam dati testovi maksimalno su bitni i za ostatak organizacije, kako bi se mogli odrediti ključni koraci za sljedeću fazu razvoja alata. Koji je rezultat cijelog procesa

5.18. Implementacija testova u CI

U redovitim intervalima razvojni inženjeri integriraju promjene s promjenama drugih članova tima, koristeći alat za upravljanje verzijama kao što je Git. Dobro uspostavljen tim ima snažnu politiku kontinuirane integracije i CI linija, koji će, nakon što detektira promjene, preuzeti novi kôd iz spremišta i pokrenuti testove na njemu. Takav pristup razvojnim inženjerima vrlo rano, s dobrim stupnjem povjerenja, daje do znanja da je nešto prestalo raditi. Kao i u mnogim aspektima života, u strategiji testiranja postoje kompromisi. Na razvojnim je inženjerima i inženjerima kvalitete da pronađu kompromis, osiguravajući kvalitetu i učinkovitost. Bitno je imati na umu da su inženjeri osiguranja kvalitete najvažniji za dostavu visokokvalitetnih proizvoda, stoga je važno dopustiti im da učinkovito obavljaju svoj posao. Njihova je zadaća da pomognu razvojnim inženjerima i tvrtkama da isporuče kvalitetan softver. Iako postoje inženjeri za kvalitetu, nikako se ne smije dogoditi da se razvojni inženjeri oslanjaju na činjenicu da će netko iza njih provjeravati kôd. Takvim pristupom doći će do puno više grešaka i smisao cijelog procesa izrade softvera bit će doveden u pitanje. Razvojni inženjeri trebaju raditi s pristupom da razvijaju kôd koji nema grešaka, a onda ako i bude grešaka koje će naći inženjeri kontrole kvalitete, tih grešaka bit će puno manje.

5.19. Integracija testova s distribucijom

Postoji više pristupa testiranju kôda. Imamo slučaj kada radimo testiranje kôda na promjene u repozitoriju, u tom slučaju radimo provjeru jesmo li spremni spojiti kôd iz naše grane s glavnom granom. Kada radimo takav tip testova, u tom slučaju ne bismo radili sve testove koje posjedujemo zbog izrazito puno vremena koje bismo morali potrošiti na izgradnju i provjeru. U takvom slučaju praksa je kreirati samo testove koje su razvili razvojni inženjeri. Kada gradimo aplikaciju iz glavne grane, bilo to na okidač spajanja grana ili kada ručno pokrećemo posao za izgradnju aplikacije, u tom slučaju željeli bismo pokrenuti sve dostupne testove.

S obzirom na zahtjevnost stvaranja testova timovi za osiguravanje kvalitete imaju alate unutar kojih generiraju testove prema scenarijima korištenja aplikacije. Jedan od najpoznatijih alata jest TestRail. TestRail je alat za upravljanje slučajevima testiranja. Koriste ga razvojni inženjeri, inženjeri kvalitete i voditelji timova za upravljanje, praćenje i organiziranje testiranja softvera. Alati poput TestRaila omogućuju stvaranje testnih slučajeva po kategorijama, organizaciju potrebnih stvari za izvršavanje testova, izvođenje testnih slučajeva i praćenje rezultata testova. Rezultati su od izrazite važnosti, s obzirom na to da na temelju njih možemo odrediti koje korake možemo staviti u paralelni način izvršavanja, takav način nazivamo horizontalni način skaliranja i njime povećamo brzinu izvršavanja, no u tom slučaju izrazito je teško shvatiti rezultate i gdje su nastale pogreške ako su nastale. Baš u takvim situacijama odlični su alati za upravljanje testova jer nam mogu pomoći u sumiranju problema. Iako smo smanjili dugotrajno izvršavanje testova, potencijalno skaliranje može prouzročiti da se određeni testovi sporije izvršavaju. Stavljanjem svega na jedan izvještaj možemo na lagan način pronaći rješenje.

Unutar etape izvršavanja testa dovoljno je pozvati funkciju koja se povezuje s alatom za osiguravanje kvalitete. Na primjeru komunikacije između TestRaila i Jenkinsa dovoljno je pozvati:

```
stage('Test'){
       steps{
          parallel(
```

```
            "Unit  testovi":{
                   testRail(

                      testrailProject:UnitTest,
                      testrailSuite:*,
                      junitResultsGlob:'lokacija_izvještaja',
                      createNewTestcases:  false]

                  )
            },
            "Integracijski   testovi":{
                   testRail(

                      testrailProject:IntegreationTest,
                      testrailSuite:*,
                      junitResultsGlob:'lokacija_izvještaja',
                      createNewTestcases:  false]

                  )
            })
        }
}
```

Automatizirano testiranje, odnosno kontinuirano testiranje kroz liniju osigurava rano pronalaženje pogreška i omogućuje popravke prije nego što krajnji korisnici dožive bilo kakav prekid. Ideja iza takvog pristupa jest da linija „brzo podbaci" jer je puno lakše popraviti grešku kada se rano pronađe.

PITANJA ZA PONAVLJANJE:

1. Zašto je bitno u linija procesu imati korak testiranja?
2. Tko piše automatizacijske testove?
3. Zašto je bitno da se na svaku sinkronizaciju kôda s udaljenim repozitorijem pokreće CI?
4. Što rade razvojni inženjeri, a što inženjeri kvalitete kôda u procesu CI?
5. Kada DevOps inženjer kreće u implementaciju koraka za testiranje?
6. Što je TestRail alat?

5.20. Distribucija aplikacija

Posljednja etapa ispravno napravljene linije jest kontinuirana implementacija. Kontinuirana implementacija proširenje je kontinuirane isporuke s ciljem objave kôda na određeni poslužitelj. Bitno je imati na umu da naša linija mora biti izrazito dobro postavljena jer se u ovoj fazi smije pojaviti isključivo kôd koji se može implementirati. Kada razmišljamo na koji poslužitelj šaljemo kôd, to ovisi o našoj situaciji. Praksa je takva da ne želimo direktno implementirati kôd naručitelja, već na poslužitelj kojem može pristupiti tim za osiguravanje kvalitete kako bi potvrdio funkcionira li kôd ispravno (iako postoje slučajevi gdje postoje linije koje direktno na poslužitelj implementiraju zadnje ispravke).

Prilikom razmišljanja kako napraviti distribuciju aplikacije na poslužitelj moramo biti svjesni da u tom slučaju postoje dva načina. Prvi se način odnosi na to da sami ručno podesimo skripte – na taj način dobivamo veću fleksibilnost u odrađivanju dodatnih konfiguracija ako su potrebne ili možemo koristiti integrirane dodatke sustava za distribuciju. U slučaju odabira integriranih dodataka dobivamo „plug and play" pristup, gdje moramo navesti jedino podatke poslužitelja.

Ako radimo u timu, moramo osigurati našem sustavu za distribuciju prava za implementaciju rješenja. Vrlo često taj korak bude izvršen ručno i u tom slučaju nismo sigurni tko je stvarno pokrenuo zadatak za implementaciju kôda na poslužitelj.

5.21. Implementacija koda ručnim podešavanjem

U ovom koraku, kada spominjemo ručno podešavanje, razmišljamo o tome da moramo sami podesiti etapu objave kôda. U tom procesu morat ćemo koristiti alat za prijenos kôda na poslužitelj i morat ćemo definirati naredbe za pokretanje kôda. Transport kôda možemo napraviti naredbama poput SCP ili SFTP. *Secure Copy* ili SCP siguran je prijenosa datoteka između dva računala na mreži. SCP koristi SSH za poboljšanu sigurnost i prilikom prijenosa tražit će od vas šifru za autentifikaciju ili ćete morati učitati SSH ključ. Osnovna naredba za prijenos datoteka izgleda:

```
scp [putanja_do_datoteke_vase_racunalo] [korisnicko_ime@udaljeno_ra-
cunalo]:[putanaj do odredisne_datoteke]
```

SFTP (*Safe File Transfer Protocol*) dio je SSH protokola dizajniranog za siguran prijenos datoteka između udaljenih sustava. Omogućuje korisnicima pregled, upravljanje i promjenu dopuštenja datoteka i direktorija na udaljenim poslužiteljima. Za rad sa SFTP-om prvo moramo ostvariti vezu s poslužiteljem.

```
sftp [korisnicko_ime]@[udaljeno_racunalo]
```

Nakon što se uspješno povežemo, moramo prenijeti datoteku. To radimo pomoću naredbe put, za prijenos datoteka s lokalnog sustava u početni direktorij udaljenog poslužitelja. To će značiti da naknadno još moramo i prenijeti datoteku na lokaciju gdje je želimo pokrenuti.

```
put [naziv_datoteke]
```

Glavna razlika između SCP-a i SFTP-a jest u tome što je SCP protokol koji omogućuje siguran prijenos datoteka s lokalnog računala na udaljeni poslužitelj, dok je SFTP protokol koji omogućuje pristup datotekama, prijenos i upravljanje preko pouzdanog toka podataka koji je brži od SCP-a. No SCP je u više slučajeva jednostavnije izvršiti s obzirom na jednostavnost koraka koje moramo odraditi.

Pa bi ručno podešena etapa za izgradnju u Docker okruženju izgledala ovako:

```
stage('Deploy'){
  steps{
    sh"Docker   save  app-v1     >     app-v1.tar"
    sh "scp     app-v1.tar buser@localhost:."
   sh "ssh buser@localhost Docker load < app-v1.tar && Docker-compose up -d"
       }
}
```

Kao što možemo vidjeti, nakon učitavanja Docker slike možemo izvršiti naredbe koje su nam potrebne za dodatnu implementaciju.

5.22. Implementacija koda s bibliotekama

Alat Jenkins ima izrazito snažnu integraciju s alatima za objavu. Alati za objavu omogućuju slanje kôda direktno na server ili kontejner. Alat uzima artefakt iz gradnje te ga šalje na poslužitelj. Tijekom procesa objave kôda alat funkcionira na principu sustava za verzioniranje kôda te u slučaju pogrešaka možemo na izrazito jednostavan način povratiti sustav iz pogreške. To je izrazito velika prednost u odnosu na ručno ažuriranje produkcije. Primjer kako bismo to mogli napravili:

```
agent   localhost
stage('Deploy'){
  steps{
    Docker.withRegistry('', registryCredential) {

      DockerImage.load()
     }

   }
}
```

Timovi koje žele poboljšati performanse i kvalitetu aplikacije trebaju pouzdanu proceduru isporuke. Zbog spomenutog razloga CI je dobio nastavak CD. Takvim pristupom timovi mogu brže odgovoriti na tržišne promjene, sigurnosne izazove, potrebe kupaca i pritiske na troškove. Na primjer, ako je potrebna nova sigurnosna značajka, vaš tim može implementirati CI/CD s automatskim testiranjem kako bi se popravak brzo i pouzdano uveo u proizvodne sustave s visokim povjerenjem. Ono za što su prije bili potrebni tjedni i mjeseci, sada se može učiniti u danima ili čak satima.

PITANJA ZA PONAVLJANJE:

1. Što je kontinuirana implementacija?
2. Nabrojite i objasnite pristupe kontinuiranoj implementaciji?
3. Što je implementacija ručnim podešavanjem kôda?
4. Objasnite razliku između SCP-a i SFTP-a.
5. Što je implementacija kôda pomoću biblioteka?

5.23. Distribucija mobilnih aplikacija

Proces distribucije za mobilne aplikacije uštedjet će ogromno vrijeme razvojnim inženjerima mobilnih aplikacija. Skratit će vrijeme priprema za slanje aplikacije na trgovinu aplikacija, popravljanje kritičnih pogrešaka i učitavanje snimki zaslona. Automatizacija će u ovom procesu optimizirati proces jer će svaki član tima moći fokusirano raditi na ispravljanju pogrešaka ili izradi ažuriranja za aplikaciju. U trenutku kad se odlučimo objaviti aplikaciju na trgovinu aplikacija, moramo biti svjesni da će naša aplikacija proći kroz rigorozne provjere. Kako bismo ubrzali proces objave aplikacije, bit će nam bitno da izvorni kôd bude pisan u ispravnom formatu koji razumije cijeli tim, a linija provjere i pronalaska pogrešaka taj će proces automatizirati. Jedan od najvažnijih aspekata implementacije CI-a/CD-a za mobilne aplikacije jest učenje o principima koji mogu povećati vrijednost vašeg trenutnog tijeka rada. U ovom procesu vidjet ćemo odmah određene razlike u odnosu na tradicionalan razvoj CI/CD linija za aplikacije.

5.24. Načela CI-a/CD-a za mobilne aplikacije

Izgradnja aplikacija za mobilne uređaje usvaja sve bitne korake kao i izgradnja bilo koje druge aplikacije. U procesu izgradnje mobilne aplikacije također ćemo imati osnovne korake kao što su izvor, izgradnja, testiranje i objava. No bitno je razumjeti da će postojati i korak između testiranja i objave s nazivom potpisivanje kôda. Potpisivanje kôda u mobilnom razvoju od izrazite je važnost. Trgovinama aplikacija izrazito je bitno pratiti jesu li provjerili ispravnost kôda i to povezuju s potpisom. Objava u etapi nije objava na određeni poslužitelj, već slanje na trgovinu aplikacija.

5.25. Izrada certifikata

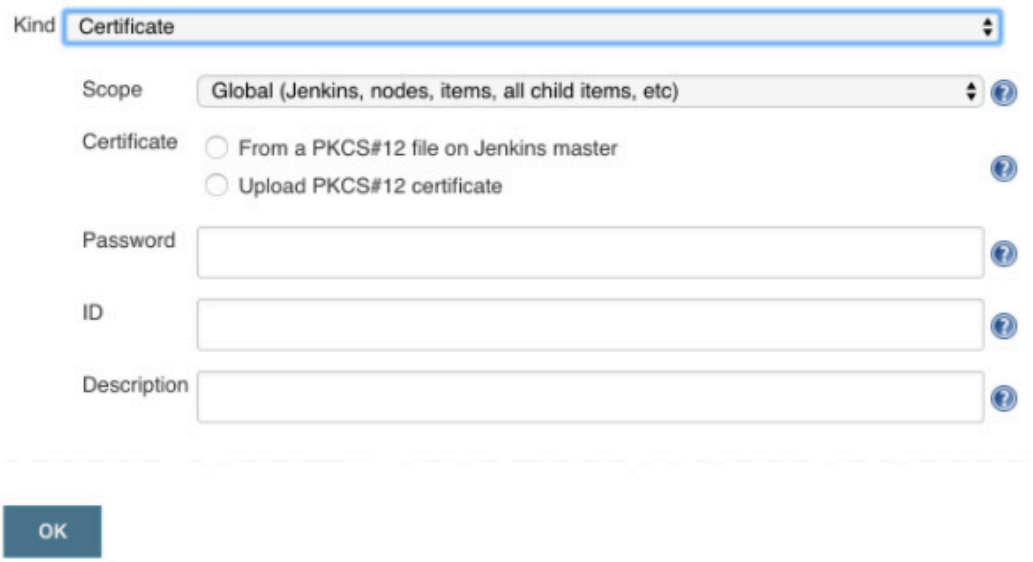
Digitalni potpis postavlja se na izvršnu datoteku i sve ostale uključene datoteke za pokretanje aplikacije. Ono što moramo imati na umu jest to da bilo tko može kreirati certifikat, ali trgovina ga mora potvrditi. Zahtjev za potpisivanje certifikata naziva se *Certificate Signing Request* (CSR). *Certificate Authoritiy* (CA), odnosno trgovina aplik-

acija, mora pregledati pojedinosti o tome tko traži izdavanje certifikata i prema provjeri kôda te istinitosti informacija izdat će validan certifikat. CSR se generira na lokalnom računalu, zajedno s parom privatnog i javnog ključa. Osoba koja je generirala CSR kada dobije privatan ključ, njezin je zadatak čuvati taj ključ od neovlaštenog pristupa. Certifikat koji dobijemo od CA-a moramo dodati u naše spremište certifikata. Uobičajen je proces da dva puta kliknemo i certifikat će se instalirati na računalo. Unutar certifikata nalaze se naši podaci i određeni parametri koje je dodijelila trgovina, a koji nam daju ovlasti za potpisivanje aplikacije.

