

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I
INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni studij

KONTINUIRANA INTEGRACIJA PRIMJENOM
DOCKERA I GITLAB PIPELINEA

Diplomski rad

Valentin Loboda

Osijek, 2023.

Sadržaj

1. Uvod	2
2. Kontinuirana integracija	4
2.1. Definicija kontinuirane integracije	4
2.2. Izazovi implementacije kontinuirane integracije	5
2.3. Proces kontinuirane integracije	7
3. Platforme za provedbu kontinuirane integracije	10
3.1. Pregled popularnih platformi	11
3.2. Karakteristike i funkcionalnosti platformi	11
3.3. Odabir najprikladnije platforme za implementaciju.	14
4. GitLab pipeline koncepti i primjena.	16
4.1. Faze i zadatci u gitLab pipelineu	16
4.2. Definiranje GitLab CI/CD konfiguracije	18
4.3. Docker i povezivanje s dockerom za kontejnerizaciju aplikacije	18
5. Implementacija web aplikacije i kontinuirane integracije pomoću Gi-	
 tLab pipelinea u razvoju aplikacije	21
5.1. Pregled i opis jave i Spring Boot frameworka	23
5.2. Arhitektura web aplikacije	24
5.3. Implementacija funkcionalnosti backend aplikacije	25
5.4. Postavljanje GitLab repozitorija za aplikaciju	26
5.5. Definiranje GitLab CI/CD konfiguracije za aplikaciju.	26
5.6. Testiranje, izgradnja i objavljivanje aplikacije putem GitLab pipelinea .	30
5.7. Automatsko pokretanje testova i osiguranje kvalitete koda	31
Sažetak	36
Abstract	37
Literatura.	38
Životopis	40
Prilog 1.	41
Prilog 2.	44
Prilog 3.	46
Prilog 4.	52
Prilog 5.	54
Prilog 6.	58

1. UVOD

U današnjem okruženju brzog razvoja softvera, ključno je usvojiti prakse koje osiguravaju učinkovitost, kvalitetu i pravovremenu isporuku softverskih rješanja. Kontinuirana integracija (engl. *Continuous Integration* - CI) pojavila se kao temeljni koncept u ovom kontekstu. Automatizacijom procesa integriranja promjena koda, izvođenja testova i izgradnje artefakata, CI omogućuje razvojnim timovima otkrivanje i rješavanje problema rano u razvojnom ciklusu. To zauzvrat dovodi do poboljšane suradnje, smanjenih rizika i brže isporuke softvera. Cilj ovog rada je istražiti i opisati koncept kontinuirane integracije na primjeru GitLab cjevovoda (engl. *pipeline*) te istražiti alternativne platforme korištene za kontinuiranu integraciju. Ispitujući temeljna načela i najbolje prakse kontinuirane integracije, pružiti opće razumijevanje njezinih prednosti i izazova. GitLab cjevovod, često je korišten CI/CD (engl. *Continuous Integration and Continuous Deployment* - CI/CD) alat koji omogućuje automatizaciju cijelog cjevovoda isporuke softvera, od upravljanja izvornim kodom, verzioniranja, testiranja do pokretanja aplikacije u oblaku, gdje je dostupna korisnicima. Za prikaz rada GitLab cjevovod i uvid u implementaciju kontinuirane integracije, razvijena je internet aplikacija u programskom jeziku java, koristeći Spring Boot okvir. Odabrana tema za aplikaciju je backend aplikacija za vođenje evidencije tarantula u hobi teraristike. Aplikacija korisnicima omogućuje upravljanje svojim terarijima, praćenje životnog ciklusa tarantula te vođenje evidencije hranjenja i ponašanja. Implementacijom kontinuirane integracije i GitLab cjevovoda u proces razvoja, cilj je prikazati prednosti ovih praksi u poboljšanju učinkovitosti, kvalitete i pouzdanosti životnog ciklusa razvoja softvera.

Rad je podijeljen u sedam poglavlja. Drugo poglavlje daje pregled kontinuirane integracije, uključujući njezinu definiciju, prednosti i izazove. Opisuje temeljna načela i najbolje prakse kontinuirane integracije, ističući njezinu ulogu u postizanju visokokvalitetnog razvoja softvera. U trećem poglavlju dan je uvid u različite platforme dostupne za implementaciju kontinuirane integracije. Ispitujući popularne alate za kontinuiranu integraciju i implementaciju kao što su Jenkins, Travis CI i GitLab CI, uspoređene su njihove značajke, funkcionalnosti i mogućnosti integracije. Poglavlje četiri usredotočeno je na GitLab cjevovode, pružajući detaljan opis njegovih koncepata i kako ih učinkovito koristiti u CI/CD tijeku rada. Uz opis koncepata prikazan je tijek konfiguracije GitLab CI/CD cjevovoda na konkretnom primjeru, definirajući faze, poslove i tijekove rada. Nadalje, opisani su koraci integracije s docker hub repozitorijem kako bi se olakšao proces pokretanja aplikacije u oblaku. U petom poglavlju prikazana je implementacija web aplikacije koja koristi javu i okvir Spring Boot. Dan je pregled programskog jezika Java i okvira Spring Boot. Uz opis prikazana je arhitektura i funkcionalnosti web aplikacije, ocrtavajući izbor dizajna i pristupe implementaciji. Uz implementaciju web aplikacije u ovom poglavlju pokrivena je i implementacija kontinuirane integracije i GitLab cjevovoda u proces razvoja. Postavljen je GitLab repozitorij za aplikaciju, definirana je CI/CD konfiguracija i uspostavljene su faze cjevovoda za verzioniranje, izgradnju, testiranje i pokretanje aplikacije u produkcijskoj okolini. Dodatno, istraženi su postupci automatiziranog testiranja i

osiguranja kvalitete koda kako bi se osigurala robusnost i pouzdanost razvijene aplikacije. U sedmom poglavlju ocijenjeni su rezultati implementirane aplikacije, analizirajući učinkovitost kontinuirane integracije i GitLabovih cjevovoda u poboljšanju tijeka razvoja. Procijenjena je izvedba i učinkovitost implementiranog sustava, ispitujući faktore kao što su vrijeme izrade, pokrivenost testom i stope uspješnosti implementacije.

2. KONTINUIRANA INTEGRACIJA

Osobe koje nisu zaposlene u softverskim tvrtkama možda nisu svjesne sveobuhvatnog procesa uključenog u razvoj softvera izvan jednostavnog pisanja koda. Nakon što se kod pošalje na repozitorij, postoji dugačak i zamršen niz zadataka koji se moraju izvršiti da bi se izradio, potvrdio, osigurao, upakirao i implementirao softver prije nego što postane dostupan korisnicima. Ti se zadaci mogu izvršiti ručno ili, pod određenim okolnostima, u određenoj mjeri automatizirani. Međutim, i ručne i automatizirane metode pripreme softvera stvaraju određene izazove.