5.26. Automatizacija procesa

U ovom procesu pokazat ćemo primjer kako napraviti potpisivanje i izradu mobilne aplikacije za operativni sustav Android. Prije nego što krenemo u samu realizaciju, moramo imati na umu da je izrazito opasno dodati certifikat za potpisivanje na udaljeni repozitorij. Tim postupkom objavljujete svoj potpisni certifikat cijelom svijetu. Na taj način dopustili biste potencijalnim napadačima da naprave lažne aplikacije i predstavljaju se poput vas. Čak i ako ste na privatnom sustavu, postavljate certifikat na računalo svakog razvojnog inženjera, što povećava šanse da certifikat postane javan. U praksi ovaj problem rješavamo tako što prenosimo certifikat za potpisivanje u datotečni sustav Jenkinsa i kasnije ga referenciramo kroz kôd.

Kako bismo uspješno podesili certifikat, potrebno je otići u postavke Jenkinsa i pronaći gumb s nazivom „Credentials”. Morat ćemo dodati barem jedan certifikat koji će se koristiti za potpisivanje aplikacija. Vidjet ćemo gumb s nazivom „Add Credentials” i nakon klika na gumb otvorit će nam se ekran gdje moramo odabrati vrstu vjerodajnica. Za podešavanje ove postavke odabrat ćemo vrstu: „Certificate”.

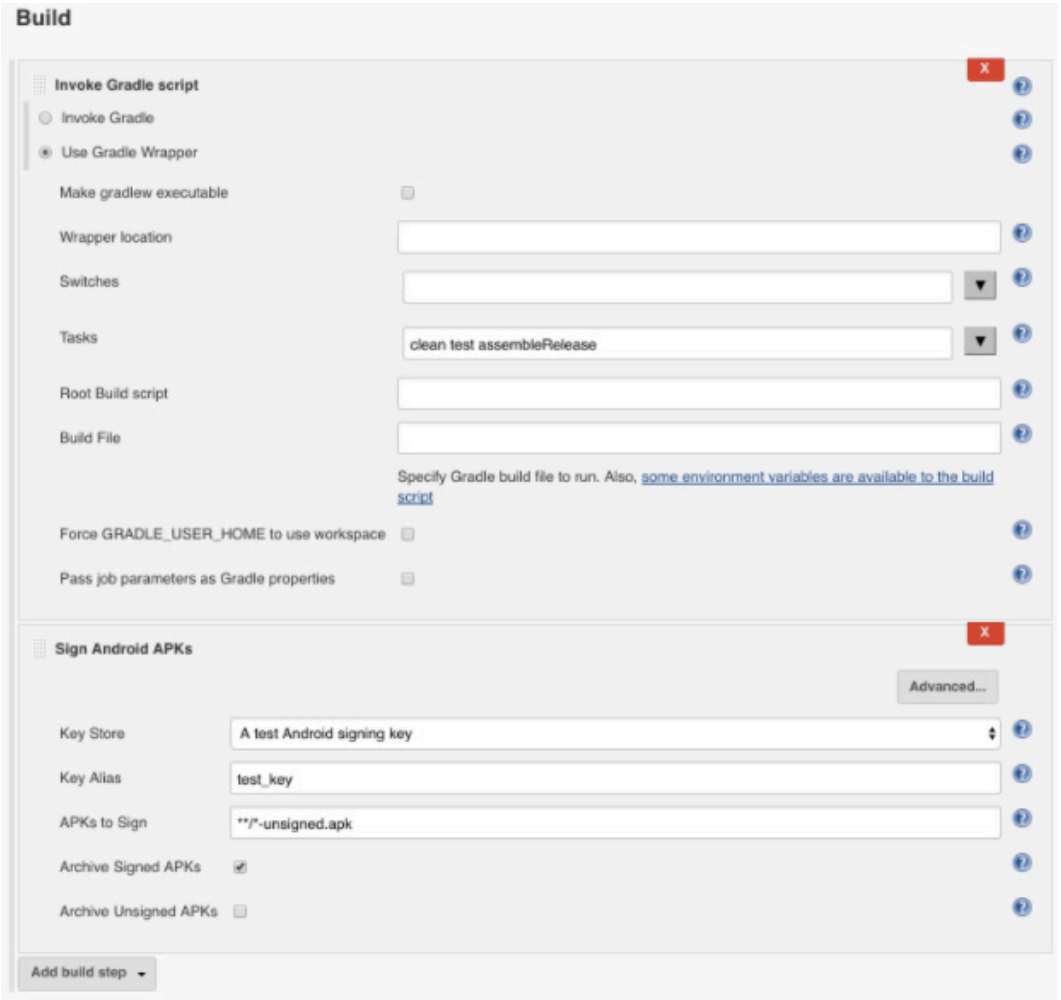


Slika 5.25.1. Dodavanje Certifikata za potpisivanje

Kao što možemo vidjeti, Jenkins podržava samo „PKCS12” certifikat. Nažalost, novije verzije Android Studija omogućuju izvoz certifikat kao „JKS” certifikat, što je nekompatibilno. Da bismo riješili ovaj problem, koristit ćemo alat naredbenog retka „keytool”, koji može pretvoriti našu „JKS” datoteku u „PKCS12” format pomoću sljedeće naredbe:

```
keytool -importkeystore -srckeystore {PUTANJA_DO_JKS_DATOTEKE} -srcstoretype JKS -deststoretype PKCS12 -destkeystore ConvertedCertificate.p12
```

Nakon što dobijemo datoteku „PKCS12”, možemo prenijeti certifikat. Obavezno moramo upisati lozinku certifikata prije prijenosa ili će prijenos prijaviti neuspjeh. Nakon što smo sve izvršili, spremni smo napraviti prvi posao za izgradnju mobilne aplikacije. Jedina napomena kod kreiranja Jenkins posla jest da za izgradnju izvornog kôda moramo ostaviti praznu definiciju „signingConfig” za „buildType” koji želite potpisati s Jenkinsom. U protivnom stvorit će se nepotpisani APK koji se zatim mora potpisati pomoću dodatka.



Slika 5.25.2. Posao za potpisivanje kôda

Kao što vidimo na slici 5.25.2., unutar Jenkins posla moramo dodati korak pod nazivom „Sign Android APKs”. On će se izvršiti odmah nakon što se aplikacija izgradi.

5.27. CI/CD alati za izgradnju za mobilnih aplikacija

U prethodnom primjeru koristili smo alat Jenkins, no za izgradnju mobilnih aplikacija postoje specijalizirani sustavi koji će pomoći u još boljoj automatizaciji. Jedan od popularnijih alata jest Fastlane. Fastlane je alat specijaliziran za izradu i implementaciju mobilnih aplikacija, a Jenkins je alat za automatizaciju izgradnje, implementaciju bilo koje vrste aplikacije (internet, mobilna, stolna računala). S Jenkinsom možete postići iste korake kao i u Fastlaneu, ali trebat ćete više vremena za postavljanje. Fastlane

ne može imati velik broj mogućnosti kao i Jenkins. Primjerice, ako želite imati dnevnu gradnju aplikacije na Fastlaneu, trebali biste to postaviti putem Cron Jobsa ili Jenkinsa.

Fastlane je izgrađen pomoću Ruby alata te da bismo koristili Fastlane na našem računalu, moramo instalirati Ruby. Fastlane nema grafičkog sučelja, pa sve funkcionalnosti pokrećemo iz naredbenog retka. Fastlane uvelike olakšava pripremu aplikacije za trgovinu. Stoga ga DevOps inženjeri vole kombinirati s Jenkinsom. Ideja je alat Fastlane pokrenuti unutar Jenkins posla.

```
lane:beta  do
    increment_build_number
    build_app
    upload_to_testflight
end

lane:appstore   do
   capture_screenshots
   build_app
   upload_to_app_store
   slack
end
```

Možemo primijetiti da ga je izrazito jednostavno konfigurirati. Bitno je da se Fastlane nalazi unutar izvornog kôda. Za pokretanje izgradnje pokrećemo:

```
bundle exec fastlane build
```

Nakon pokretanja gornje naredbe aplikacija je spremna za daljnju fazu, odnosno objavu.

Specijalizirana rješenja mogu izrazito ubrzati proces u stvaranju izvršnog proizvoda u odnosu na rješenja koja su specijalizirana za različita rješenja. DevOps inženjeri često

imaju problem da moraju koristiti i održavati veliku količinu alata. No postoje specijalizirana rješenja koja se mogu integrirati na našu postojeću arhitekturu tehnologija. Uobičajeno je u procesu kada implementiramo specijalizirana rješenja da budu kompatibilna s postojećim rješenjima. To možemo primijetiti i na primjeru gdje integriramo Fastlane s Jenkinsom i dobivamo još veću efikasnost.

PITANJA ZA PONAVLJANJE:

1. Zašto koristiti automatizaciju u izgradnji mobilnih aplikacija?
2. Koji dodatan korak postoji u izgradnji mobilnih aplikacija u odnosnu na standardne linije?
3. Što je proces potpisivanje kôda?
4. Što je Certificate Authority?
5. Što je Certificate Signing Request?
6. Što je keytool alat?
7. Što je Fastlane?

6. Infrastruktura u obliku koda

U OVOM POGLAVLJU NAUČIT ĆETE:

- razlikovati pristupe definiranju infrastrukture kao koda
- razlikovati infrastrukturu kao kôd od ručne konfiguracije
- mogućnosti dijeljenja konfiguracije i stanja s timom
- razlikovati sustave za sinkronizaciju stanja infrastrukture
- korake izmjene infrastrukture kao koda putem Git repozitorija.

6.1. Infrastruktura u obliku koda

Izgradnja današnjih aplikacija zahtijevaju izrazito agilna okruženja. Danas sve češće tvrtke koje imaju vlastita okruženja i poslužiteljske sale shvaćaju koliko je skup proces održavanja i zahtijeva posvećene timove. Posvećeni timovi u procesu pripreme okruženja za razvojne timove često stvaraju pogreške i kvarove jer ne postoji čista slika okruženja na samom početku izgradnje aplikacija. Spomenuti problemi potaknuli su DevOps inženjere na razmišljanje kako izgraditi infrastrukturu za aplikaciju ovisno o potrebama, a još bolje bi bilo ako se to može napraviti u procesu CI-a/CD-a. U 2009. godini tada izrazito poznat alat za održavanje CI-a/CD-a pod nazivom „Chef" došao je do rješenja razvijanjem alata koji će kroz programski kôd izgraditi sve resurse koje aplikacija zahtijeva. Bitno je naglasiti da izgradnja resursa mora biti u oblaku kod

određenog pružatelja usluga koji omogućuje podizanje resursa pomoću programskih sučelja aplikacije. To su zapravo bili začeci „Infrastrukture kao kôd" (IaC), odnosno automatiziranog pružanja infrastrukture, omogućivanja implementacije i skaliranje aplikacija u oblaku većom brzinom, s manjim rizikom i smanjenim troškovima. Takav pristup razvojnim inženjerima omogućuje nesmetano podizanje ili održavanja resursa u podatkovnim centrima putem definicijskih datoteka, bez da se moraju uvjeriti posjeduje li tvrtka resurse i infrastrukturu za zahtjeve koji njima trebaju. Jednostavnim riječima, IaC će omogućiti proces zamjene ručnog rada potrebnog za upravljanje resursima i pružanja usluga jednostavnim linijama kôda u oblaku.

6.2. Deklarativna i imperativna definicija infrastrukture kao kôd

Pisanje kôda za IaC ne označava samo znanje pisanja kôda koji stvara infrastrukturu u repozitorij. Postoje različite pristupi u izgradnji infrastrukture kao kôd. Određeni pristupi, odnosno stilovi infrastrukture kao kôd daju nam velike prednosti, dok druge vrste mogu uzrokovati glavobolje. U cjelini deklarativnih i imperativnih definicija naučit ćemo kako odrediti koji pristup najbolje odgovara projektu na kojem radimo.

Krenimo s razumijevanjem razlika između izraza deklarativno i imperativno. Primjer deklarativnog izraza izgledao bi ovako: „Mogu li dobiti čaj na svoj stol u 9 ujutro u ponedjeljak?"

S druge strane, imperativni je izraz izrazito detaljan i u tim koracima rekli bismo sljedeće: „Idi do aparata za čaj u 8:30, zatim uzmi staklenu posudu, napuni je vodom, zatim je vrati u aparat i pokreni zagrijavanje. Nakon što se voda zagrije, stavi u nju čaj i pričekaj 10 minuta…" Odmah primjećujemo razliku u postupcima i duljini i shvaćamo da je imperativni primjer puno duži. Zašto je bitno razumjeti oba pristupa pisanja infrastrukture kao kôd? U imperativnom primjeru, ako bi se pojavio problem u određenom koraku, mogli bismo djelovati i u slučaju nedostatka određenog resursa mogli bismo nadopuniti da kôd zna raditi s pogreškama. Primjerice, ako staklena posuda za čaj nedostaje u aparatu, deklarativnim pristupom ne bismo ništa napravili. Kôd bi preskočio taj dio ili bi se srušio, dok u imperativnim stilu možemo djelovati na pogreške jer pratimo trenutno stanje infrastrukture. Imperativni pristup čini se uvijek boljom opcijom,

no on će zahtijevati dodatne resurse, mjesto za pohranu stanja i akcije koje su se dogodile, dodatna skriptiranja i slično. DevOps inženjeri u ovakvim slučajevima često s razvojnim inženjerima definiraju infrastrukturu i na temelju zahtjeva određuju što je bolje. Primjerice, ako za proces testiranja koji radi tim za osiguravanje kvalitete uvijek moramo dizati novu infrastrukturu na kojoj nitko nije radio, tada ćemo koristiti deklarativni pristup. Zašto deklarativni? Ne moramo provjeravati postoji li već baza podataka ili postoje li određene konfiguracije na poslužitelju jer će to IaC izgraditi ispočetka. Ali ako želimo održavati produkcijsko okruženje klijenta i raditi određene nadogradnja, tada bismo odabrali imperativan pristup.

Konfiguracijski pomak – konfiguracijski pomak je kada se infrastruktura sporo mijenja tijekom vremena. Imperativni stilovi infrastrukture kao kôda teško će se prilagoditi promjenama konfiguracije, budući da su tipično pisani za jednu vrstu životnog ciklusa: ažuriranje, stvaranje, brisanje itd., dok će se deklarativna infrastruktura kao kôd moći lakše prilagoditi promjenama, prijavljujući razlike, a vama prepušta odluku kako dalje.

Ponovno korištenje – ako smo napisali kôd koji ćemo više puta koristiti za podizanje infrastrukture u različitim okruženjima... Nakon pokretanje skripte u testnom okruženju želimo biti sigurni da će se promjena primijeniti na isti način u produkcijskom okruženju. S imperativnim stilom programiranja može doći do različitih stanja u različitim okruženjima, a prednosti ponovnog korištenja smanjene su.

Idempotencija – sposobnost pokretanja iste naredbe i postizanja istog rezultata, odnosno ako je nešto već napravljeno, ne želimo ponovno to raditi. Deklarativna infrastruktura kao kôd može se izvoditi više puta i svaki put stvorit će se novi resurs, odnosno ako već postoji resurs, on će to zanemariti jer ga nije svjestan, dok se imperativna infrastruktura kao kôd može moći pokrenuti samo jednom jer je svjestan da postoji resurs.

6.3. Kako funkcionira infrastruktura kao kôd

Danas se većina svjetske infrastrukture nalazi u podatkovnim centrima u vlasništvu pružatelja usluga oblaka, kao što su Amazon Web Services (AWS), Google Cloud Platform (GCP) i Microsoft Azure. Dok stvaramo novu infrastrukturu, zapravo smo u interakciji sa sučeljima za programiranje aplikacija iza kulisa, koje pruža svaka platforma. Ovo otvara vrata potpuno novoj paradigmi. Infrastruktura se može implementirati i konfigurirati putem poziva prema sučeljima za programiranje aplikacija, koje koriste brojni alati na tržištu poput: Terraform, CloudFormation, Azure Resource Manager i slično. Često se postavlja pitanje – možemo li mi koristiti IaC za resurse koji nisu stvoreni u oblaku? Naravno, alati za IaC podržavaju razne pružatelje usluga, module za pružatelje razvija zapravo zajednica razvojnih inženjera. U slučaju odabira alata za upravljanje infrastrukture kao kôd moramo biti svjesni ograničenja koja nam alati zadaju. Primjerice, u nabrojanim alatima možemo primijetiti CloudFormation i Azure Resource Manager, koji su zaduženi za upravljanje infrastrukturom koja se odnosi samo na njihov poslužitelj. Dok je Terraform izrazito neovisan o platformi i može upravljati bilo kojim pružateljem usluga na tržištu.