DevOps je nedavno nastala metodologija usmjerena na rješavanje ovih zadataka. Kombinira elemente automatizacije, suradnje, brze povratne informacije i iterativnog poboljšanja, omogućujući timovima da poboljšaju kvalitetu softvera, ubrzaju proces razvoja i smanje troškove [7].

Rasprostranjena karakteristika brojnih softverskih projekata su produljena razdoblja tijekom razvoja u kojima aplikacija ostaje nefunkcionalna. Zapravo, značajan dio vremenskog okvira razvoja softvera koji su izradili veliki timovi provodi se u neupotrebljivom stanju [10]. Temeljni razlog za ovaj fenomen sasvim je razumljiv: nema poticaja za pokušaj pokretanja cijele aplikacije dok se ona u potpunosti ne dovrši. Programeri doprinose promjenama i mogu provoditi automatizirana testiranja jedinica, ali se ne provode stvarno pokretanje i korištenje aplikacije u okruženju sličnom produkcijskom.

Ova je okolnost dodatno naglašena u projektima koji koriste proširene grane ili odgađaju testiranje prihvatljivosti (engl. *Acceptance testing*) za kasnije faze. U takvim projektima obično se značajne faze integracije vrše pred kraj razvojnog procesa, dajući razvojnom timu vremena da spoji grane i osigura da aplikacija funkcionira adekvatno za testiranje prihvaćanja [10]. Da stvar bude složenija, određeni projekti nailaze na nesretnu spoznaju da njihov softver nije prikladan za namijenjenu svrhu tijekom integracijskog razdoblja. Ove faze integracije mogu postati iznimno dugotrajne, a što je najvažnije, njihovo trajanje ostaje nepredvidivo.

2.1. Definicija kontinuirane integracije

Koncept kontinuirane integracije prvi puta je predstavljen u knjizi Kenta Becka "Extreme Programming Explained" [3], prvobitno objavljenoj 1999. Slijedeći načela ekstremnog programiranja, kontinuirana integracija proizašla je iz ideje da ako je redovita integracija baze koda korisna, zašto je ne učiniti stalnom praksom? U domeni integracije, "cijelo vrijeme" se odnosi na svaku instancu kada je izvršena bilo kakva izmjena u sustavu kontrole verzija.

Kontinuirana integracija označava značajan pomak u načinu razmišljanja. U nedostatku kontinuirane integracije, softver ostaje neispravan sve dok netko ne dokaže njegovu funkcionalnost, obično tijekom faza testiranja ili implementacije [10]. Međutim, uz kontinuiranu integraciju, funkcionalnost softvera se provjerava (pod pretpostavkom prisutnosti odgovarajuće sveobuhvatnog paketa automatiziranih testova) sa svakom novom promjenom, a svi problemi se odmah identificiraju za trenutačnu verziju koda. Timovi koji učinkovito provode kontinuiranu integraciju mogu ubrzati isporuku softvera i naići na manje nedostataka u usporedbi s

onima koji to ne čine. Otkrivanjem grešaka u ranoj fazi procesa isporuke, kada je manji trošak otkloniti ih, postižu se znatne uštede u pogledu vremena i troškova. Uslijed toga, kontinuiranu integraciju smatramo nezamjenjivom praksom za profesionalne timove.

2.2. Izazovi implementacije kontinuirane integracije

Kroz eksperimentalno istraživanje [8] provedeno na Odjelu za računalne znanosti i inženjerstvo na Sveučilištu Chalmers u Göteborgu, Švedskoj, praćeno je sedam timova kroz 13 polu strukturiranih intervjuja u periodu implementacije kontinuirane integracije u razvojni ciklus softvera unutar firme. Kroz provedeno istraživanje izazovi implementacije kontinuirane integracije podijeljeni su u sedam grupa prikazanih na slici 2.1.

Prva zamjećena prepreka u implementaciji kontinuirane integracije je način razmišljanja programera. Sedam od 13 ispitanika bilo je skeptično o prednostima implementacije kontinuirane integracije izjavljujući da ne vjeruju u prednosti dok ih ne iskuse u praksi. Neki programeri i dalje nisu uvjereni u prednosti CI-ja, dovodeći u pitanje njegovu vrijednost u usporedbi s njihovim postojećim pojednostavljenim i mjesečnim procesima isporuke. Promjena starih navika može biti izazovna za programere koji su navikli na određeni način rada, posebice pri prijelazu na kontinuiranu integraciju. Dugogodišnji zaposlenici često su otporniji na promjene u usporedbi s mlađim osobljem jer su duboko ukorijenjeni u svojim ustaljenim rutinama. CI stavlja snažan naglasak na rane i česte integracije, što može biti uznemirujuće za neke programere. Četiri od 13 ispitanika izrazilo je zabrinutost zbog ove promjene, jer su navikli provoditi značajne integracije na kraju sprinta. To im je dalo dovoljno vremena da poboljšaju svoj kod prije nego što ga integriraju. Uz CI, povećana učestalost integracije izazvala je zabrinutost oko uključivanja koda niske kvalitete koji bi se mogao suočiti s nadzorom stručnjaka i menadžera.

Drugi izazov implementacije kontinuirane integracije vezan je za interne alate i infrastrukturu poduzeća. Ovi interni resursi, koji obuhvaćaju mehanizme pregleda koda, vizualizaciju rezultata regresijskih testova, automatizirane postupke testiranja i protokole integracije koda, igraju ključnu ulogu u olakšavanju CI praksi. Međutim, tvrtke su naišle na prepreke u optimizaciji tih resursa. Prvo, njihovi alati za pregled koda kritizirani su zbog nedostatka kapaciteta za procjenu šireg konteksta promjena, što ometa učinkovitost CI procesa. Drugo, dovedena je u pitanje zrelost postojećih alata i infrastrukture, što je rezultiralo produljenim rokovima integracije koda, posebno za određene grane. Treće, povratna informacija iz automatiziranih regresijskih testova patila je od produženog vremena odgovora, u rasponu od par sati do cijelog dana, čime je bila ugrožena pravodobnost adresiranja promjena koda. Nadalje, upravljanje redom čekanja za integraciju pokazalo se složenim zadatkom zbog velikog broja promjena koda od raznih programera na različitim lokacijama, što otežava praćenje isporuka i dovodi do mogućih blokada reda čekanja. Na kraju, prepoznata je potreba za poboljšanom podrškom za automatizaciju testiranja unutar postojeće infrastrukture kako bi se smanjilo oslanjanje na ručne postupke testiranja. Ukratko, put tvrtke prema CI-ju obilježen je ovim ogromnim izazovima povezanim s njihovim internim alatima i infrastrukturom, što zahtijeva strateške prilagodbe kako bi se osigurao besprijekoran i učinkovit proces usvajanja kontinuirane integracije.