Tradicionalno, odnosno ručno postavljanje infrastrukture kod različitih pružatelja usluga kroz grafičko sučelje izrazito je težak i zahtjevan proces. Zahtijevat će od autora proučavanje grafičkog sučelja i poznavanje termina koje koristi pružatelj usluga. Takvim pristupom izrazito često izgubimo pojam i trag što smo zadnje izmijenili i što smo dodali klijentu. Ako radimo s velikim brojem klijenata, teško ćemo pratiti što smo sve implementirali i što njihova infrastruktura zahtijeva. Implementacija današnjih aplikacija i sustava zahtijeva što veću brzinu, a vrijeme između pokretanja izgradnje i postavljanja okruženja postaje sve kraće i kraće. Kako bi se DevOps zadaci skalirali, potrebni su nam alati koji nam pružaju ponovljive, skalabilne i transparentne načine upravljanja našom infrastrukturom u oblaku. Osim postavljanja infrastrukture izrazito često problem su migracije između pružatelja usluga u oblaku. Uspostavljanje novog okruženja kroz IaC sastoji se od kopiranja datoteka varijabli, njezina podešavanja i postavljanja. Ovo je u velikoj suprotnosti s tradicionalnim postupkom: prolaziti kroz konzolu, odabirati opcije i nadati se da niste ništa propustili dok konfigurirate mnoštvo resursa u oblaku s mnoštvom potencijalnih opcija konfiguracije.

6.4. Implementacija infrastrukture u obliku koda

Izgradnja današnjih aplikacije zahtijeva izrazito agilna okruženja. Danas sve češće tvrtke koje imaju vlastita okruženja i poslužiteljske sale shvaćaju koliko je skup proces održavanja i zahtijeva posvećene timove. Posvećeni timovi u procesu pripreme okruženja za razvojne timove često stvaraju pogreške i kvarove jer ne postoji čista slika okruženja na samom početku izgradnje aplikacija. Spomenuti problemi potaknuli su DevOps inženjere na razmišljanje kako izgraditi infrastrukturu za aplikaciju ovisno o potrebama, a još bolje bi bilo ako se to može napraviti u procesu CI-a/CD-a. U 2009. godini tada izrazito poznati alat za održavanje CI-a/CD-a pod nazivom „Chef" došao je do rješenja razvijanjem alata koji će kroz programski kôd izgraditi sve resurse koje aplikacija zahtijeva. Bitno je naglasiti da izgradnja resursa mora biti u oblaku kod određenog pružatelja usluga koji omogućuje podizanje resursa pomoću programskih sučelja aplikacije. To su zapravo bili začeci Infrastrukture kao kôd (IaC), odnosno automatiziranog pružanja infrastrukture, omogućivanja implementacije i skaliranje aplikacija u oblaku većom brzinom, s manjim rizikom i smanjenim troškovima. Takav pristup razvojnim inženjerima omogućuje nesmetano podizanje ili održavanja resursa u podatkovnim centrima putem definicijskih datoteka, bez da se moraju uvjeriti posjeduje li tvrtka resurse i infrastrukturu za zahtjeve koji njima trebaju. Jednostavnim riječima, IaC će omogućiti proces zamjene ručnog rada potrebnog za upravljanje resursima i pružanja usluga jednostavnim linijama kôda u oblaku.

https://ec2.amazonaws.com/?Action=RunInstances **&ImageId=ami-31814f58&InstanceType=m4.large&MaxCount=2&MinCount=1&KeyName=my-key-pair&SubnetId=subnet-b2a249da&PARAMETRI_AUTORIZACIJE**

Kada pogledamo proces primjerice kreiranja instance, prije toga morali bismo uvijek napraviti provjere, postoji li ta instanca, tražimo li instancu po određenom nazivu, tagu, identifikaciji. Sve opisano izrazito je mukotrpan posao. S vremenom su pružatelji usluga počeli biti sve svjesniji kako je korisnicima izrazito teško stvarati skripte koje će održavati infrastrukturu i napravljeni su razni kompleti za razvoj softvera (engl. Software Development Kit, skraćeno SDK). Alati za razvoj softvera skup su softverskih alata i programa koje osiguravaju pružatelji usluga, a koje razvojni inženjeri mogu koristiti za izradu aplikacija za određene platforme. Pružatelji usluga stavljaju svo-

je SDK-ove na raspolaganje kako bi pomogli razvojnim inženjerima da jednostavno integriraju svoje aplikacije sa svojim uslugama. Pružatelj usluge izdaje SDK-ove za različite programske jezike, omogućujući da koristimo naš željeni jezik. Važno je znati da se značajke mogu razlikovati među izdanjima na različitim programskim jezicima. Zbog toga je bitno razlikovati SDK-ove, kako bi znali koji koristiti. Primjerice, direktne pozive prema pružatelju neovisno o programskom jeziku mogli smo raditi putem programskih sučelja, dok nas SDK usmjerava na korištenje specifičnog programskog jezika. Na prednjem dijelu SDK pruža sučelje kroz funkcije, metode i klase. Ovo pojednostavljuje zadatak pisanja koda od nule jer je većina posla već obavljena. Općenito govoreći, sučelje omogućuje GET, PUT i POST operacije. Logiku korištenja i manipulacije trebamo sami implementirati. Takav pristup pruža razinu razdvajanja koda poslovne logike i koda pozadinskog sustava. Ključna prednost jest to što je kod čitanja poslovne logike izrazito jasniji. Pristup kroz programska sučelja izrazito je kompleksan i tvrtke ga pokušavaju izbjeći. Problem jest to što je izrazito kompleksno pratiti status infrastrukture i zahtijevat će od DevOps inženjera implementaciju logike koja će provjeravati svaki detalj stanja. Ovakav pristup često rezultira pogreškama i izrazito je težak za održavanje.

6.5. Deklarativno upravljanje

Danas je na tržištu mnoštvo alata koji omogućuju pisanje infrastrukture kao kôd, pri čemu je jedan od najpoznatijih i najkorištenijih Terraform. Terraform je klijentska aplikacija koja zahtijeva instalaciju na računalo i koja je neovisna o drugim alatima, a koristi se u svrhu pisanja IaC infrastrukture. Terraform je deklarativna tehnologija koja ima mogućnost proširenja s dodacima koji omogućuju da Terraform radi po principima imperativnog pristupa. To znači da Terraform može raditi u oba stanja, ali primaran je deklarativan pristup. Kada kažemo da posjeduje imperativni dio, pod tim mislimo da će pohranjivati sva stanja koja smo napravili jedino putem Terraform skripti. Ako se dogodi ručna izmjena na infrastrukturi, Terraform neće biti svjestan toga. U praksi ćemo čuti da je Terraform imperativan pristup ograničen deklarativnim HCL kodom. HCL (punog naziva *Hashicorp Configuration Language*) jezik je koji je nalik Jenkins fileu koji smo spominjali u poglavlju CI/CD. HCL je sintaksa pomoću koje gradimo Terraform skripte. Sintaksno, jezik je izrazito jednostavan jer posjeduje definiranu strukturu koja se mijenja ovisno o potrebi koju moramo napraviti.

```
terraform_naredba "naziv_resursa_pružatelja_usluge_u_oblaku" "naziv_
resursa" {
   "opcija": "izmjena",
}
```

terraform_naredba – umjesto ove varijable navodimo što želimo napraviti, često u ovom odjeljku navodimo „resource” jer dohvaćamo postojeći ili dodajemo novi resurs.

naziv_resursa_pružatelja_usluge_u_oblaku – umjesto ove varijable navodimo vrstu resursa koju želite stvoriti ili izmijeniti, to može biti EC2 instanca, S3 spremnik, baza podataka i slično. Ove nazive resursa daje pružatelj usluga u oblaku. Ako ste konfigurirali AWS, moći ćete koristiti AWS resurse kao ovaj argument. Ako koristite GCP, moći ćete raditi s GCP resursima. Resursi su raspoređeni iz imenskog prostora pružatelja usluga.

naziv_resursa – umjesto ove varijable navodimo naziv resursa koji stvaramo. Naziv je važan jer se koristi kasnije prilikom referenciranja vašeg resursa. Ime uglavnom ovisi o vama.

opcija – umjesto ove varijable navodimo opcije koje želimo postaviti nad resursom, a ovaj dio ponovno definira pružatelj usluge.

Nakon shvaćanja kako Terraform funkcionira dolazimo do trenutka kada smo svjesni da postoji potreba za implementacijom infrastrukture kao kôd. U tom trenutku shvatit ćemo da već postoji napravljena infrastruktura i nećemo moći raditi sve iz početka. Morat ćemo razumjeti kako funkcionira trenutna infrastruktura i koji resursi postoje. Nakon toga krenut ćemo u proces implementacije korak po korak. Budući da se radi o klijentskoj aplikaciji, Terraform mora pratiti resurse koje stvara. I to čini kroz koncept stanja, koji možemo zamisliti kao bazu podataka. Svaki put kada ažuriramo svoju infrastrukturu, Terraform ažurira našu datoteku stanja. Datoteka stanja je JSON datoteka koja navodi naziv, svojstva i ovisnosti resursa. Terraform stanje prema zadanim postavkama pohranjuje lokalno, aplikacija stvara datoteku pod nazivom terraform.tfstate. Ali možemo koristiti i ono što Terraform naziva *backendom* i pohraniti datoteku stanja na udaljeni poslužitelj. Na taj način podijelit ćete stanje infrastrukture sa svim korisnicima koji koriste vaš kôd.

Sada želimo pokrenuti Terraform skriptu i napraviti izmjene. Dolazimo do procesa koji Terraform naziva plan i primjena. Korak koji ćete svaki put primjenjivati kada želite napraviti izmjene. Naredba: **`terraform plan`** usporedit će kôd koji smo napisali s našim stanjem kako bi provjerila ima li promjena. Proces koji se mora redovito odraditi kako bi Terraform bio svjestan toga što on zapravo treba napraviti i u tom procesu ispisat će **deltu**, koja vam pokazuje koji će resursi biti uništeni (označeni oznakom: **–**), koji će biti dodani (označeni oznakom: **+**) i koji će se ažurirati (označeni oznakom: ~). Kada ste zadovoljni promjenama koje **`terraform plan`** ponudi, možete pokrenuti **`terraform apply`**, što će zatim izvršiti te promjene.

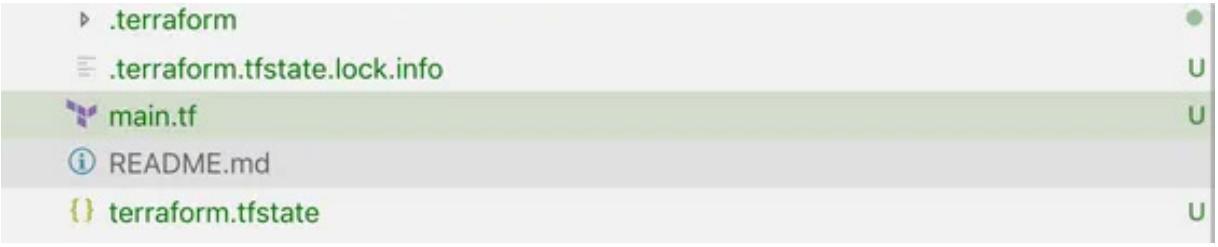
Prilikom pokretanje naredbe za primjenu izmjena ljudi se često pitaju što je s ovisnostima? Kada gradimo infrastrukturu, često stvaramo puno resursa u isto vrijeme, koji svi ovise jedni o drugima. Jedinstvena i sjajna značajka Terraforma jest to da uspijeva utvrditi koji od vaših resursa ovise jedni o drugima i stoga kako napraviti plan izvršenja koji će ažurirati resurse ispravnim redoslijedom. Kako bi se osiguralo stvaranje resursa uzastopno, Terraform stvara grafikon, koji se može vidjeti ako pokrenete naredbu **`terraform graph`**.

```
digraph {
  compound = "true"
  newrank = "true"
  subgraph "root" {
  "[root] aws_53_bucket.your_new_bucket" [label ="aws_53_bucket.your_
new_bucket", shape = "box"]
   "[root] provider.aws" [label = "provider.aws", shape = "diamond"]
   "[root] aws_53_bucket.your_new_bucket" -> "[root] provider.aws"
    "[root] meta.count-boundary (count boundary fixup)" -> "[root]
aws_53_bucket.your_new_bucket"
    "[root] provider.aws (close)" -> "[root] aws_s3_bucket.your_new_
bucket"
   "[root] root" -> "[root] meta.count-boundary (count boundary fixup)"
   "[root] root" -> "[root] provider.aws (close)"
  }
}
```

Ako sami želite djelovati na redoslijed, morat ćete koristiti koncept koji se zove interpolacija. Tipična interpolacija izgleda ovako:

```
terraform_naredba "naziv_resursa_pružatelja_usluge_u_oblaku" "naziv_
resursa" {
   "opcija": "izmjena",
   depends_on = ${aws_instance.master_instance.id}
}
```

Za uspješno izvršavanje Terraform skripte moramo biti svjesni izgleda strukture naše mape.



Slika 6.5.1. Izgled strukture Terraform skripte

Kada pokrenemo naredbu kao što je **`terraform plan`**, Terraform će prema zadanim postavkama tražiti samo datoteke iz trenutne mape. Terraform će uzeti bilo koju datoteku s ekstenzijom **`.tf`** i pokrenuti naredbu koju smo naveli. Zbog „magije" ovisnosti o kojoj smo prethodno govorili Terraformu nije bitan redoslijed učitavanja datoteka. Kako bismo osigurali ulaznu točku za naš Terraform, uobičajena je konvencija imati datoteku **`main.tf`**. Međutim, budući da Terraform uvozi sve tf datoteke, ne moramo se pridržavati ovog standarda. A što je s imperativnim pristupom? Kada razvojni inženjeri rade s alatom Terraform, a potreban im je imperativan pristup, tada vrlo često kombiniraju Terraform s alatom kao što je Ansible. Ansible koristi YAML sintaksu za definiranje procedure koju treba izvesti na ciljnoj infrastrukturi. Ansible YAML skripte proceduralne su prirode – što znači da kada pišete skriptu, ona će se izvršavati od vrha prema dolje. Ansible skripte nazivaju se „ansible playbooks". Kada moramo izvršiti određeni niz zadataka, to definiramo u priručniku. Pa zadatke provjere postoji li određeni resurs, odnosno instanca u pružatelju usluga, radili bismo putem Ansible skripti. Na taj način nadogradili bismo postojeće Terraform skripte određenim uvjeti-

ma koje bismo pisali unutar Ansiblea i tada će Ansible pokrenuti Terraform. Možemo zaključiti da u odnosu na ručnu konfiguraciju imamo jako puno prednosti u radu s infrastrukturom kao kôd.

6.6. Brzina, konzistentnost i odgovornost

Snaga infrastrukture kao koda proizlazi iz agilnosti. Primjerice, ne možemo kopirati i zalijepiti poslužitelje, baze podataka i slične resurse kroz grafičko sučelje pružatelja usluge. Pružatelji usluga prije samog IaC-a ponudili su razne verzije SDK-ova koje su omogućile upravljanje resursa samo za njihove usluge, ali to nije rješenje s obzirom na to da bismo to radili kroz određene sigurnosne kopije i ponovna podizanja resursa. S IaC-om možete iskoristiti module otvorenog koda. Ovo značajno smanjuje količinu HCL koda za pokretanje novih komponenti u infrastrukturi. Što se tiče primjene promjena, kao i drugi alati za upravljanje konfiguracijom Terraform je sposoban koristiti višedretvenost za istovremeno postavljanje infrastrukture, čime se smanjuje vrijeme potrebno za stvarno postavljanje promjena. Sintaksa je također mnogo deklarativnija umjesto jednog divovskog JSON ili YAML bloba s kojim je teško ili glomazno raditi.