Tijekom prijelaza tvrtke na kontinuiranu integraciju, pojavili su se i izazovi vezani uz testiranje. To uključuje nepostojanje automatiziranih testova, nestabilnost u testnim slučajevima, pretjerano oslanjanje na ručno testiranje, poteškoće u sinkronizaciji izrade testnih slučajeva s implementacijom koda i zabrinutosti oko održavanja kvalitete proizvoda tijekom usvajanja kontinuirane integracije. Varijacije u pokrivenosti testovima (engl. *Test coverage*) na različitim razinama podružnica dodatno kompliciraju proces evaluacije. Uspostavljanje ravnoteže između povećane učestalosti integracije i očuvanja kvalitete vrlo je bitno pitanje koje se nameće prilikom implementacije kontinuirane integracije.

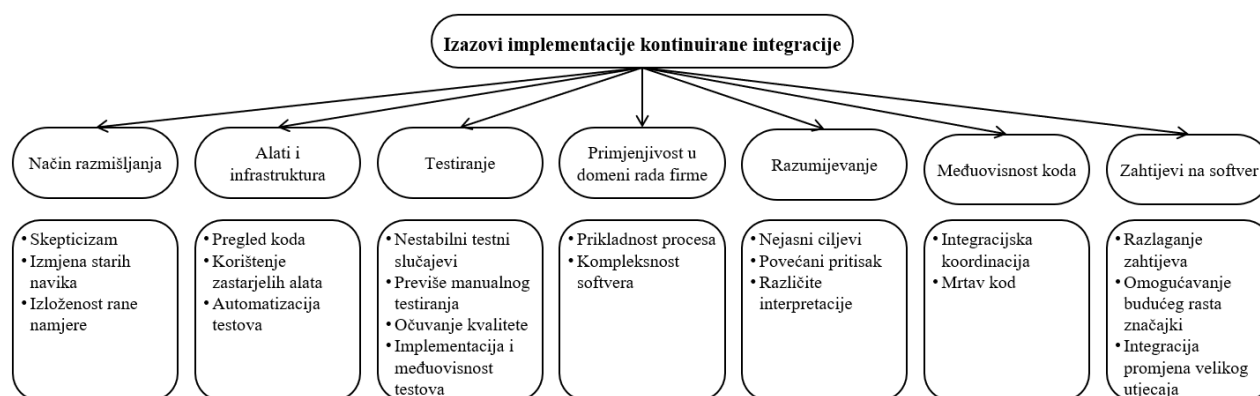
Postizanje jedinstvene učestalosti integracije u svim segmentima proizvoda pokazalo se teškim, budući da primjenjivost kontinuirane integracije varira ovisno o prirodi posla. Drugo, složenost proizvoda komplicira usvajanje kontinuirane integracije. Veličina organizacije i uključenost mnogih pojedinaca također doprinose izazovu. Ove prepreke naglašavaju potrebu za prilagođenim CI pristupima kako bi se prilagodili specifičnim zahtjevima i proizvodima tvrtke i različitim razvojnim procesima.

Istraživanje naglašava različita tumačenja koncepata i ciljeva kontinuirane integracije između timova i uprave, što rezultira s nekoliko izazova. Prvo, nedostatak jasnih ciljeva za timove koji prelaze na CI predstavlja prepreku jer neki dionici traže standardiziranije razvojnog procesa. Drugo, poticaj za usvajanje kontinuirane integracije doveo je do povećanog pritiska na timove da ubrzaju prijelaz, za što se neki osjećaju nespremima. To zahtijeva postupni pristup za ublažavanje rizika. Na kraju, početni pristup odozdo prema gore, s pilot timovima koji predvode put, općenito je dobro prihvaćen, ali postoji želja za aktivnijim uključivanjem menadžmenta u proces usvajanja kontinuirane integracije. Ukratko, organizacija se suočava s problemima tumačenja, jasnoće cilja, upravljanja pritiskom i ravnoteže uloga u usvajanju kontinuirane integracije.

Prelazak na kontinuiranu integraciju zahtijeva razbijanje rada na manje jedinice, potencijalno fragmentirajući zadatke koji bi tradicionalno bili integrirani kao jedinstvena cjelina. Posljedično, ova podjela može zahtijevati uključivanje više programera, naglašavajući važnost pažljivog upravljanja ovisnostima koda unutar procesa integracije. Koordinacija integracijskih ovisnosti pojavila se kao izazovniji zadatak u sklopu kontinuirane integracije, što dovodi do četiri prijavljena problema: potreba za jasnijim definicijama sučelja komponenti, povećane poteškoće u identificiranju izvora integracijskih pogrešaka zbog doprinosa različitih timova, veća učestalost kvarova integracije i čekanje na dovršetak drugih komponenti ili dijelova prije integriranja. Kako bi se riješili ti izazovi, predložen je koncept mrtvog koda (engl. *dead code*) — pojam gdje se djelomični kod za značajku (engl. *Feature*), korisničku priču (engl. *User story*) ili isporuku može integrirati i testirati odvojeno, aktivirajući ih tek kada su svi dijelovi spremni. Međutim, vrijedno je napomenuti da implementacija podrške za ovaj mehanizam aktivacije/deaktivacije može uzrokovati dodatne troškove.

Usvajanje kontinuirane integracije uvelo je i izazove sa zahtjevima softvera (engl. *Software requirements*). Povećana učestalost integracije zahtijevala je razbijanje zahtjeva na manje zadatke, zadatak koji su ispitanici prijavili kao izazovan. Poteškoća leži u uspostavljanju ravnoteže

između veličine ovih raščlanjenih zahtjeva, njihove mogućnosti testiranja i osiguravanja kvalitete duž linije integracije. Štoviše, čimbenici kao što su nedostatak iskustva, stalno ponovno određivanje prioriteta i evoluirajuće odluke o zahtjevima pogoršali su izazov pronalaženja ove ravnoteže. Osim toga, prilikom razlaganja zahtjeva, problem omogućavanja lakog kasnijeg rasta značajki isplivao je na površinu. Postaje izazovno odrediti vrijednost integriranja malih promjena koje možda neće izravno povećati vrijednost značajke. Te promjene, poput refaktoriranja koda ili manjih prilagodbi, možda neće izravno pridonijeti rastu značajki, ali ostaju bitne.



SL. 2.1: izazovi implementacije kontinuirane integracije [8]

2.3. Proces kontinuirane integracije

Tijekom faze kontinuirane integracije (CI), promjene razvojnog programera spajaju se i provjeravaju kako bi se osiguralo da kod ostaje funkcionalan. Glavni cilj kontinuirane integracije je brzo provjeriti te izmjene koda i odmah obavijestiti programera o svim problemima. Ovaj proces minimizira vrijeme tijekom kojeg je baza koda neupotrebljiva uslijed novo uvedenih pogrešaka. CI otkriva promjene koda i izvršava relevantne procese izgradnje kako bi pokazao da se promjene koda mogu uspješno izgraditi. Također može provoditi ciljane testove kako bi potvrdio da promjene koda rade neovisno, pri čemu ulazi proizvode željene izlaze i identificiraju i postupaju očekivano rukovajući pogrešnim ulazima.

Kontinuirana isporuka odnosi se na niz automatiziranih procesa, poznatih kao cjevovod, koji uključuje promjene koda i izvršava izgradnju, testiranje, pakiranje i povezane operacije za generiranje verzije softvera koji se može koristiti. Općenito, to se postiže s malo ili nimalo ljudske intervencije.

Kontinuirana isporuka preuzima promjene koje je potvrdio i spojio CI i nastavlja s preostalim procesima u cjevovodu kako bi se proizveli željeni rezultati. Po izboru, može pokrenuti procese kontinuirane implementacije kako bi se izdanja (engl. *builds*) automatski učinila dostupnima korisnicima. Mehanizam koji se koristi za kontinuiranu isporuku obično se naziva cjevovod kontinuirane isporuke, iako može imati i druga imena.

Iako krajnji rezultat cjevovoda često percipiramo kao ispravan kod ili softver koji se može koristiti, postoje ključni privremeni izlazi na putu. Zapravo, jedan od bitnih koraka u cjevovodu

uključuje kombiniranje novopotvrđenih i spojenih promjena s drugim kodom o kojem ovise ili s kojim trebaju raditi za generiranje artefakata. Upravljanje ovim srednjim rezultatima vrijedno je daljnjeg istraživanja.

Kontinuirano testiranje uključuje izvođenje automatiziranih testova ili drugih oblika analize koji se postupno šire kako kod napreduje kroz cjevovod kontinuirane isporuke. Neophodne su i preporučuju se različite vrste testiranja, uključujući:

- Unit testiranje, integrirano s procesima izgradnje tijekom CI faze, fokusira se na testiranje koda izolirano od drugog koda s kojim je u interakciji.
- Integracijsko testiranje (engl. *Integration testing*) potvrđuje funkcioniranje grupa komponenti i usluga u kombinaciji.
- Funkcionalno testiranje (engl. *Functional testing*) osigurava da izvršavanje funkcija u proizvodu daje očekivane rezultate.
- Testiranje prihvatljivosti (engl. *Acceptance testing*) mjeri specifične karakteristike sustava prema unaprijed određenim kriterijima, kao što su performanse, skalabilnost, stres i kapacitet.

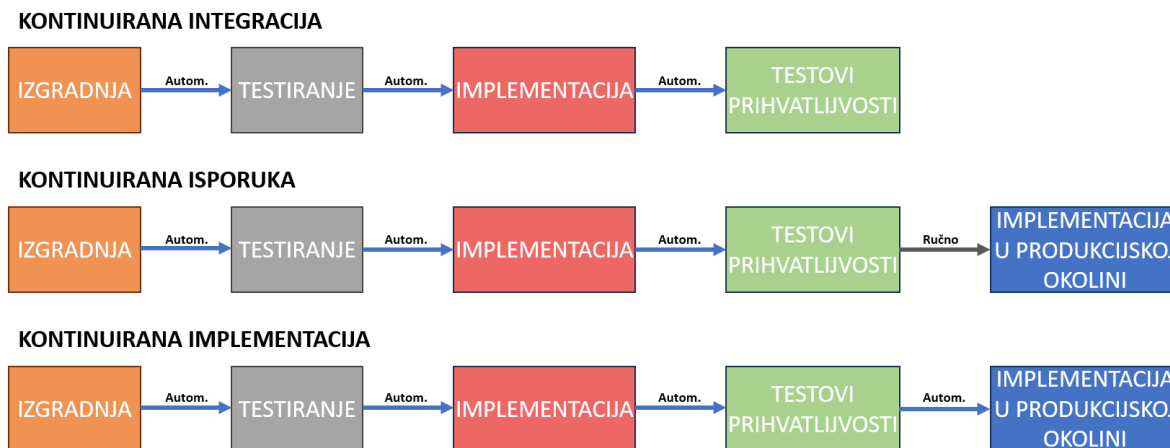
Mjerni podaci i analiza kodiranja ne spadaju u istu kategoriju kao testiranje prošao/nije prošao, ali doprinose kontinuiranom testiranju procjenom koda i kvalitete testiranja. Također se mogu koristiti kao kriteriji za pristupni (blokirajući ili dopuštajući) kod u različitim fazama cjevovoda. Neki primjeri ovih metrika i analiza uključuju:

- Analizu dijela koda pokrivenog testnim slučajevima, poznatog kao pokrivenost koda testovima (engl. *Test coverage*), koja se može mjeriti pomoću alata kao što je JaCoCo za Java kod.
- Brojanje redaka koda, mjerenje složenosti i usporedba strukture i stila kodiranja s utvrđenim najboljim praksama, što se može postići pomoću alata kao što je SonarQube. Takvi alati provode provjere, uspoređuju rezultate sa željenim pragovima, kontroliraju daljnju obradu u cjevovodu i daju integrirana izvješća o ishodima.

Važno je napomenuti da nisu sve vrste testiranja prisutne u automatiziranom cjevovodu i mogu postojati nejasne granice između nekih od ovih kategorija testiranja. Međutim, krajnji cilj kontinuiranog testiranja unutar cjevovoda isporuke ostaje isti: postupno demonstrirati kroz testiranje i analizu da trenutna verzija koda zadovoljava potrebne standarde kvalitete.

Kontinuirana implementacija odnosi se na mogućnost preuzimanja izdanja (engl. *build*) koda koji je generirao cjevovod isporuke i automatskog stavljanja na raspolaganje krajnjim korisnicima. Ova vrsta cjevovoda često se naziva cjevovod za implementaciju. Proces implementacije može uključivati implementaciju u oblaku, ažuriranje web stranice, stavljanje ažuriranja na raspolaganje ili jednostavno ažuriranje popisa dostupnih izdanja (engl. *build*), ovisno o tome kako bi korisnici "instalirali" softver.

Vrijedno je naglasiti da samo zato što se može postići kontinuirana implementacija ne znači da je svaki skup isporučenih rezultata iz cjevovoda uvijek implementiran ili da su nove funkcionalnosti odmah omogućene. Umjesto toga, cjevovod osigurava da je svaki skup isporučenih proizvoda dokazano sposoban za implementaciju kroz mehanizme poput kontinuiranog testiranja.



SL. 2.2: *kontinuirana integracija, isporuka i implementacija [1]*

Vraćanje ili poništavanje implementacije za sve korisnike može biti skupa situacija, i tehnički i u smislu percepcije korisnika. Stoga odluka o tome treba li primijeniti oslobađanje iz cjevovoda može uključivati ljudsku prosudbu. Mogu se upotrijebiti različite metode za "testiranje" verzije aplikacije prije njegove potpune implementacije, stoga je preporuka da se posljednja verzija testira u integracijskoj okolini prije njezine implementacije za korisnike u produkcijsku okolinu.[14] Na slici 2.2 prikazana je razlika kontinuirane integracije, isporuke i implementacije.