Promjenjivost konfiguracije u više okruženja može biti noćna mora za DevOpse i razvojne inženjere. Često se javljaju pitanja poput: „Zašto ovo dobro funkcionira na papiru, ali ne i u produkciji?" Kada koristite alate za verzioniranje IaC koda za uvođenje promjena u svoje upravljane resurse, možete vidjeti „plan" i „rezultate" svojih promjena prije primjene. Ove informacije možete koristiti kasnije, ako se pojave bilo kakvi problemi.

U modernom poslovanju riječ odgovornost može značiti puno stvari. Budući da se IaC kôd nalazi u kontroli izvora, tom kodu možemo vjerovati kao jedinom izvoru istine i na taj način možemo pratiti tko je napravio određenu promjenu na određenoj komponenti infrastrukture. Ovo potiče bolju komunikaciju, budući da je u većini vremena bitno znati tko je napravio koju promjenu, tako da možemo i razumjeti zašto.

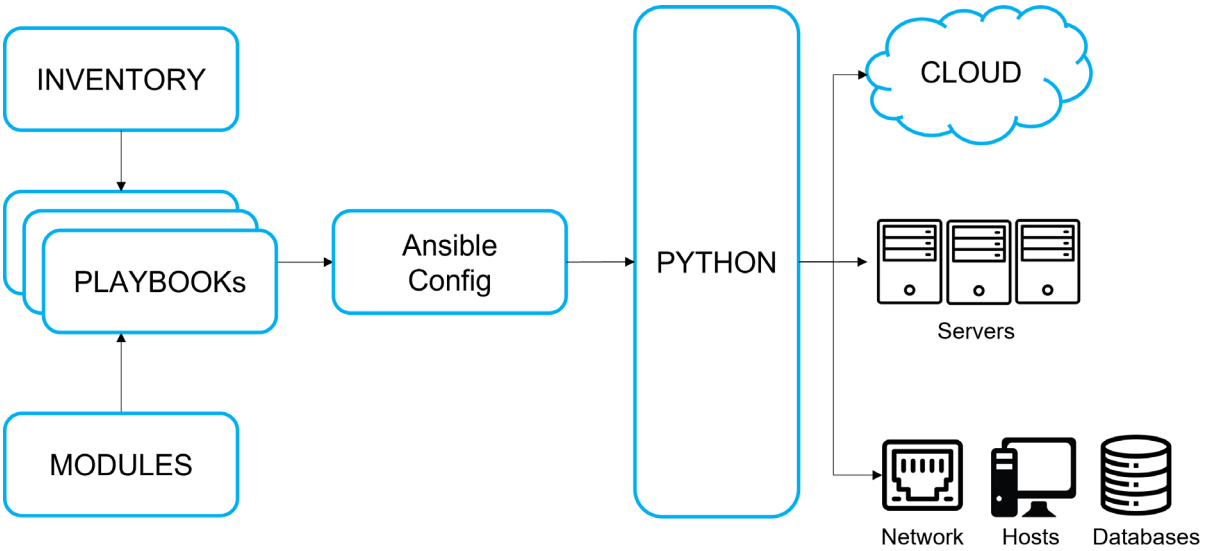
6.7. Imperativno upravljanje

Ansible je alat za automatizaciju bez agenta koji omogućuje brzo i jednostavno upravljanje poslužitelja koji se nalaze u oblaku. Dizajniran za višeslojnu implementaciju, Ansible brine o upravljanju konfiguracijom, implementaciji aplikacija, orkestraciji unutar usluge i ostalim IT potrebama. Koristeći Ansible, možemo modelirati svoju infrastrukturu u oblaku jednostavnom sintaksom. Ne zahtijeva agente ni dodatno prilagođenu sigurnosnu infrastrukturu, već se sve izvršava korištenjem pouzdanog SSH protokola. Sintaksa od nas zahtijeva definiranje zadataka koji mogu biti nešto jednostavno poput definiranja paketa koje treba instalirati, deklariranja postavki web-poslužitelja ili čak postavljanja potpune orkestracije. Ansible u svom sklopu zahtijeva rad s modulima. Ansible moduli napisani su za modeliranje željenog stanja resursa sustava. Primjerice, ako želimo definirati ponašanje web-poslužitelja, potrebno bi bilo instalirati modul koji u sebi sadržava set biblioteka koje znaju raditi s tim web-poslužiteljem. Ansible održava veliku biblioteku zajedničkih modula koji se mogu pobrinuti za većinu opskrbe popularnim softverom. Međutim, korisnici su više nego dobrodošli da po potrebi napišu svoje vlastite module, a vaša vlastita biblioteka modula može se nalaziti na bilo kojem računalu. Nadovezujući se na modularnost i pisanje modula, Ansible omogućuje definiranje zadataka i procesa u lako čitljivim konfiguracijskim datotekama koje se nazivaju Playbooks. Najjednostavnije rečeno, Ansible Playbooks opisuju poslove automatizacije pomoću YAML-a. Jednostavan jezik označavanja čini konfiguracije jednostavnima za pisanje na engleskom jeziku bez složene sintakse ili posebnih značajki. Smisao jednostavnosti jezika jest da omogući vraćanje na stari kôd čak i godinama kasnije i da ga razumijemo. Bitno je razumjeti da je Playbook datoteka unutar koje definiramo module po zadacima. Također, unutar Playbooka moramo definirati popis iliti inventar poslužitelja na kojem će se izvršavati Playbook. Zaključak funkcioniranja Ansiblea jest da ako su Ansible moduli alati u vašoj radionici, playbooks su vaši priručnici s uputama, a vaš inventar poslužitelja vaša je sirovina. Sama prednost Ansiblea jest to što je dostupan na svim operativnim sustavima jer je potreban Python za pokretanje skripta.

U usporedbi Ansible i Terraforma možemo primijetiti da Ansible zapravo ima više funkcionalnosti od same implementacije IaC-a. Ansible omogućuje pisanje skripta koje opisuju stanje vašeg sustava. To znači da, jednom napisane, te su skripte višekratno upotrebljive i pouzdane. Kada pričamo o stanju sustava, tada ne razmišljamo samo

o definiranju resursa koji moraju postojati, već i o paketima i ostalim ovisnostima koje taj resurs mora imati. Ansible skripte napisane su u YAML-u. Ansible koristi iste alate koji se inače koriste za administraciju u Linux sustavima. Ansible vam daje moć upravljanja višeslojnim implementacijama. Kroz Ansible zapravo možete postići paralelne implementacije, no bitno je imati na umu da Ansible ne razumije stanja sustava. Ansible je imperativan pristup unutar kojeg moramo definirati granja koja bi glasila: ako već postoji resurs, preskoči taj zadatak unutar Playbooka ili neka zadatak dohvati informacije o resursu kako bismo nešto dalje napravili.

Budući da Ansible nije samo za upravljanje IaC-om, ako bismo željeli održavati infrastrukturu potrebno je implementirati uzorak koji se naziva Ansible toranj (engl. *Ansible Tower*). Ansible toranj omogućuje centralizaciju i kontrolu cijele IT infrastrukture pomoću vizualne nadzorne ploče. Prikazuje sve što se događa u vašem Ansible okruženju u stvarnom vremenu. Značajke, uključujući kontrolu pristupa utemeljenu na ulogama, raspoređivanje poslova, praćenje inventara preko više pružatelja usluga u oblaku, mogu se jednostavno organizirati i automatizirati.



Slika 6.7.1. Prikaz infrastrukture Ansible tornja

Iako se čini da je ovaj pristup dosta modularan i agilan, nije česta praksa koristiti Ansible za upravljanje infrastrukturom. Terraform u odnosu na Ansible posjeduje praćenja stanja infrastrukture, timovi lakše surađuju i prate dešavanja.

6.8. Upravljanje konfiguracijom nasuprot orkestraciji

Terraform i Ansible imaju toliko sličnosti i razlika u isto vrijeme. Razlika dolazi kada pogledamo dva značajna koncepta DevOpsa: orkestracija i upravljanje konfiguracijom.

Alati za upravljanje konfiguracijom rješavaju probleme lokalno, a ne zamjenjuju sustav u cijelosti. Ansible pomaže konfigurirati svaku radnju i instrument te osigurava glatko funkcioniranje bez oštećenja ili pogreške. Osim toga, Ansible dolazi s hibridnim mogućnostima za izvođenje i orkestraciju i zamjenu infrastrukture. Alati za orkestraciju osiguravaju da okruženje bude kontinuirano u željenom stanju. Terraform je izričito dizajniran za pohranjivanje stanja domene. Kad god postoji bilo kakva greška u sustavu, Terraform automatski vraća i izračunava cijeli proces u sustavu nakon ponovnog učitavanja. Najbolje odgovara situacijama u kojima je potrebno stalno i nepromjenjivo stanje. Terraform Apply pomaže u učinkovitom rješavanju svih anomalija.

Zaključak je da će se zapravo eksperti složiti da je deklarativan pristup najbolji način za upravljanje infrastrukturom, odnosno definiranje IaC-a, dok će imperativan pristup više gledati na način da nakon što deklarativno nešto definiramo, imperativnim pristupom odradit ćemo dodatnu konfiguraciju sustava.

6.9. Promjenjivost protiv nepromjenjivosti infrastrukture

Tijek rada za implementaciju aplikacije uključuje pružanje infrastrukture, instaliranje ispravne verzije izvornog kôda i instaliranje svih ovisnosti. Infrastruktura koja služi kao temelj za kasnije verzije aplikacija i usluga ima svojstvo poznato kao promjenjivost. Ili se postojeća infrastruktura koristi za implementaciju ili možemo stvoriti potpuno novu infrastrukturu za nju. O postupcima postavljanja ovisi je li infrastruktura promjenjiva ili nepromjenjiva. Kaže se da je promjenjiva kada se sljedeće verzije aplikacija objavljuju na istoj infrastrukturi. S druge strane, kaže se da je nepromjenjiva ako se implementacija odvija tijekom izdanja na potpuno novoj infrastrukturi. Iako se promjenjivost čini zgodnom, postoji veća vjerojatnost neuspjeha i pogreške. Prethodna verzija najprije se mora deinstalirati prije nego što se željena verzija može instalirati kada se konfiguracije aplikacije ponovno primjenjuju na istoj infrastrukturi. Dodatni koraci pov-

ećavaju vjerojatnost neuspjeha. To može dovesti do nedosljednih postavki i nepredvidivog ponašanja u klasteru poslužitelja. Umjesto toga, ako se usredotočimo na minimiziranje ovih koraka preskakanjem postupka deinstalacije i izvođenjem instalacije na svježim infrastrukturnim resursima, imat ćemo priliku testirati novu implementaciju i vratiti je unatrag ako ne radi. Ovaj način tretiranja infrastrukture kao nepromjenjive daje administratorima više kontuloge nad unošenjem promjena. Deklarativnim pristupom definirat ćemo ponovno podizanje konfiguriranih instanca, dok će balanser opterećenja određivati gdje slati korisnike, tako da korisnici ne dobiju dojam da aplikacija trenutno ne funkcionira. Imperativnim pristupom imamo veću šansu da nešto možda i neće raditi kako treba, jer može se dogoditi bezbroj iznimka.

6.10. Upravitelj stanja

Upravljanje punim životnim ciklusom resursa s Terraformom izrazito je jednostavno jer Terraform generira datoteku stanja unutar koje se zapisuje trenutno stanje. Održava mapiranje infrastrukturnih resursa datoteka stanja s najnovijim konfiguracijama. Upravljanje stanjem ključno je za rad Terraforma. Stanja se koriste za praćenje promjena konfiguracije i pružanje istih. Dodatno, moguće je uvesti već postojeće resurse kojima upravlja Terraform dovođenjem datoteka stanja iz infrastrukture stvarnog svijeta. U bilo kojem trenutku moguće je izvršiti upit u datotekama stanja Terraform kako bismo saznali više o komponentama infrastrukture i njihovim dostupnim karakteristikama. Ansible, nasuprot tome, ne pruža nikakav oblik upravljanja životnim ciklusom infrastrukture. Sve promjene konfiguracije automatski se implementiraju na ciljni resurs jer se Ansible fokusira na upravljanje konfiguracijom i prema zadanim postavkama pretpostavlja nepromjenjivu infrastrukturu.

6.11. Što odabrati

Terraform se bavi automatizacijom infrastrukture. Njegovom trenutnom deklarativnom modelu nedostaju neke značajke koje stvaraju složenost. Koristeći Terraform, elementi potrebnih okruženja zasebno se opisuju, uključujući njihove odnose. Procjenjuje model, stvara plan na temelju ovisnosti i daje optimizirane naredbe za infrastrukturu

kao uslugu. Ako nema promjene u okruženju ili strategiji, ponovljena izvođenja neće učiniti ništa. Ako postoji bilo kakvo ažuriranje u planu ili okruženju, ono će sinkronizirati infrastrukturu oblaka. Terraform dolazi s dobrim mogućnostima zakazivanja i vrlo je jednostavan za korištenje. Dobro se integrira s Dockerom jer se Docker malo bolje nosi s upravljanjem konfiguracijom nego Terraform. Ali nedostaju jasni zapisi o tome kako se ciljni containeri dovode u konačno stanje, a ponekad je konačna konfiguracija nepotrebna.

Ansible slijedi proceduralni, odnosno imperativan pristup. Različiti korisnici stvaraju priručnike koji se ocjenjuju pristupom od vrha do dna i izvršavaju u nizu ili paralelno. Playbooks je odgovoran za konfiguraciju mrežnih uređaja, što pridonosi proceduralnom pristupu. Naravno, Ansible osigurava i infrastrukturu u oblaku. Ali njegov proceduralni pristup ograničava ga na velike implementacije infrastrukture. Ansible dolazi s boljom sigurnošću i funkcionalnošću. Smatra se zrelim alatom jer se lako prilagođava tradicionalnim okvirima automatizacije. Nudi jednostavne operacije i pomaže u brzom pisanju kôda. No s druge strane, nije dobar u uslugama poput logičkih ovisnosti, usluga orkestracije i međusobno povezanih aplikacija.

Odabir i odluku o korištenju alata donijet ćete u skladu sa zahtjevima situacije i posla. Na primjer, ako se *containersko* rješenje koristi za pružanje softvera unutar oblaka, onda je Terraform poželjan. S druge strane, ako želite dobiti razumnu kontrolu nad svojim uređajima i pronaći druge načine za implementaciju temeljnih usluga, Ansible je prikladniji. Bitno je razumjeti da su ovi alati još u razvoju i ovi će alati u budućnosti pružiti sveobuhvatnija rješenja. S vremenom će se vrlo vjerojatno dogoditi promjene vrsta pristupa rješavanju problema.

6.12. Implementacija infrastrukture kao kôd na postojeću infrastrukturu

Implementacija na postojeću infrastrukturu izrazito ovisi o deklarativnom ili imperativnom pristupu. Imperativan pristup od nas će zahtijevati pisanje kôda, pri čemu ćemo morati postavljati razne upite programskom sučelju. Imperativan pristup zahtijevat će odrađivanje iznimka jer je izrazito teško pratiti resurse kod pružatelja usluga. Izrazito

je teško pratiti koje su izmjene napravljene, stoga DevOps inženjeri stvaraju različita ograničenja. Recimo, ako koristimo imperativan pristup, morat ćemo ograničiti razvojnim inženjerima prava kod pružatelja usluga. Pa tako kada jednom preslikamo postojeću infrastrukturu, imperativnim pristupom želimo se ograničiti na to da ne dolazi do izmjena koje smo obuhvatili iznimkama. Cijeli opisani proces izrazito često može eskalirati, pa ono što je jednostavno postane komplicirano. Do sada smo koristili Terraform i shvatili smo da je izrazito deklarativan, no Terraform posjeduje izrazito snažnu funkcionalnost koja omogućuje izlazak iz okvira imperativnog pristupa. Terraform posjeduje naredbu: **`terraform import`**. Terraform CLI naredba koristi se za čitanje već postavljane infrastrukture i ažuriranje stanja.

Postupak uvoza postojeće infrastrukture izrazito je zbunjujući jer Terraform neće sam generirati datoteku stanja, već smo mi zaduženi za taj postupak. Primjer uvoza postojeće infrastrukture bazirat ćemo na AWS-ovim uslugama. U procesu uvoza moramo sami ručno izraditi odgovarajuću konfiguracijsku datoteku za primjerice EC2 instancu. U glavnu main.tf datoteku morat ćemo definirati EC2 konfiguraciju baš kao što smo definirali konfiguraciju prilikom kreiranja resursa. Jedina je razlika to što ćemo morati dodati atribut ami, koji predstavlja jedinstvenu identifikaciju slike operativnog sustava i instance_type kako bi Terraform bio siguran da je dobra jačina instance postavljena.

```
resource "aws_instance" "naziv_resursa" {
 ami = "indetifikacijska_oznaka_instance"
 instance_type = "veličina_instance"
}
```

Zamislite to kao da su resurs u oblaku (EC2 instanca) i njegova odgovarajuća konfiguracija dostupni u našim datotekama. Sve što je preostalo učiniti jest preslikati to dvoje u našu datoteku stanja. To činimo pokretanjem naredbe import na sljedeći način:

```
terraform import aws_instance.<Naziv_Resursa> <Identifikacija_Instance>
```

Naredba preslikava konfiguraciju **`aws_instance.<Naziv_Resursa>`** na EC2 instan-

cu pomoću jedinstvene identifikacije. Pod preslikavanjem mislimo na to da datoteka stanja sada „zna" za postojanje EC2 instance s danom identifikacijom. Datoteka stanja također sadrži informacije o svakom atributu koji je dodijeljen EC2 instanci. Bitno je razumjeti da svaki resurs kod svakog ponuditelja usluge mora imati jedinstveni identifikator. Jedinstveni identifikator kreira se prilikom stvaranja usluge i dostupan je u obavijesti prilikom procesa kreiranja usluge. Identifikator usluge također možemo pronaći odmah pokraj našeg imenovanja resursa.

6.13. Implementacija infrastrukture kao kôd na različita okruženja

Vrlo se često postavlja pitanje što ako određeni dio infrastrukture želimo imati na specifičnom pružatelju usluga, a ostatak na drugom pružatelju usluga. Ili što ako želimo napraviti migraciju s jednog pružatelja usluge na drugi? To ćemo postići u Terraformu modulom „providers". „Providers", odnosno davatelji, imaju komplicirano zvučno ime za jednostavan koncept. Davatelji su dodaci koje su napravile treće strane, a koji djeluje kao most između Terraforma i bilo kojih davatelja usluga, kao što su AWS, GCP, Azure i drugi. Za konfiguriranje pružatelja trebamo definirati blok kôda koji izgleda:

```
provider "aws" {
    region = "eu-central-1"
}
```

Imajte na umu da ovaj put ipak koristimo naredbu **„provider"**, a ne naredbu **„resource"**. I umjesto vrste resursa navodimo vrstu našeg pružatelja usluga. Bitno je znati da nakon što definiramo sve pružatelje usluga unutar Terraform datoteke, moramo još promijeniti i **naziv_resursa_pružatelja_usluge_u_oblaku**. Nakon toga spremni smo agilno mijenjati infrastrukturu ovisno o pružatelju usluga.

6.14. Infrastruktura kao kôd unutar CI-a/CD-a

Na prvi pogled čini se da infrastruktura kao kôd (IaC) i kontinuirana integracija / kontinuirana implementacija (CI/CD) ne idu zajedno. Oni predstavljaju dva potpuno različita procesa, zar ne? Jedan je mehanizam za definiranje načina na koji konfigurirate i gradite svoj hardver, drugi je za izgradnju, testiranje i implementaciju vašeg softvera. Ako spojimo CI/CD i IaC alate, dobit ćemo maksimalnu optimizaciju vremena i novaca. Možemo utjecati na osiguravanje novih resursa na zahtjev kao dio implementacije. To bi značilo da ne moramo pripremati okruženje na kojem će se izvršavati testovi prije pokretanja linija i naravno nakon izvršavanja testova možemo automatski obrisati resurse. Na taj način izbjegli smo tradicionalno podešavanje okruženja koje radi i troši novac bez obzira na to koristimo li to okruženje ili ne. Često se odlazi korak dalje, pa nakon što se promjene temeljito testiraju, zapakiraju se unutar verzioniranog artefakta i učine se dostupnima za kasnije linije za korištenje i implementaciju infrastrukture kao kôd.

Korištenjem infrastrukture kao kôd dobit ćemo mogućnost automatskog skaliranja i povećanja poslužiteljskih resursa u trenutku. To će nam omogućiti uštedu vremena i smanjenje financijskih izdataka s obzirom na to da ne moramo brinuti o opskrbljivanju hardvera. Na taj način isto tako smanjujemo održavanje i brigu o sigurnosnim procesima s obzirom na to da cijeli taj proces preuzima pružatelj usluga u oblaku.

PITANJA ZA PONAVLJANJE:

1. Koja je razlika u pisanju deklarativnih i imperativnih definicija infrastrukture kao kôd?
2. Što je konfiguracijski pomak?
3. Što je idempotencija?
4. Što je Terraform?
5. Što je stanje infrastrukture kao kôd?
6. Što je Terraform plan?
7. Što je Ansible?
8. Objasnite razliku između Terraforma i Ansiblea.

6.15. Sinkronizacija izmjena nad infrastrukturom

S razvojem infrastrukture kao kôd dolazi do pitanja hoćemo li pohranjivati kôd u udaljeni repozitorij? Hoćemo li pohranjivati konfiguracije datoteke unutar već postojećeg repozitorija s kôdom ili je potrebno napraviti poseban repozitorij? Ova pitanja javljaju se jer je izrazito zbunjujuće smatrati IaC datoteke softverskim kôdom ili konfiguracijskim datotekama. Moramo razumjeti da s razvojem infrastrukture kao kôd tim će sve više iskorištavati konfiguracije datoteke tako da je izrazito bitno što prije razmišljati o pohranjivanju kôda na udaljeni repozitorij. Iako je diskutabilno što se smatra programiranjem, ostanimo samo na izvornom pitanju trebamo li pohranjivati IaC datoteke u zaseban repozitorij ili bi ih trebalo pohraniti uz kôd napisan za poslovnu funkcionalnost. Iako se čini da je potrebno više posla za održavanje zasebnih repozitorija za IaC datoteke i kôd za poslovne funkcije, ovakav pristup dolazi s puno prednosti. Na primjer, s vremenom biste otkrili da je broj promjena kôda potrebnih za održavanje ili ažuriranje infrastrukture dosta manji od broja promjena kôda potrebnih za održavanje ili ažuriranje poslovne funkcionalnosti. Njihovo pohranjivanje na isto mjesto konfiguracije datoteke čini sklonijima svim neželjenim ili nenamjernim promjenama. Ovo se izrazito često veže i s CI dijelom, gdje ne bismo željeli implementirati novu infrastrukturu svaki put kada se dogodi izmjena u repozitoriju. Isto tako željeli bismo odvojiti odgovornosti te ne bismo željeli da razvojni inženjeri ili inženjeri osiguranja kvalitete utječu ili mijenjaju konfiguracijske datoteke. Točnije, ovaj korak trebali bi provjeravati i nadograđivati DevOps inženjeri prema zahtjevima ostatka tima. Stoga se uvijek smatra najboljom praksom pohraniti IaC datoteke (bilo da su Terraform, AzureARM, Cloudformation…) u njihov vlastiti repozitorij. Međutim, ovo je samo smjernica, a ne zakon. Slobodno slijedite ono što je najbolje za ispunjavanje vaših zahtjeva.

6.16. Git repozitorij za pohranu infrastrukture kao kôd

Razmišljati o pohrani kôda na udaljeni repozitorij. Glavna ideja IaC-a jest upravljanje – što je više moguće – svom vašom infrastrukturom pomoću kôda. Svaka promjena potrebna na poslužitelju, u aplikaciji ili konfiguraciji mora biti definirana u kôdu. Konfiguracijske datoteke mogu se pretvoriti u predloške kako bi se omogućila veća fleksibilnost i ponovna upotreba. Postavke specifične za aplikacije ili poslužitelje također moraju

biti kôdirane, obično u varijabilnim datotekama. Baš zbog toga želimo pohraniti naše konfiguracijske datoteke unutar udaljenog repozitorija i taj repozitorij moramo smatrati centralnom točkom istine. Pomoću toga možemo biti sigurni u slučaju određene ručne izmjene i promjena koje se dese, a stvore probleme, što moramo ispraviti. Na taj način svatko u timu može biti svjestan tijekom otklanjanja pogreške što je pogrešku stvorilo. Tijekom kreiranja automatizacije ključno je zapamtiti i idempotenciju: bez obzira na to koliko se puta kôd izvršio, uvijek bi trebao imati isti rezultat. Isti unos, isti rezultat. Na primjer, kada pišete dio kôda koji modificira datoteku, morate osigurati da će datoteka izgledati isto ako se isti kôd ponovno izvrši. Na taj način na lakši način upravljamo pogreškama koje se mogu desiti. Osim pohrane kôda i konfiguracijskih datoteka, IaC često uz svoj kôd posjeduje datoteke stanja. U prošloj cjelini spomenuli smo na primjeru Terraforma datoteku stanja. Datoteka stanja prati koja se konfiguracija resursa preslikava na koji resurs unutar pružatelja usluge. Stanje također pohranjuje metapodatke o tim resursima, kao što je redoslijed ovisnosti za stvaranje resursa. Datoteku stanja također možemo pohraniti na Git, što će nam uvelike olakšati da naš kolega također može izvršiti skriptu za podizanje resursa, no postavlja se pitanje hoće li to biti najbolje rješenje?

6.17. Sinkronizacija stanja s timom

U procesu redovite implementacije i korištenja infrastrukture kao kôd shvaćamo da radimo s timom koji također ima potrebe korištenja naših skripti. Budući da koristimo infrastrukturu kao kôd, potrebno je na određeno mjesto pohranjivati stanje infrastrukture i izmjene koje smo radili. Taj podatak trebat će svi pojedinci iz tima kako ne bi direktno djelovali na ono što smo mi napravili i time napravili štetu. Stanje infrastrukture unutar vlastite konfiguracije ima osjetljive podatke. Na primjeru Terraforma, generirane datoteke stanja sadržavaju ključeve za kriptiranje i razne lozinke infrastrukture unutar nje, koje nisu sigurno pohranjene. Pohranjivanje lozinki itd. u običnom tekstu u repozitoriju loša je praksa, svatko tko dobije pristup vašem repozitoriju može dobiti te osjetljive informacije. Ono što nam ovakav pristup također može donijeti, jest problem pogrešnog stanja. Kada se Terraform izvršava, izvršava se pomoću datoteke stanja. Terraformu je to centralna točka vjernosti, koja mu govori što je potrebno napraviti. Kada radimo s datotekom stanja koja je pohranjena na Git, riskiramo pokrenuti Terraform sa starom datotekom stanja. Na primjer, ako zaboravite povući najnovije

promjene, datoteka stanja na vašem računalu mogla bi se razlikovati od onih ostalih članova tima. Terraform *backend* konfiguracija je koju možete postaviti, a koja govori Terraformu gdje treba pohraniti vaše stanje. Uz backend postavljanje stanje se prenosi na udaljenu lokaciju i može mu pristupiti cijeli tim istovremeno. Međutim, istovremeno pokretanje na više računala Terraforma moglo bi dovesti do loših ishoda jer više računala pokušava manipulirati datotekom stanja u isto vrijeme. I zato većina *backend* konfiguracija također podržava „zaključavanje". Zaključavanjem možete osigurati da samo jedna osoba može pokrenuti izvršenja nad datotekom stanja odjednom. Prednost Terraform *backenda* jest to što je dostupna na različitim tehnologijama, kao što je AWS, Azure... Budući da svaki pružatelj usluga radi na drugačijoj implementaciji, postoje razlike, ali funkcionalnost ostaje ista. Možete konfigurirati *backend* s malim blokom kôda koji se dodaje bilo kojoj od vaših terraform .tf datoteka. Međutim, obično je pohranjen u datoteci koja se zove backend.tf ili jednostavno u vaš main.tf. I u tom slučaju označava da se naša datoteka stanja nalazi pohranjena u S3 za AWS ili u Azure Storage za AWS. Osim spomenutih načina postoje i specijalizirani servisi za pohranu samih stanja. Pa tako u našem primjeru Terraforma postoji servis s nazivom Terraform Cloud. Terraform Cloud platforma je koja izvodi Terraform pokretanja za pružanje infrastrukture, bilo na zahtjev ili kao odgovor na različite događaje. Za razliku od sustava kontinuirane integracije opće namjene (CI), on je duboko integriran s Terraformovim radnim procesima i podacima, što mu omogućuje da Terraform učini znatno praktičnijim i lakšim za korištenje. Pa ćete tako korištenjem specijaliziranih servisa dobiti mogućnost da uz minimalno održavanje dobijete maksimalnu fleksibilnost.

6.18. Dijeljenje stanja infrastrukture pomoću specijaliziranih servisa

Specijalizirani servisi za upravljanje infrastrukturom kao kôd omogućit će nam lakše upravljanje infrastrukturom jer su baš takvi servisi stvoreni kako bi imali sve funkcionalnosti koje su nam potrebne. Spomenuli smo pohranu stanja infrastrukture kod pružatelja usluge koji će nas svakako dodatno koštati i zahtijevat će dodatne napore održavanja. Velik broj alata za upravljanje IaC-om posjeduje vlastita specijalizirana rješenja koja uz to nude privatne repozitorije unutar kojih ćemo pohraniti sve konfiguracije datoteke i stanja infrastrukture. Osim olakšanog održavanja nudi nam mogućnost jednostavne integracije s git poslužiteljem te dijeljenja informacija s ostatkom tima. DevOps inženjerima često je zbunjujuće zašto koristiti specijalizirani servis, tj.

koje su nam dodane vrijednosti korištenja specijaliziranih servisa.

Specijalizirani servisi prije svega omogućuju brže i olakšano podešavanje okruženja. Primjerice, dosad ste pohranjivali veliki broj stvari na Git udaljeni repozitorij, na takav način na jednostavan način sa svim članovima dijelimo sve informacije. U konfiguracijama IaC-a postoje informacije koje mogu predstavljati sigurnosne rizike. Kako bismo izbjegli taj proces, specijalizirana rješenja olakšavaju održavanje i skrivanje podataka koje ne bismo trebali podijeliti s ostatkom organizacije. Uobičajena je praksa da specijalizirano rješenje generira API token i implementira višestruku autentifikaciju. Što će reći da na svaku izmjenu ili dohvaćanje sigurnosnih podataka moramo potvrditi naš identitet. U tradicionalnim načinima pohrane takvu mogućnost zapravo nemamo. Spominjali smo praćenje stanja, specijalizirana rješenja nude pregled detaljnije povijesti stanja, što nam omogućuje praćenje izmjena poput onih na udaljenim repozitorijima. Svaka promjena mora proći kroz verifikaciju ostalih članova tima. Dakle bolje upravljanje u slučaju nevaljanog stanja ili potrebe za preuzimanjem starije verzije. Ponekad se čini da je dovoljan Git, no svaku izmjenu potrebno je pohraniti na udaljeni repozitorij, dok specijalizirana rješenja imaju automatizirani proces. Specijalizirana rješenja također imaju bolji način zaključavanja izmjena nad infrastrukturom. Ako sami implementiramo proces zaključavanja za izmjene, taj proces može dovesti do čestih pogrešaka i nezaključanog stanja. Tako da u takvim slučaja ne bismo željeli pisati dugačke uvjete koji će nas ograničiti od promjena, već možemo olakšati proces zaključavanja kroz specijalizirane servise.

6.19. Integracija infrastrukture kao kôd s CI-jem

Lokalno pokretanje Terraforma općenito znači da su sve ovisnosti već na mjestu: imate Terraform instaliran i prisutan u korisničkom PATH-u, a pružatelji usluga već su pohranjeni u mapi .terraform. No u procesu prebacivanja izvođenja Terraforma s lokalnog računala na linije to nije baš najjednostavniji slučaj. Često se dešava da linije izvršavaju naredbe na prvoj dostupnoj instanci. Što će reći da se etape linija često izvršavaju na različitim instancama. Moguće je definirati etapu da se uvijek izvodi na jednakoj instanci, ali takve linije izrazito su dugotrajne. Česta je praksa izgradnja Docker slike koja će u sebi imati predefinirane postavke i podizanje Docker instance

s tom slikom izrazito će ubrzati proces i omogućiti kontrolu nad procesom. Takav proces olakšat će nam i čuvanje stanja jer možemo predefinirati udaljenu pohranu stanja ili možemo sliku stanja održavati lokalnom i svaki je put montirati kao Docker volumen. U praksi vidjet ćete da se može koristiti Docker slika tvrtke Hashicorp, ali ponekad je mudro održavati vlastite Docker slike jer možete dodati dodatne alate koji vam mogu zatrebati unutar Docker *containera*.