3. PLATFORME ZA PROVEDBU KONTINUIRANE INTEGRACIJE

Poduzeća sve više iskazuju afinitet prema DevOps metodologijama i agilnom pristupu razvoja softvera kako bi se ubrzao razvoj i isporuka kvalitetnih softverskih rješenja. Međutim, uspjeh ovog pristupa uvelike ovisi o odabiru pravih alata za kontinuiranu integraciju koji su usklađeni sa zahtjevima i potrebama poduzeća ili projekta.

Trenutačno je dostupan širok niz platformi za implementaciju kontinuirane integracije, što menadžerima predstavlja izazov pri odabiru najprikladnije za njihove projekte. Kako bi bili sigurni da je odabrana najbolja platforma, bitno je uzeti u obzir specifične tehničke karakteristike pojedine platforme.

Prvo, idealna platforma za kontinuiranu integraciju trebala bi ponuditi robustan ekosustav koji ubrzava isporuku projekta bez stvaranja zastoja. Osim toga, trebala bi se neprimjetno integrirati s uslugama u oblaku, omogućujući prijenos podataka u oblak i iz njega bez napora. Nadalje, platforma bi trebala pružiti pouzdane mogućnosti implementacije i podržavati integraciju s drugim alatima i uslugama koje se koriste u projektu. Ključno je dati prioritet sigurnosti, osiguravajući da odabrana CI platforma ne predstavlja rizik za podatke projekta, bez obzira na to radi li se o komercijalnom ili otvorenom programskom rješenju.

Osim tehničke sofisticiranosti, bitno je da se CI platforma uskladi s projektom i potrebama tvrtke. Ovisno o poslovnoj strategiji, moguće je odlučiti se za besplatni alat otvorenog koda ili komercijalno CI rješenje. Uz to, odabrana platforma trebala bi olakšati jednostavno upravljanje projektima i prijenos podataka, a istovremeno nuditi mogućnosti vizualizacije za bolje razumijevanje sadržaja.

Uzimajući u obzir tehničke aspekte i kompatibilnost s projektom i zahtjevima tvrtke, moguće je odabrati dobro zaokruženu CI platformu koja usmjerava razvojni proces i pridonosi cjelokupnom uspjehu projekta.

3.1. Pregled popularnih platformi

Postoje brojne platforme za kontinuiranu integraciju no neke od najpopularnijih su:

- GitLab CI
- Jenkins
- CircleCI
- TeamCity
- Bamboo
- Travis CI
- Buddy
- Codeship
- GoCD
- Semaphore

3.2. Karakteristike i funkcionalnosti platformi

GitLab CI/CD, Jenkins, CircleCI, Travis CI i TeamCity neke su od najpopularnijih CI/CD platformi koje nude opsežnu podršku i integraciju za rad s Docker kontenjerima [4, 6, 19]. Izbor između njih može ovisiti o čimbenicima kao što su specifični zahtjevi, preference alata i cjelokupni DevOps ekosustav u organizaciji. Činjenica da su ove platforme najpopularnije, ne znači da ostale platforme ne pružaju gotovo iste mogućnosti. U nastavku dan je pregled glavnih karakteristika i funkcionalnosti prethodno nabrojanih platformi.

1. GitLab CI

GitLab [9] je opsežan skup alata dizajniranih za nadgledanje različitih aspekata procesa razvoja softvera. U svojoj srži, GitLab je platforma temeljena na webu koja služi kao Git repozitorij, nudeći funkcionalnosti kao što su praćenje problema, analitika i Wiki za podršku kolaborativnom razvoju.

Jedna od bitnih značajki GitLaba je njegova sposobnost pokretanja izgradnje, izvršavanja testova i automatske implementacije koda sa svakom objavljenom promjenom. To znači da kad god se naprave promjene u bazi koda, GitLab može pokrenuti potrebne procese kako bi osigurao integritet i kvalitetu softvera. Ti se procesi mogu provesti unutar virtualnog stroja, Docker kontejnera ili na drugom poslužitelju, pružajući fleksibilnost i prilagodljivost razvojnom okruženju.

Neke od glavnih karakteristika su:

- GitLab je komercijalni alat i besplatni paket. Nudi hosting SaaS na GitLabu, na lokalnoj instanci i/ili u oblaku.
- Učinkovito upravljajte pregledom, stvaranjem i upravljanjem kodom i projektnim podacima korištenjem alata za grananje.
- Omogućuje brzu iteraciju i isporuku poslovnih vrijednosti kroz dizajn, razvoj i upravljanje kodom i projektnim podacima unutar objedinjenog distribuiranog sustava kontrole verzija.

- Pouzdana i skalabilna platforma koja služi kao mjerodavan izvor za suradnju na projektima i kodu, osiguravajući jedinstveni izvor istine.
- Olakšava usvajanje CI praksi od strane timova za isporuku kroz automatizaciju procesa izgradnje, integracije i verifikacije izvornog koda.
- Omogućuje isporuku sigurnih aplikacija i usklađenost sa zahtjevima licenciranja pružanjem skeniranja repozitorija, testiranja sigurnosti statičke aplikacije (eng. *Static Application Security Testing* - SAST), testiranja sigurnosti dinamičke aplikacije (engl. *Dynamic Application Security Testing* - DAST) i skeniranja ovisnosti.
- Automatiziranje i pojednostavljeno izdavanje i isporuka aplikacija, smanjujući ručne napore i ubrzavajući proces implementacije.

2. Jenkins

Jenkins [11] je besplatno dostupan poslužitelj za automatizaciju zadataka, poput izgradnje i kontinuirane integracije, u području razvoja softvera. Razvijen u Javi i sposoban za rad na različitim operativnim sustavima kao što su Windows, macOS i sustavima sličnim Unixu, Jenkins je opremljen širokim rasponom dodataka koji olakšavaju izgradnju, implementaciju i automatizaciju softverskih projekata.

Neke od glavnih karakteristika su:

- Jenkins je alat otvorenog koda s aktivnom zajednicom.
- Jednostavna instalacija i nadogradnje na različitim operacijskim sustavima.
- Sučelje je jednostavno i prilagođeno korisniku.
- Jenkins se može proširiti velikom zbirkom dodataka koje je pridonijela zajednica.
- Konfiguraciju okruženja lako je postaviti unutar korisničkog sučelja.
- Jenkins podržava distribuiranu izgradnju kroz master-slave arhitekturu.
- Rasporedi izrade mogu se prilagoditi na temelju izraza.
- Podržava pokretanje shell i Windows naredbi u koracima prije izgradnje
- Korisnici mogu primiti obavijesti o statusu izgradnje.