Prilikom generiranje Doker slika potrebno je imati na umu da glavne funkcionalnosti IaC-a, i također u našem slučaju Terraforma, pronalazimo kroz *providere*. Funkcionalnosti, odnosno knjižnice, svakim ćemo pokretanjem *containera* morati instalirati ponovno, a to će uzeti puno vremena jer svaka knjižica ima i više stotina megabajta. Na jednom pokretanju to se ne čini problem, no na više pokretanje dnevno to čini znatnu razliku. Postoje dva uobičajena načina za rješavanje tog problema: ili koristite zajedničku predmemoriju koja je dostupna vašim radnim opterećenjima linija ili pripremite binarne datoteke pružatelja u runtime okruženju (tj. pripremite datoteke tijekom izgradnje Docker slike). Kritični element oba pristupa konfiguracijska je putanja mape predmemorije dodatka. Prema zadanim postavkama Terraform traži dodatke i preuzima ih u direktoriju .terraform, koji je lokalan u odnosu na direktorij glavnog projekta. Ali to možete nadjačati korištenjem varijable okoline `TF_PLUGIN_CACHE_DIR` za preusmjeravanje lokacije pohrane dodataka. CI/CD alat podržava dijeljenje predmemorije, što omogućuje značajno smanjenje operativnog opterećenja jer u tom trenutku sva okruženja za izvođenje linija mogu koristiti sve dijeljene datoteke/podatke.

6.20. Korištenje Terraform alata

CI/CD linija općenito se izvodi u okruženjima bez stanja. Dakle, svako sljedeće pokretanje Terraforma izgleda kao novi početak, tako da je projekt potrebno inicijalizirati prije nego što se mogu izvesti druge radnje. Upotreba naredbe init u CI-u/CD-u malo se razlikuje od uobičajene lokalne upotrebe:

```
terraform init -input=false
```

Opcija **`-input=false`** sprečava Terraform CLI traženje korisničkih radnji (izbacit će pogrešku ako je unos bio potreban). Također, postoji opcija **`-no-color`**, koja spreča-

va korištenje kôdova boja u *shellu*, tako da će izlaz izgledati mnogo čišći ako CI/CD sustav za praćenje ne može prikazati formatiranje terminala. Ono što također DevOpsi vole jest utjecaj na *backend* konfiguraciju koju možemo uzeti iz cloud pružatelja usluga. Na to se nadovezuju i stanja infrastrukture. Ako želimo utjecati na taj ishod, potrebno je dodati zastavicu: **`-backend-config`**. Spomenuta opcija omogućuje nadjačavanje pozadinske konfiguracije u vašem kôdu ili je definirate ako više volite koristiti djelomičnu konfiguraciju, čime se stvaraju ujednačeniji *pipelieni*.

```
terraform init -input=false -backend-config="uloge_arn=arn:aws:iam::
<vaš_korisnički_identifikator>:uloge/DevOpsAutomation"
```

Međutim, potrebno je paziti na automatski generiranu datoteku s nazivom: **`.terraform.lock.hcl`** kao što je predložio HashiCorp (datoteka zaključavanja ovisnosti): na taj ćete način moći temeljitije kontrolirati ovisnosti bez brige o prijenosu ove datoteke između etapa izgradnje. Tako da je preporučeno pohraniti datoteku na lokaciju kojoj mogu pristupiti sve etape izgradnje.

6.21. Generiranje plana infrastrukture unutar etapa linija

Naredbu terraform plan spominjali smo u procesu generiranja infrastrukture, koja nam također pomaže potvrditi promjene. Spomenuta naredba obavezna je jer će nam na taj način terraform generirati datoteku s planom izmjena. U tom procesu morat ćemo ručno potvrditi slažemo li se s izmjenama. U tom procesu nažalost to je izrazito kompleksno promijeniti u procesu CI-a. Prema zadanim postavkama Terraform ispisuje izlaz plana u formatu prilagođenom ljudima, ali također podržava strojno čitljiv JSON. S dodatnim opcijama naredbenog retka možete proširiti svoje CI iskustvo. Na primjer, možete koristiti uvjete provjere valjanosti da odlučite hoćete li promjene primijeniti automatski, ili možete analizirati detalje plana i integrirati sažetak u opis zahtjeva za povlačenjem. Da bismo to napravili, prvo moramo dobiti ispis promjena u određenoj datoteci koju možemo usporediti:

```
terraform plan -input=false -compact-warnings -out=plan.file
```

Glavna točka ovdje je opcija **-out** – govori Terraformu da spremi svoj izlaz u datoteku binarnog plana, dok opcija **-compact-warnings** potiskuje poruke na razini upozorenja koje proizvodi Terraform. Također, naredba plan ima opciju **-detailed-exitcode**, koja vraća detaljne izlazne kôdove kada naredba izađe. Na primjer, možemo to iskoristiti u skripti koja pokreće Terraform i tada možemo dodati uvjetne logike njegovu izvršenju, jer CI općenito neće uspjeti u liniji na izlaznom kôdu naredbe koji nije nula. Nakon generiranja datoteke u koju smo pohranili izmjene ispisat ćemo izmjene koje je potrebno napraviti.

```
terraform plan -json|jq 'select(.type==change_summary")|."@message"'
```

Ovakav pristup u automatiziranom dijelu može učiniti Pull Request poruke informativnijima i poboljšati suradnju s timom. Ono što je očekivano ako želimo dodati dodatne provjere, primjerice je li plan u skladu s očekivanim. Nakon generirane datoteke možemo je usporediti s onom koja se nalazi u centralnom repozitoriju. Na taj način možemo utjecati na to hoćemo li zaustaviti etapu.

6.22. Primjena plana infrastrukture unutar etapa linija

Nakon što je datoteka plana spremna, a predložene promjene očekivane i odobrene, vrijeme je da primijenimo:

```
terraform apply -input=false -compact-warnings plan.file
```

U ovom trenutku izrazito je bitno referencirati se na terraform plan. U tradicionalnom slučaju, kada radimo lokalno, potvrdili bi izmjene kroz terminal, no budući da je ovdje to nemoguće, potrebno je kao referencu postaviti plan izmjena. Ponekad ovisno o zahtjevima sustava postoji potreba da se ne generira plan, već da se preskoči taj korak. U tom slučaju sljedeća naredba odmah će primijeniti konfiguraciju, bez potrebe za planom:

```
terraform apply -input=false -compact-warnings -auto-approve
```

Ovdje opcija **`–auto–approve`** govori Terraformu da implicitno izradi plan i preskoči interaktivno odobrenje tog plana prije primjene. Koji god način odabrali, imajte na umu destruktivnu prirodu naredbe apply. Stoga potpuno automatizirana primjena konfiguracije općenito dobro funkcionira s okruženjima koja toleriraju neočekivane zastoje, poput razvoja ili testiranja. S druge strane, pregled plana preporučuje se za produkcijska okruženja.

Opisan proces izrazito je jednostavan sve do trenutka kad ga moramo izvršiti unutar različitih etapa. Zamislimo da u etapi pripreme pripremamo Docker okruženje, a izvršavanje terraforma radimo u posljednjoj etapi izgradnje. Ako pokrećemo **`init`**, **`plan`** i **`apply`** naredbe u različitim etapama i okruženjima, morat ćemo se pobrinuti za artefakte koje proizvodi Terraform.

Mapa **`.terraform`** sadrži informacije o modulima, pružateljima usluga i datoteci stanja (čak i u slučaju udaljenog stanja, zato što izmjene neće biti dostupne istog trenutka na udaljenoj lokaciji).

Datoteka **`.terraform.lock.hcl`** – datoteka zaključavanja ovisnosti koju Terraform koristi za provjeru integriteta verzija pružatelja usluga koje se koriste za projekt.

Izlazna datoteka naredbe plan neophodna je za naredbu primjene. Ova datoteka uključuje punu kopiju konfiguracije projekta, stanja i varijabli proslijeđenih naredbi plana (ako ih ima). Osim što je moramo prenositi između etapa, moramo voditi računa o sigurnosnim mjerama opreza jer tamo mogu biti prisutne osjetljive informacije.

Tijekom prenošenja između etapa i pohrane na lokaciju gdje sve etape mogu koristiti izmjene moramo paziti da ih uzastopno prosljeđujemo između operacija: **`init -> plan -> apply`**. Najčešći primjer u praksi jest to da se naredbe **`init`** i **`plan`** izvršavaju unutar iste etape i prenose se spomenuti artefakti samo jednom u etapi izvršenja primjene.

6.23. Varijable okruženja

Tijekom rada s različitim etapama mogu se desiti situacije kada se određene varijable mijenjaju. Ako nemamo kontrolu nad varijablama, desit će se neželjeni ishod. Odnosno doći će do promjena koje se ne bi trebale dogoditi. Baš zbog toga želimo koristiti varijable okruženja. Varijabla okruženja dinamična je vrijednost koju operativni sustav i drugi softver mogu koristiti za određivanje informacija specifičnih za instancu. Zamislimo varijable okruženja kao globalne varijable definirane na razini operativnog sustava. Tim globalnim varijablama zapravo može pristupati bilo koji softver. Na taj način definirana varijabla ne može se samo tako izmijeniti, a možemo je iskoristiti kalo glavnu referencu za određene podatke.

Terraform trenutno posjeduje dva različita načina za korištenje varijabla okruženja:

1. Korištenje opcije **`–var–file`** s datotekom definicija varijable – naziv datoteke završava na **`.tfvars`** ili **`.tfvars.json`**
   `terraform apply –var–file=development.tfvars –input=false –no–color –compact–warnings –auto–approve`
   Također, Terraform može automatski učitati varijable iz datoteka točno nazvanih **`terraform.tfvars`** ili **`terraform.tfvars.json`**: tim pristupom ne morate eksplicitno navesti **`tfvar`** datoteku kao opciju naredbe.

2. Korištenje varijabli okruženja s prefiksom **`TF_VAR_`**. Implicitno, Terraform uvijek traži varijable okruženja (unutar konteksta procesa) s tim prefiksom, tako da se iste varijable **`„instance_type”`** iz gornjeg primjera mogu proslijediti na sljedeći način:
   `export TF_VAR_instance_type=t3.nano`
   `terraform –input=false –no–color –compact–warnings –auto–approve`

Metoda varijabla okruženja naširoko se koristi u CI-ju jer moderni CI/CD alati podržavaju upravljanje varijablama okruženja za poslove automatizacije. Ono što će korištenje varijabla okruženja također olakšati jest veličina terraform naredbi koje moramo svaki puta pisati. Jasno je da su spomenuti parametri neobavezni, no bez njih terraform u liniji ne bi funkcionirao. Pa primjerice, ako bismo podesili sljedeće varijable okruženja:

TF_INPUT – kada je postavljeno na „**false**” ili „**0**”, govori Terraformu da se ponaša na isti način kao i s oznakom **–input=false**

TF_CLI_ARGS – može sadržavati skup opcija naredbenog retka koje će se proslijediti jednoj od Terraform naredbi. Stoga sljedeća notacija može pojednostaviti izvršenje naredbi primjene i planiranja objedinjavanjem njihovih opcija za CI:

```
export TF_CLI_ARGS="–input=false –no–color –compact–warnings"
terraform plan ...
terraform apply ..
```

Osim spomenutih varijabli okruženja postoje i naprednije, koje omogućuju daleko bolju konfiguraciju etapa ili cijelih poslova u CI/CD alatu.

TF_IN_AUTOMATION – kada se postavi na bilo koju vrijednost koja nije prazna (npr. „**true**”), Terraform prestaje predlagati naredbe koje se izvode nakon one koju izvršavate, stoga proizvodi manje izlaza.

Sinkronizacija stanja u infrastrukturi kao kôd jedan je od najbitnijih dijelova s obzirom na to da stanje infrastrukture dijelimo s timom. Unutar stanja infrastrukture zapisani su svi resursi te ako bilo koji član nema posljednju verziju s izmjenama, obrisat će sve izmjene koje smo mi napravili. Kako se to ne bi desilo, moramo implementirati načine upravljanja infrastrukturom.

PITANJA ZA PONAVLJANJE:

1. Je li sigurno pohranjivati stanje infrastrukture kao kôd u udaljeni repozitorij?
2. Što je *backend* konfiguracija u infrastrukturi kao kôd?
3. Objasnite najbolju praksu dijeljenja stanja infrastrukture kao kôd.
4. Objasnite najbolji način za pokretanje Terraform skripti.
5. Što predstavlja datoteka **`.terraform.lock.hcl`**?
6. Što u izvođenju terraform plana naredba označava **`-out=plan.file`**?
7. Što radi naredba: terraform apply?
8. Što su varijable okruženja?

6.24. Apliciranje promjena nad infrastrukturom

U trenutku kada smo postavili infrastrukturu i aplikacija se izvršava određeno vrijeme, doći će vrijeme kada ćemo morati raditi ažuriranja nad infrastrukturom, bilo to zbog održavanja ili zbog dolaska novih funkcionalnosti aplikacije. Apliciranje i testiranje izmjena nad infrastrukturom izrazito je kompleksan proces s obzirom na to da želimo napraviti izmjene koje neće utjecati na kvalitetu izvršavanja aplikacije. Prilikom svake izmjene nad aplikacijom u produkciji želimo izbjeći pad rada usluge. Moramo biti svjesni da je to kontinuirani proces koji će se često dešavati i da tada dolazi do potrebe implementacije IaC tijeka rada (engl. *IaC workflov*). U procesu implementacije tijeka rada IaC nije dovoljno samo razmišljati o Clu, već i što to donosi pisanje kôda i podešavanje komponenti na infrastrukturu. Kada je riječ o softveru, znamo koliko je važno testirati aplikaciju prije nego što se objavi. Želimo osigurati da se značajke koje izgradimo stvarno ponašaju onako kako smo namjeravali. To je posebice bitno u kasnijim etapama, kada dodajemo dodatne funkcionalnost u aplikaciju, jer kako se naša aplikacija mijenja, pravilna postavka testa osigurava da ništa što dodajemo neće pokvariti naše postojeće ponašanje. Nakon što aplikacija postane živa i bude objavljena, praćenje je sljedeći korak u provjeri logike našeg kôda. Dok korisnici stupaju u interakciju s aplikacijom, razvojni inženjeri mogu kategorizirati i reagirati na probleme kojih nisu bili svjesni. Drugim riječima, cilj testiranja zaštita je od poznatih problema, poput toga kako bi aplikacija trebala reagirati ako korisnik pokuša prenijeti datoteku koja je prevelika, dok je cilj nadzora zaštita od nepoznatih problema, poput onoga što se događa ako mnogi korisnici pokušavaju učitati datoteke u isto vrijeme. Uz infrastrukturu koja je sada dostupna kao kôd, također možemo testirati i nadzirati tu logiku, baš kao što bismo to učinili za ponašanje svoje aplikacije.

6.25. Testiranje, praćenje i promjene infrastrukture

Kako bismo izbjegli kompleksnosti i probleme koji se mogu dogoditi u kasnijem razvoju infrastrukture, potrebno je implementirati određene metode testiranja IaC-a. U svijetu DevOpsa izrazito je poznat alat pod nazivom Serverspec. Serverspec je alat za testiranje infrastrukture koji je napisan u programskom jeziku Ruby i omogućuje pisanje testova kako biste provjerili je li poslužitelj ispravno konfiguriran i radi li onako

kako smo zamislili. Pruža osiguranje za to da su usluge pokrenute i omogućene, kao i za provjeru rade li procesi. Koristi se i za testiranje CPU-a ili za dodjeljivanja memorije kako bismo osigurali optimiziranu kvalitetu rada. Tako da bismo u sklopu skripte za izgradnju infrastrukture kao kôd implementirali testove koji bi provjeravali, primjerice, ponašanje web-poslužitelja ili sličnih servisa. Bitno je naglasiti da se alati za testiranje IaC-a ne izvršavaju u sklopu alata za upravljanje resursa u oblaku. Uobičajena je praksa kreirati skriptu koja će prvo pokrenuti IaC, a nakon toga izvršiti testove.