3. CircleCI

CircleCI [5] je CI/CD alat koji olakšava brz razvoj i procese izdavanja softvera. Omogućuje automatizaciju u različitim fazama korisničkog cjevovoda, uključujući kompilaciju koda, testiranje i implementaciju.

Integracijom CircleCI-ja s platformama kao što su GitHub, GitHub Enterprise i Bitbucket, korisnici mogu pokrenuti izgradnju kad god se izvrše nove promjene koda. CircleCI nudi fleksibilnost hostinga kontinuirane integracije bilo u postavkama kojima upravlja oblak ili ih pokreće interno iza vatrozida na privatnoj infrastrukturi.

Neke od glavnih karakteristika su:

- Linux planovi počinju s opcijom pokretanja jednog posla bez paralelizma bez naknade. Projekti otvorenog koda dobivaju tri dodatna besplatna kontejnera. Tijekom prijave prikazane su cijene kako bi se korisnici odlučili koji plan(ove) trebaju.
- Integracija je dostupna s Bitbucketom, GitHubom i GitHub Enterpriseom.
- Izgradnje koda se mogu izvršiti pomoću kontejnera ili virtualnih strojeva.
- Jednostavno otklanjanje pogrešaka.
- Dostupne su mogućnosti automatizirane paralelizacije.
- Brzi procesi testiranja.
- Personalizirane obavijesti mogu se slati putem e-pošte i izravnih poruka.
- Podržava kontinuiranu implementaciju, uključujući implementacije specifične za git grane.
- Visoka razina prilagodbe.
- Dostupno je automatsko spajanje i mogućnost izvršavanja prilagođenih naredbi za učitavanje paketa.
- Proces postavljanja je brz i nema ograničenja u broju nadogradnji koje se mogu izvesti.

4. Travis CI

Travis CI [20] je usluga specijalizirana za izgradnju i testiranje projekata kroz kontinuiranu integraciju. Integracijom s GitHub repozitorijem, Travis CI automatski identificira nove izmjene koda nakon čega pokreće izgradnju projekta i izvršava odgovarajuće testove i zadatke.

Travis CI nudi opsežnu podršku za različite konfiguracije izgradnje koda i programske jezike, uključujući, ali ne ograničavajući se na Node, PHP, Python, Javu i Perl.

Neke od glavnih karakteristika su:

- Travis CI je hosted CI/CD usluga. Privatni projekti mogu se testirati na travis-ci.com uz naknadu. Projekti otvorenog koda mogu se besplatno prijaviti na travis-ci.org
- Proces postavljanja je brz i jednostavan.
- Projekti na GitHub-a mogu se pratiti prikazom izgradnje uživo.
- Podržani su zahtjevi za spajanje koda (engl. *Merge request*) iz grane u granu.
- Implementacija se može izvršiti na više usluga u oblaku.
- Dostupne su unaprijed instalirane usluge baze podataka.
- Automatske implementacije pokreću se nakon uspješne izgradnje koda.
- Svaka izgradnja radi na čistom virtualnom računalu.

- Podržava macOS, Linux i iOS platforme.
- Podržano je više programskih jezika, uključujući Android, C, C#, C++, Java, JavaScript (s Node.js), Perl, PHP, Python, R, Ruby i brojni drugi.

5. TeamCity

TeamCity [18], razvio je JetBrains, poslužitelj je za upravljanje izgradnjom i kontinuiranom integracijom. Služi kao vrijedan alat za izgradnju i implementaciju različitih vrsta projekata. TeamCity radi unutar Java okruženja i lako se integrira s Visual Studio i ostalim IDE-ima. Ovaj svestrani alat može se instalirati na Windows i Linux poslužitelje, prilagođavajući se potrebama .NET i open-stack projekata.

U svojoj verziji 2019.1 TeamCity predstavlja novo korisničko sučelje zajedno s izvornom integracijom za GitLab. Također proširuje svoju podršku za GitLab i Bitbucket zah-tjeve spajanja koda iz grane u granu. Osim toga, ovo izdanje uključuje značajke kao što su autentifikacija na temelju tokena, otkrivanje i izvješćivanje o Go testovima, kao i mogućnost rukovanja zahtjevima na AWS Spot Fleet.

Neke od glavnih karakteristika su:

- TeamCity je komercijalni alat s besplatnim i vlasničkim licencama.
- Nudi različite metode za primjenu postavki i konfiguracija nadređenog projekta na podprojekte, promičući ponovnu upotrebu.
- Podržava paralelne izgradnje koda u više okruženja, omogućujući istovremeno izvođenje.
- Omogućuje značajke kao što su pregled izvješća o povijesti testiranja, prikvačivanje, označavanje i dodavanje međuverzija u favorite za učinkovito praćenje i upravljanje povijesti verzija.
- Poslužitelj je visoko prilagodljiv, interaktivan i proširiv, što omogućuje prilagođene konfiguracije.
- Osigurava funkcionalnost i stabilnost CI poslužitelja, promičući neometan rad.
- Nudi fleksibilno upravljanje korisnicima, omogućava dodjelu korisničkih uloga, grupiranje korisnika, višestruke metode provjere autentičnosti i održavanje dnevnika radnji korisnika za transparentno praćenje aktivnosti poslužitelja.

3.3. Odabir najprikladnije platforme za implementaciju

Pri odabiru platforme za implementaciju kontinuirane integracije, za aplikaciju implementiranu u sklopu ovoga rada, Jenkins i Gitlab platforme su bile glavni kandidati. I Jenkins i GitLab nude vrijedne značajke u svojim domenama. Jenkins pruža opsežnu jezičnu podršku i može se pohvaliti golemom bibliotekom dodataka. Njegovo korisničko sučelje pojednostavljuje zadatke kao što su postavljanje čvora, otklanjanje pogrešaka pokretača (engl. *Gitlab runner*) i

implementacija koda. Alat je vrlo prilagodljiv, što omogućuje fleksibilno uređivanje konfiguracije. Budući da ga je moguće postaviti na vlastitom poslužitelju, Jenkins korisnicima pruža veću kontrolu nad radnim prostorima te je upravljanje pristupom jednostavno. Međutim, Jenkins zaostaje kada je u pitanju analitika praćenja cjevovoda, što može biti nedostatak. Osim toga, konfiguriranje integracije dodataka i postavljanje alata može oduzimati puno vremena, otežavajući proces implementacije kontinuirane integracije.

S druge strane, GitLab služi kao sveobuhvatan DevOps alat koji obrađuje različite razvojne zadatke. Dolazi opremljen ugrađenim Git sustavom za kontrolu verzija, što olakšava integraciju s drugim rješenjima.

Jedna značajka GitLaba vrijedna pažnje je njegova sposobnost pružanja pregleda analitike i poslovnih uvida (engl. *Analytics and insights*), nudeći mogućnost praćenja utjecaja novih promjena na performanse aplikacije dajući uvid u učestalost korištenja pojedinih značajki aplikacije. Statistika korisnika pomaže u praćenju korištenja resursa i optimizaciji procesa, no ova je značajka dostupna samo korisnicima koji plaćaju za GitLab CI usluge. Praćenje problema je još jedna vitalna sposobnost, podržana značajkama kao što su rasprave u nitima, oznake i popisi zadataka, omogućujući učinkovito praćenje i dodjelu problema za brzo rješavanje. GitLab također omogućuje uvoz zadataka iz JIRA-e. Upravljanje spajanjem koda iz grane u granu olakšava suradnju i kontrolu verzija projekata.

GitLab se ističe u omogućavanju paralelnog izvođenja kroz faze, pojednostavljujući skaliranje pokretača (engl. *GitLab runner*). Dodavanje poslova i rješavanje konflikta u kodu je vrlo jednostavno. Alat daje prioritet sigurnosti projekta s odgovarajućim pravilima o privatnosti i neprimjetno se integrira s Dockerom.

S negativne strane, GitLab uvodi neke složenosti i napor, jer definiranje i rukovanje artefaktima za svaki posao postaje neophodno jer ne postoji predefinirana logika verzioniranja i spremanja artefakta. Testiranje spojenog stanja koda iz grane u granu (engl. *Merged code*) zahtijeva da se stvarno spajanje dovrši prije, stoga je bitno da razvojni programeri prije spajanja svog koda u *main* granu, spoje grane lokalno i tako spojen kod pošalju na git. Ovome je moguće doskočiti dodavanjem tijeka izvođenja cjevovoda (engl. *Workflow*) konfiguracijom dva cjevovoda koji će se pokrenuti na zahtjevu spajanja koda (engl. *merge request*). Prvi cjevovod pokreće se normalno, izgrađujući kod na izvornoj grani, dok drugi cjevovod spaja izvornu granu s *main* granom i pokreće izvođenje definiranih koraka cjevovoda nad kodom koji će spajanjem tek biti na *main* grani.

Iako GitLab uvodi neke složenosti, poput definiranja i rukovanja artefaktima za svaki posao, njegove sveobuhvatne značajke i mogućnosti integracije s docker registrima čine ga prikladnim za kreiranu aplikaciju. Naravno, to ne znači da je Jenkins ili bilo koja druga platforma lošiji odabir.

4. GITLAB PIPELINE KONCEPTI I PRIMJENA

Cjevovodi se najjednostavnije mogu objasniti kao niz automatiziranih *shell* naredbi, minimizirajući potrebu za ljudskim sudjelovanjem. Razumijevanje ovog temeljnog koncepta ključno je za razumijevanje CI/CD cjevovoda. Drugim riječima, CI/CD cjevovod može se opisati kao lanac naredbi koje izvršava robot i koje obuhvaćaju zadatke povezane s izgradnjom softvera, testiranjem i implementacijom.

Izvršenje ovih naredbi provode GitLab Runneri, robotski entiteti u procesu. Iz tehničke perspektive, GitLab Runner je kompaktni program koji prima naredbe od GitLab instance i izvršava ih u skladu s tim. Moguće je koristiti besplatni, zadani GitLab runner koji nije potrebno dodatno konfigurirati, no također je moguća instalacija vlastite instance GitLab runnera na virtualnom stroju.

Svaki put kada se cjevovod projekta izvede, on radi na određenoj verziji datoteka projekta. To implicira da se tijekom CI faze cjevovoda automatizirani testovi i skeniranja provode na jednoj verziji datoteka. Nakon toga, u fazi CD-a, ista verzija datoteka se postavlja u odgovarajuće okruženje. Ovaj se koncept također može izraziti kao cjevovod koji radi s određenom verzijom datoteka projekta (engl. *Works against a specific version*).

Svrha cjevovoda je procijeniti stanje koda i implementirati ga kad god se naprave promjene. Posljedično, pokretanje cjevovoda projekta na kodu od prethodnog dana dalo bi poseban skup rezultata u usporedbi s pokretanjem s kodom iz tekućeg dana, čak i ako se cjevovod sastoji od identičnih faza, poslova i naredbi. Razlika u rezultatima nastaje zbog mogućih čimbenika kao što je dodavanje novih automatiziranih testova, uvođenje softverskih grešaka koje dovode do neuspjeha testa ili uključivanje ovisnosti sa sigurnosnim propustima. Imajući to u vidu, dva pokretanja cjevovoda generirala bi suprotne ocjene u pogledu kvalitete koda.

4.1. Faze i zadatci u gitLab pipelineu

Svaki cjevovod se sastoji od jedne ili više faza, koje su skupine povezanih zadataka. Tri najčešće korištene faze su sljedeće:

1. Izgradnja: Ova faza sadrži zadatke koji kompiliraju i pakiraju izvorni kod u format koji se može primijeniti.
2. Testiranje: Ova faza obuhvaća zadatke koji izvršavaju automatizirane testove, skeniranje kvalitete koda, linting i potencijalno sigurnosno skeniranje.
3. Implementacija: Ova faza je odgovorna za slanje koda u odgovarajuće okruženje na temelju Git grane ili oznake prema kojoj se cjevovod pokreće.

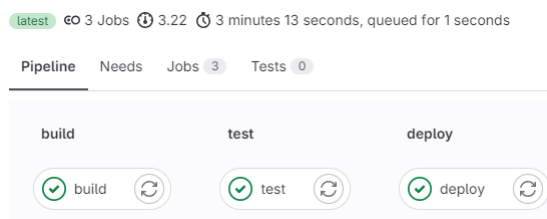
Ove tri faze unaprijed su konfigurirane u GitLabovim zadanim postavkama cjevovoda. Međutim, GitLab pruža fleksibilnost mijenjanja ove zadane konfiguracije dodavanjem, uklanjanjem ili zamjenom faza. Bez obzira na odabrane faze, preporuka je da ih se eksplicitno

definira, čak i ako je krajnja odluka korištenje zadane tri faze. Iako se ovo može činiti opširnim, poboljšava čitljivost, pomaže u rješavanju problema i sprječava zabune u budućnosti.

Ne postoji ograničenje broja faza koje je moguće definirati. Čak i za vrlo jednostavne projekte, možemo stvoriti pojednostavljeni cjevovod sa samo jednom fazom. Imena faza mogu se birati slobodno, dopuštajući razmake i razne interpunkcijske simbole. Kako bi se osigurala čitljivost, predlaže se da imena faza budu što sažetija, bez žrtvovanja jasnoće, jer duga imena mogu biti skraćena u GitLabovom grafičkom korisničkom sučelju.