U trenutku svjesnosti za implementacijom toka rada IaC-a i testiranja infrastrukture dolazimo do praćenja i zahtjevnosti praćenja infrastrukture. Moderna arhitektura poslužitelja rijetko je standardizirana među organizacijama. Infrastrukture su izrazito skalabilne i tijek ponašanja infrastrukture izgleda, primjerice, na sljedeći način. S početkom dana desi se da infrastruktura posjeduje stotine poslužitelja, tijekom podnevna infrastruktura se skalira na tisuće instanci, a na desetke instanci navečer. Često je slučaj da u infrastrukturama nije samo jedan ponuditelj usluga u oblaku, već više njih. Precizno praćenje stanja usluga na različitim platforma velik je izazov. U ovom slučaju postoji specijalizirani alat pod nazivom Sensu. Sensu je stvoren zbog potrebe za praćenjem dinamičnih i raznolikih infrastruktura i koristi dodatke otvorenog kôda za povezivanje s DevOps alatima. Sensu također nudi REST API, koji omogućuje postavljanje upita krajnjim točkama za informacije o svim poslužiteljima i uslugama. Ovaj nadzor funkcionira korištenjem Sensu provjera (engl. *Sensu checks*), izvršnih skripti i naredbi koje je jednostavno napisati i razumjeti. Sensu provjere provode Sensu agenti u cijeloj infrastrukturi. U svojoj srži Sensu provjera pokreće naredbu u intervalu koji ste definirali. Sensu obavještava status na standardni izlaz ljuske i vraća statusni kôd koji ukazuje na ozbiljnost poruke, koja se zatim može preusmjeriti na bilo koji sustav obavijesti. Provjere se mogu koristiti za nadzor resursa poslužitelja, infrastrukturnih komponenti kao što su baze podataka ili ukupne dostupnosti usluge.

U prošle dvije cjeline objasnili smo kako funkcionira proces testiranja i praćenja. Zapravo, oba segmenta izrazito su bitna u procesu izmjena IaC-a. Vrlo često DevOps inženjeri ne spominju praćenje infrastrukture kao zasebnu cjelinu. Taj dio spada u proces testiranja. Cijelu implementaciju testiranja izrazito je bitno dodati u linija proces kako bismo prije nadogradnje provjerili ispravnost rada infrastrukture. Linija bi u tom trenutku trebala imati etapu pripreme unutar čega bismo dohvaćali trenutno stanje infrastrukture, dohvati bi iz našeg centralnog repozitorija izmjene nad infrastrukturom.

Etapa izgradnje odnosila bi se na dohvaćanja plana izmjena, s kojim bismo se trebali složiti i nakon toga dešava se primjena plana kako bi se testna okolina izgradila. Nakon toga bismo u etapi testiranja pokrenuli testiranje i instalirati potrebne agente za nadzor. Proces testiranja zapravo traje najduže jer ne želimo da testiranje završi što brže zato što želimo vidjeti ponašanje infrastrukture na određeno vrijeme. Naposljetku etapa testiranja generira izvješće o statusu i linija čeka na našu potvrdu želimo li stvarno prenijeti izmjene na produkcijsko okruženje.

U fazi apliciranja izmjena nad infrastrukturom izrazito je bitno napraviti testove kako bismo smanjili broj pogreška koje se mogu desiti. Često se izmjene apliciraju na produkcijskom okruženju, odnosno na okruženju koje koriste stvarni korisnici te u tom procesu ne bismo željeli izazvati prekid aplikacije. Nakon što smo izvršili izmjene i testovi su uspješno potvrdili izmjene, potrebno je podesiti praćenje metrika jer se pogreške mogu desiti i tek nakon određenog vremena.

PITANJA ZA PONAVLJANJE:

1. Što je IaC tijek rada? Od kojih se etapa sastoji?
2. Kako ispravno testirati promjene nad IaC infrastrukturom?
3. Što je serverspec?
4. Kako ispravno pratiti IaC infrastrukturu nakon apliciranih izmjena?
5. Što je alat Sensu?

7. Zaštita na radu

U OVOM POGLAVLJU NAUČIT ĆETE:

- razlikovati mjere zaštite na radu
- razlikovati različite izvore opasnosti pri radu s računalom
- navesti mjere zaštite od električnog udara na radnom mjestu
- objasniti upotrebu računala na siguran način primjenom mjera zaštite od ozljeda
- opisati i demonstrirati vježbe opuštanja i razgibavanja radi sprečavanja ozljeda na radnom mjestu

7.1. Osnove zaštite na radu

Zaštita na radu u Republici Hrvatskoj regulirana je Zakonom o zaštiti na radu („Narodne novine", br. 071/2014, 118/2014, 094/2018, 096/2018) te podzakonskim propisima donesenima na temelju Zakona o zaštiti na radu. Svi poslodavci trebaju je organizirati i provoditi.

Zaštita na radu sadrži sustav pravila, mjera i aktivnosti čijom se primjenom osiguravaju sigurni uvjeti na radu te zaštita zdravlja na radu. Cilj zaštite na radu prevencija je, odnosno sprečavanje ozljeda na radu te profesionalnih bolesti koje se mogu javiti prilikom obavljanja nekog posla.

Poslodavac može na različite načine organizirati zaštitu na radu, a to ovisi o djelatnosti

kojom se bavi i o broju djelatnika. Razina opasnosti utvrđuje se postupkom procjene rizika, što poslodavac provodi za sve poslove, odnosno sva postojeća radna mjesta unutar ustanove.

Osnovni pojmovi vezani uz zaštitu na radu jesu:

- **Poslodavac** je fizička ili pravna osoba za koju radnik obavlja posao.
- **Radnik** je fizička osoba koja obavlja poslove za poslodavca.
- **Mjesto rada** svako je mjesto na kojemu se obavljaju poslovi iz djelokruga poslodavca.
- **Izdvojeno mjesto rada** mjesto je na kojemu radnik obavlja posao, a nije prostor poslodavca, može biti kod kuće ili u drugom prostoru.
- **Povjerenik** za zaštitu na radu osoba je koja je izabrana da zastupa prava radnika iz područja zaštite na radu.
- **Stručnjak zaštite na radu** osoba je koju je poslodavac odredio za obavljanje poslova zaštite na radu.
- **Ozljeda na radu** ozljeda je koju je radnik zadobio tijekom rada u prostoru rada, bilo kod poslodavca ili drugom prostoru gdje se obavlja rad.
- **Profesionalna bolest** jest bolest nastala kao posljedica djelovanja štetnosti u tijeku rada i/ili radnoj okolini.
- **Stres** predstavljaju zdravstvene i psihičke promjene koje su nastale kao posljedica kontinuiranog utjecaja izvora stresa na radu kroz duži vremenski period.

Slika 7.1.1. Ilustracija loga zaštite na radu

Nezgodom na radu smatra se svaki neželjeni, nepredviđeni događaj koji za posljedicu ima ozljedu na radu koja dovodi do bolovanja, materijalnih gubitaka radnika i poslodavca zbog izgubljenih sati rada ili materijalni gubitak zbog oštećenja opreme ili zastoja u proizvodnom procesu. Nezgode na radu nastaju najčešće kao posljedica ljudske pogreške jer radnik ne zna sigurno raditi, ne može raditi na siguran način ili ne želi raditi na siguran način.

Može biti da radnik nije osposobljen za rad na siguran način i da ne zna do kojih posljedica njegov način rada može dovesti.

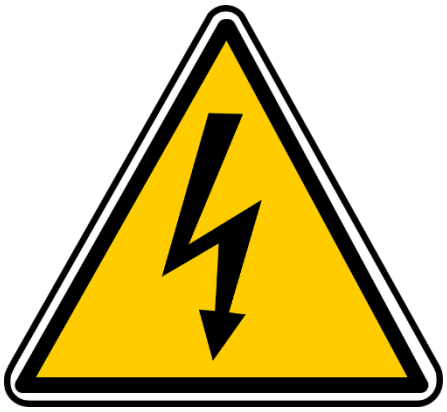
Najčešće zbog zdravstvenih razloga radnik ne može raditi na siguran način. U tom slučaju poslodavac je dužan rasporediti ga na poslove koje je u mogućnosti obavljati.

Kad radnik ne želi raditi na siguran način iako poznaje mjere zaštite na radu, on ugrožava sebe i druge djelatnike i poslodavac mora reagirati na adekvatan način, upozorenjem, razgovorom ili ga udaljiti s tog radnog mjesta.

7.2. Opasnosti i način zaštite na radnom mjestu

Izvori opasnosti pri radu s računalom mogu biti:

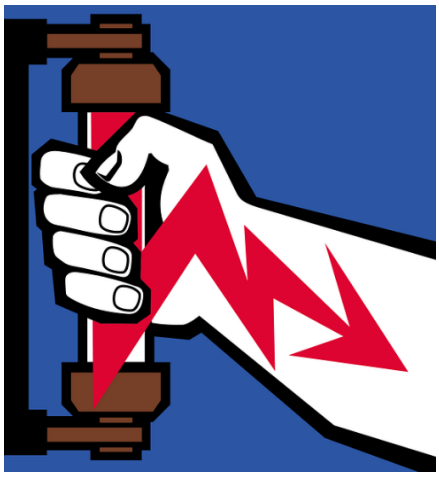
- opasnost od električkog udara
- opasnosti od buke
- opasnosti od štetnih zračenja
- opasnosti od požara i eksplozija, sredstva za zaštitu od požara.



Opasnosti od električkog udara izloženi su svi zaposlenici i posjetitelji ustanove kada električne instalacije i informatička oprema i drugi uređaji nisu ispravni ili pravilno održavani.

Električna struja ako prolazi kroz ljudski organizam može izazvati teške ozljede ili smrtne posljedice.

Slika 7.2.1. Oznaka za opasnost od strujnog udara



Slika 7.2.2.. Ilustracija strujnog udara

Nezgode od strujnog udara događaju se zbog izravnog dodira s neispravnom opremom ili oštećenom izolacijom na kablovima, sklopkama, utičnicama i slično. Elektroničku opremu i električne instalacije potrebno je periodički redovito pregledavati i održavati i na taj način osigurati da ne postoji opasnost od izravnog slučajnog dodira.

Kod uočenih smetnji ili kvarova potrebno je odmah isključiti napon ili izvući utikač uređaja iz utičnice. Isto tako ne smiju se dirati oštećeni prekidači ili utikači jer mogu biti pod naponom. Prilikom zamjene žarulje potrebno je isključiti napon. U slučaju pregrijane ili na dodir vruće utičnice ili uređaja potrebo ih je isključiti kako ne bi izazvali požar, a nakon toga pozvati električara da otkloni kvar.

U slučaju ozljede električnim udarom i ako se ozlijeđeni nalazi u strujnom krugu, potrebno ga je osloboditi od strujnog kruga poštujući mjere sigurnosti za sebe i ozlijeđenog. Najsigurnija opcija isključiti je strujni krug prekidačem ili uklanjanjem vodiča iz blizine ozlijeđenog. U tome vam može pomoći komad suhog drveta.

Osiguranje od buke jedan je segment koji također utječe na sigurnost zaposlenika. Bukom se smatra svaki neželjeni zvuk koji je zaposleniku neugodan ili mu smeta. Djelovanje buke na čovjeka ovisi o njezinoj jačini i frekvenciji. Buka može izazvati gubitak sluha, napetost, smanjenje koncentracije, probleme s razumijevanjem govora, umor i razdražljivost. Posebno je opasno oštećenje sluha jer ima trajnu posljedicu. Sluh izgubljen zbog buke ne može se liječiti. Buka se razlikuje po visini frekvencije, a buka visokih frekvencija štetnija je od buke dubokih tonova.

Buku izražavamo u mjernim jedinicama:

- decibelima (dB) – intenzitet buke
- Hertzima (Hz) – frekvencija.

Prag čujnosti je intenzitet buke 0 dB kod frekvencije 1000 Hz. Kod opremanja ureda treba paziti da uređaji koji se koriste ne stvaraju buku veću od 60 dB, koliko je propisano pravilnikom za takve prostore. Ukoliko je za rad potrebna oprema koja stvara buku veću od propisane, potrebno ju je smjestiti u zasebnu prostoriju. Stalna izloženost buci manje je opasna za zdravlje zaposlenika od povremene zato što se organizam čovjeka prilagođava okolini u kojoj stalno boravi. Osobna zaštitna sredstva koja se mogu koristiti slušalice su i čepići.

Čovjek je svakodnevno izložen nekim vrsta štetnog zračenja i one mogu utjecati na njegovo zdravstveno stanje. Dijele se na ionizirajuća zračenja, čiji su izvor radioaktivni elementi i susreću se u zdravstvenoj djelatnosti, te neionizirajuća zračenja, koja proizvode određeni uređaji. U neionizirajuća zračenja spadaju elektromegnetsko polje i elektromegnetski valovi.

Izvori neionizirajućih elektromagnetskih polja kojima su zaposlenici u uredu svakodnevno izloženi jesu antene baznih postaja, radarski sustava, televizijskih i radijskih postaja, mikrovalne pećnice, mobilni telefoni, bežični internet, bežični telefoni, daljinski upravljači, indukcijske ploče za kuhanje te protuprovalni sustavi. Udaljenost od takvih uređaja smanjuje razinu zračenja.

Preporuke za prevenciju odnose se na izbjegavanje prekomjerne upotrebe mobilnih uređaja, izbjegavanje boravka pored uređaja koji su izvor elektromagnetskih polja, izbjegavanje kontinuiranog nošenja mobilnih uređaja uz tijelo, korištenje slušalica tijekom razgovora ili pisanje poruka umjesto telefonskih razgovora.

Slika 7.2.3. Oznaka za opasnost od elektromagnetskog polja

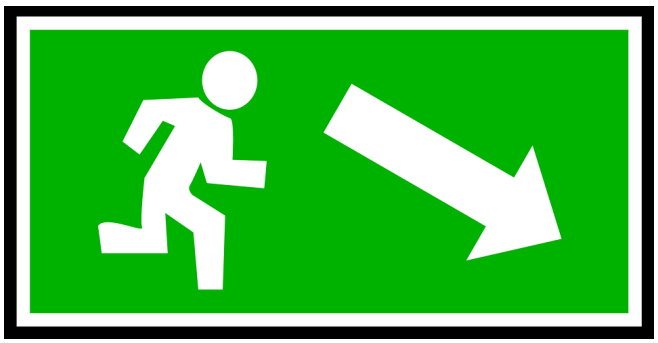
Uzrok požara može biti neodgovarajuća upotreba radne opreme ili kvar na elektroinstalacijama, kao i nepažnja zaposlenika ili posjetitelja. U slučaju curenja plina može doći do eksplozije jer čak i iskra prekidača može izazvati eksploziju. Mjere prevencije redovito su održavanje plinskih uređaja i instalacija, kao i postavljanje detektora.

Slika 7.2.4. Oznaka za opasnost od požara

Najvažnije je da zaposlenici budu osposobljeni za zaštitu na radu iz područja zaštite od požara kako bi, u slučaju nastanka požara, mogli intervenirati brzo i učinkovito. Uredi moraju imati vatrogasne aparate za slučaj požara, a zaposlenici trebaju znati kako njima rukovati. Vatrogasni aparati moraju se servisirati jednom godišnje, pristup do njih uvijek mora biti slobodan. Ustanova mora imati jasno istaknut plan evakuacije, a putevi predviđeni za evakuaciju u svakom trenutku trebaju biti prohodni. Ista pravila vrijede i za hidrante, kojima uvijek mora biti dostupan pristup i u blizini smješten ormarić s opremom za gašenje požara.

Mjere prevencije osposobljenost su zaposlenika za brzu reakciju i početno gašenje požara, svakodnevni pregled radnog mjesta, isključivanje opreme i rasvjete i slično.

U slučaju opeklina opečeni dio staviti pod tekuću čistu vodu ili u posudu s hladnom vodom. Potrebno je skinuti nakit i odjeću s opečenog dijela tijela. Ako postoji dio odjeće koji je zalijepljen za opeklinu, ne diramo ga. Iznimno dio odjeće uklanjamo s opekline kada je riječ o kemikalijama koje mogu izazvati dodatna oštećenja kože. Nastale plikove ne dirati i ne bušiti. Opeklinu zaštititi sterilnom gazom i previti zavojem ili maramom. Ukoliko se radi o većim opeklinama, ozlijeđenom treba dati da pije što više tekućine.



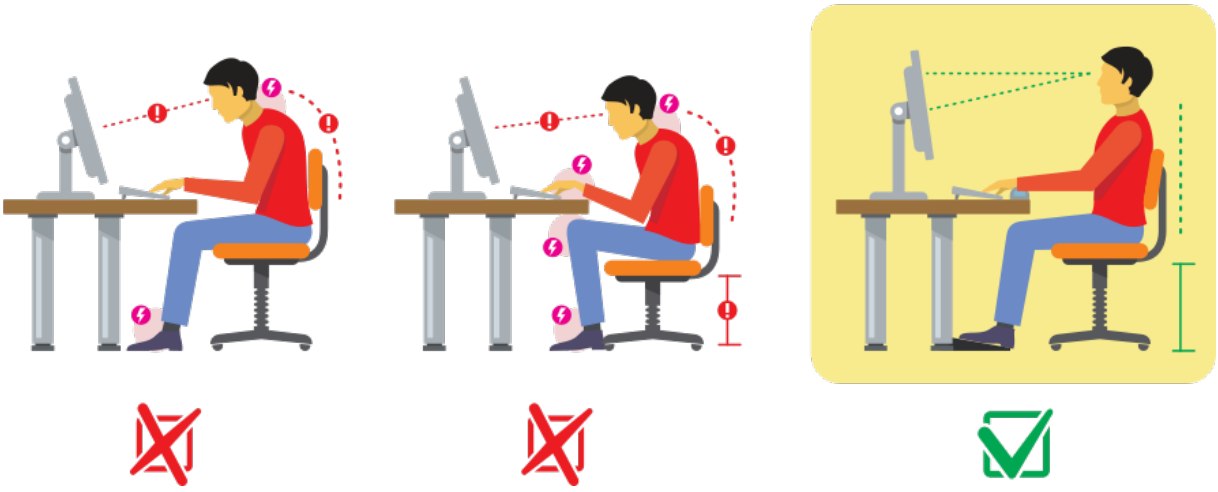
Slika 7.2.5. Oznaka puta za evakuaciju

Obveza je poslodavca osigurati sredstva za rad i osobnu zaštitnu oprema zaposlenika kako bi bili sigurni pri radu, kao i održavati istu. Oštećenu opremu ili sredstva osobne zaštite mora ukloniti i isključiti iz upotrebe ako na njoj postoje oštećenja i kao takva predstavljaju rizik za djelatnike.

7.3. Ergonomija i mjere prevencije pri radu s računalom

U radu s računalom treba osigurati da radnik može često mijenjati svoj položaj te da oprema ne bude izvor opasnosti od ozljede ili oštećenja zdravlja djelatnika. Poteškoće uzrokuju dugotrajno sjedenje u istom položaju, dugotrajno gledanje u zaslon, ponavljajući pokreti izazivaju bolove u mišićima i zglobovima. Bolesti povezane uz rad s računalom jesu bolesti vida, mišićnog sustava, koštano-zglobnog i kardiovaskularnog sustava. Nažalost, posljedice se osjete tek u kroničnoj fazi bolesti.

Ergonomija treba rad učiniti što ugodnijim i sigurnijim te smanjiti broj profesionalnih oboljenja. Ergonomija se koristi znanstvenim informacijama prilikom dizajniranja radne opreme i radne okoline.



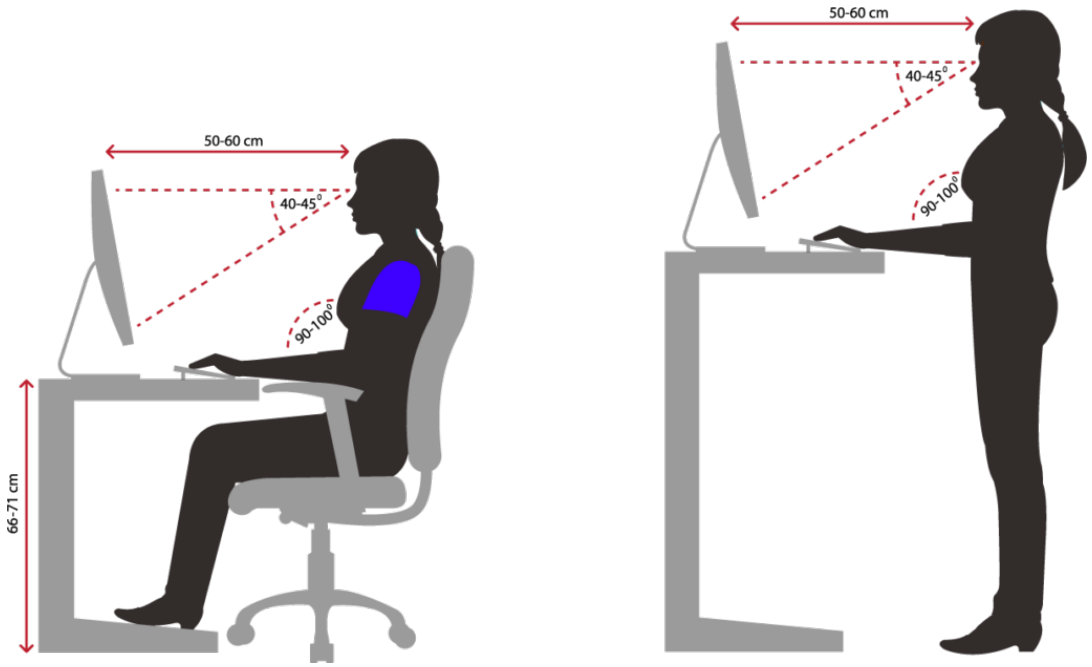
Slika 7.3.1. Prikaz ispravnog i neispravnog položaja tijela pri radu s računalom

Radno mjesto potrebno je prilagoditi svakom djelatniku. Pravilnim držanjem tijekom rada s računalom moguće je izbjeći zdravstvene probleme.

Preporučeni položaji za:

- **ruke i nadlaktice** – vodoravan položaj, paralelan s površinom poda
- **glava** – ravan položaj ili lagano naginjanje prema naprijed
- **ramena** – opuštena s nadlakticama koje vise uz tijelo
- **laktovi** – položeni uz tijelo pod kutom između 90° i 120°
- **stopala** – potpuno spuštena na pod ili oslonac za noge
- **leđa** – u potpunosti naslonjena na naslon stolice
- **koljena** – u ravnini s bokovima, savijena pod 90°.

Osnovni element uredske opreme je radni stol jer određuje raspored ostalih uređaja, a i položaj tijela tijekom rada. Radni stol mora imati dovoljno mjesta da se na njega smjeste svi potrebni uređaji za rad i da omogući djelatniku komotan rad s monitorom na njemu. Odnosno, dubina stola mora omogućiti da na stol stanu ruke i podlaktica dok se koristi tipkovnica. Poželjno je da radni stol bude podesiv po visini i da ima stalak za miš i tipkovnicu. Preporučena visina sjedećeg stola je između 66 i 71 cm. Minimum širine stola je 120 cm. Ispod stola treba biti dovoljno prostora za udobno sjedenje.



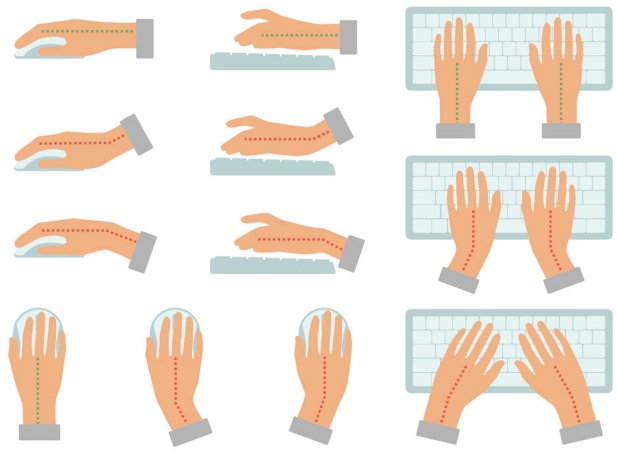
Slika 7.3.2. Prikaz ispravnog položaj tijela kod podesivih stolova

Kod stola za rad u stajaćem položaju visina radnog stola mora biti između 95 i 118 cm, ovisno o visini zaposlenika. Kod ovakve vrste stolova potrebno je paziti da se osigura potrebna dubina za stopala, 15 cm u dubinu i 12 cm u visinu.

Zaslon monitora od očiju treba biti udaljen najmanje 50 cm. Gornji rub zaslona treba biti u visini očiju. Monitor treba biti pomičan kako bi svaki djelatnik, ovisno o položaju za vrijeme rada, mogao prilagoditi njegov smjer i nagib.

Uredska stolica treba biti stabilna, prilagodljive visine, tako da svakom djelatniku može omogućiti udoban položaj i prilagodbu. Naslon mora biti potpora cijelom dužinom leđa i imati mogućnost podešavanja nagiba i visine. Stolica mora biti pokretna na 5 kotačića.

Zadovoljavajuću razinu osvijetljenosti prostorije treba osigurati općom i/ili lokalnom rasvjetom, kao i primjeren kontrast između zaslona i pozadine na računalu, poštujući specifične zahtjeve djelatnika i njegova radnog mjesta. Minimalna propisna razina osvijetljenosti za radna mjesta za računalom jest 300 luxa. Zaslon mora biti namješten i nagnut tako da se izbjegne zrcaljenje rasvjete na zaslonu. Problem refleksije i bliještanja moguće je spriječiti pravilnim smještajem radnog mjesta i uređaja sa zaslonom u odnosu na izvori svjetlosti, prozore, druge otvore ili svijetle površine. Kako bi se spriječio ulazak Sunčeve svjetlosti, na prozorima trebaju biti odgovarajući zastori ili kapci. Zaslon treba biti postavljen pod kutom od 90° prema prozoru ili nekom drugom izvoru svjetla.



Slika 7.3.3. Prikaz pravilnog i nepravilnog položaja ruku pri upotrebi miša i tipkovnice

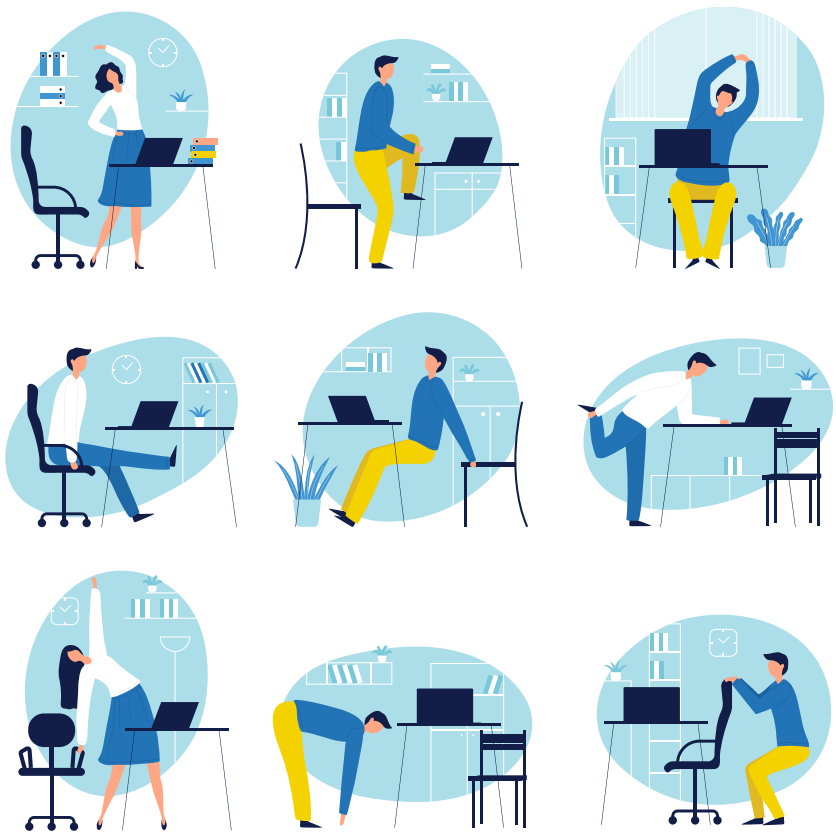
Tijekom upotrebe miša i tipkovnice treba paziti na pravilan položaj ruku kako ne bi došlo do ozljede zbog opterećenja zglobova, ruke i šake prilikom ponavljajućih pokreta (engl. *repetiton strain injury*).

Za prirodno držanje ruku poželjno je da tipkovnica bude pokretna na radnoj površini stola. Poželjno je ko-

ristiti ergonomske tipkovnice koje imaju raspored tipki prilagođen prirodnom položaju ruku. Nagib tipkovnice na podlogu trebao bi iznositi 5 – 10°, visina tipkovnice 3 cm, a visina tipki 12 – 15 mm s razmakom između tipki 18 – 20 mm.

Preporuke za prevenciju ozljeda ruku jesu:

- šaka i prsti moraju biti u ravnini s podlakticom
- prsti trebaju biti ravni u odnosu na šaku, bez savijanja zgloba
- razgibavati zglobove i prste više puta tijekom dana
- tipkovnice moraju biti na ravnoj i tvrdoj podlozi i udaljene tako da ispred njih na stol stane cijela podlaktica
- miš i tipkovnica moraju biti dostupni iz istog položaja
- pomicati cijelu ruku pri upotrebi miša, a ne samo zglobove
- ramena trebaju biti opuštena
- stolica treba imati naslon za ruke.



Slika 7.3.4. Primjer vježbi rasterećenja

Za očuvanje zdravlja kod rada u uredu pravilno sjedenje nije dovoljno, već je od velike važnosti promijeniti položaj tijela ili aktivnosti tijekom rada. Kod organizacije posla treba omogućiti radniku promjenu aktivnosti ili stanku od 5 minuta nakon svakog sata rada. U toj pauzi mogu se organizirati vježbe rasterećenja. Preporuka je više puta tijekom dana uraditi vježbe rasterećenja, koje će pomoći u smanjenju napora tijekom rada.

Vijeće ministara europske zajednice 1989. i 1990. godine donijelo je odluku o unošenju Direktive o zdravlju i zaštiti na radu u povelju o socijalnim pravima (engl. Social Charter). Direktiva 89/391/EEC ima namjeru poticanja poboljšanja zdravstvenih i sigurnosnih uvjeta na radu. Odnosi se na radni prostor, radnu opremu, osobnu zaštitnu opremu, ručno rukovanje teškim predmetima, rad s monitorima, rad s kancerogenim tvarima te rad s biološkim agentima.

Posebnu važnost ima direktiva 90/270/EEC o minimalnim zahtjevima u pogledu sigurnosti i zaštite zdravlja pri radu sa zaslonima, ali isključuje prijenosna računala.

Ova direktiva definira zahtjeve za:

- **monitore** – oblik slova, veličina i definicija, stabilnost i treperenje slike, kon trola svjetline i kontrasta, podešavanje kuta zaslona, sjaj i refleksija
- **tipkovnice** – podešavanje nagiba i odvajanje od zaslona, podrška za šaku i ruku, sjaj i refleksija, raspored tipki i čitljivost
- **radne površine** – sjaj i refleksija, prostor za opremu, držači dokumenata
- **stolice** – podešavanje i oslonac za noge
- **radnu okolinu** – rasvjeta, sjaj i refleksija, buka, toplina, zračenje i vlažnost zraka
- **korisnička sučelja** – prikladnost softvera, lakoća korištenja prilagođena vještini korisnika, vrijeme odziva i svojstva, izgled.

Pitanja za ponavljanje

1. Kojim su zakonskim aktima propisane mjere zaštite na radu?
2. Navedite i opišite osnovne pojmove vezane uz zaštitu na radu.
3. Koje su obveze poslodavca?
4. Koji su izvori opasnosti pri radu s računalom?
5. Nabrojite mjere prevencije strujnog udara.
6. Kako tretirati opekline u slučaju požara?
7. Nabrojite i opišite mjere prevencije rada s računalom.
8. Što propisuje direktiva 90/270/EEC?

„Sadržaj publikacije/emitiranog materijala isključiva je odgovornost RCK ELPROS“

Projekt je sufinancirala Europska unija iz Europskog socijalnog fonda.