Važno je napomenuti da GitLab ne provjerava tematsku vezu između zadataka unutar faze; ova odgovornost leži na osobi koja ih implementira. Posljedično, postoji sloboda kreiranja neorganiziranog i neorganiziranog cjevovoda. Na primjer, moguće je pokrenuti automatizirane regresijske testove u fazi pod nazivom "Primjena dokumentacije" ili implementirati dokumentaciju u fazi pod nazivom "priprema-testnog okruženja". Podjela cjevododa u faze i dodjela zadataka svakoj fazi u potpunosti je na osobi koja vrši implementaciju. Smatra se najboljom praksom povremeno pregledavanje i refaktoriranje strukture cjevovoda kako bi se osigurala jasnoća i dosljednost.

Kada su u pitanju komponente GitLab CI/CD cjevovoda, poslovi se mogu promatrati kao sljedeća razina ispod faza. Svaki stupanj sadrži jedan ili više poslova, a svaki posao pripada određenom stupnju.



SL. 4.3: prikaz GitLab posla s pripadajućim fazama

Gledajući sliku zaslona 4.3, možemo uočiti da faza izgradnje uključuje posao pod nazivom `build`, faza testiranja uključuje posao pod nazivom `test`, a faza implementacije uključuje posao pod nazivom `deploy`.

Kao što pokazuju ovi nazivi poslova, obično je svaki posao odgovoran za obavljanje određenog zadatka. Na primjer, posao bi mogao kompajlirati Java izvorni kod u klase, drugi posao bi mogao resetirati podatke testne baze podataka, a još jedan posao bi mogao gurnuti Docker sliku u registar. Međutim, baš kao što GitLab ne provodi tematsku sličnost poslova unutar faza, također ne potvrđuje ispunjavaju li poslovi stvarno zadatak koji sugeriraju njihova imena. Drugim riječima, imate slobodu kreirati posao pod nazivom `compile-java` koji briše zalutale datoteke generirane automatiziranim testovima ili posao pod nazivom `deploy-to-production` koji pokreće sigurnosni skener. Stoga je važno biti oprezan i promišljen pri imenovanju svojih poslova, povremeno ih pregledavajući kako biste osigurali točnost i čitljivost.

GitLab ne nameće zahtjev da svaki posao obavlja samo jedan zadatak. To znači da je moguće kreirati posao pod nazivom `test` koji izvršava višestruke automatizirane pakete testova, testove performansi i sigurnosne skenere. Međutim, smatra se najboljom praksom svaki posao

fokusirati na jedan zadatak.

4.2. Definiranje GitLab CI/CD konfiguracije

U prethodnom poglavlju definiran je proces konfiguriranja CI/CD cjevovoda, koji uključuje definiranje faza, poslova i naredbi, no kako zapravo izvesti ovu konfiguraciju i gdje? Sva konfiguracija za CI/CD cjevovod obavlja se unutar datoteke pod nazivom `gitlab-ci.yml`, koja se nalazi u korijenskom direktoriju repozitorija projekta. Ako istražimo bilo koji javni GitLab projekt s implementiranim cjevovodom, možemo pronaći datoteku s ovim nazivom koja određuje radnje konfigurirane za taj projekt.

Svaka datoteka `gitlab-ci.yml` koristi jezik YAML (*yet another markup language* ili YAML ain't markup language) dizajniran za tu svrhu. Ovaj se jezik sastoji od ključnih riječi, vrijednosti i nekih sintaktičkih elemenata. Određene ključne riječi koriste se za definiranje faza i poslova unutar tih faza. Ostale ključne riječi koriste se za prilagodbu poslova i njihovog ponašanja unutar cjevovoda. Dodatno, postoje ključne riječi za postavljanje varijabli, određivanje Docker slika za poslove i utjecaj na cjelokupni cjevovod na različite načine. Ovaj jezik je dovoljno fleksibilan da se prilagodi širokom rasponu zadataka u CI/CD cjevovodima, ali ipak nije pretjerano složen (barem nakon što se stekne neko iskustvo u pisanju i razumijevanju ovih CI/CD konfiguracijskih datoteka).

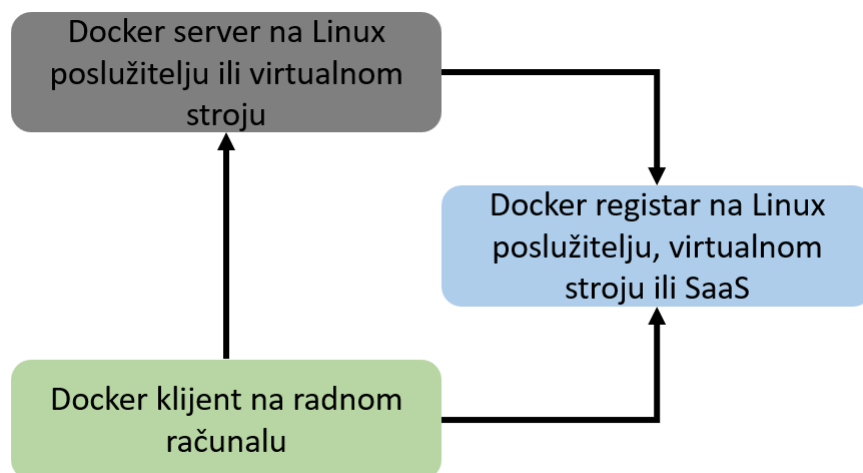
Postoji otprilike 30 ključnih riječi dostupnih za korištenje u datoteci `gitlab-ci.yml`. Umjesto pamćenja svih pojedinosti i opcija konfiguracije povezane sa svakom ključnom riječi, preporuča se usredotočavanje na shvaćanje ukupnih mogućnosti koje nude CI/CD cjevovodi. Nakon shvaćanja općih mogućnosti, lakše je upoznati se sa specifičnim ključnim riječima prema potrebi. Službena dokumentacija GitLaba najpouzdaniji je izvor informacija o ključnim riječima, osobito jer se one mogu mijenjati tijekom vremena.

4.3. Docker i povezivanje s dockerom za kontejnerizaciju aplikacije

Docker ekosustav potaknuo je uspješnu zajednicu koja se sastoji od programera i administratora sustava. Slično pokretu DevOps, ova je zajednica prepoznala vrijednost rješavanja operativnih izazova putem koda, što je dovelo do razvoja poboljšanih alata. U slučajevima kada Dockerov izvorni alat zakaže, razne tvrtke i pojedinci su preuzeli inicijativu da popune te nedostatke pa tako primjerice GitLab ima dostupan vlastiti Docker registar u koji se mogu spremati kontejnerizirane verzije aplikacije. Mnogi Docker alati su otvorenog koda, što omogućuje njihovo proširenje i prilagodbu drugim organizacijama kako bi zadovoljile svoje specifične zahtjeve.

Docker se može podijeliti u dvije glavne komponente: klijent i poslužitelj/daemon. Dodatno, postoji izborna treća komponenta koja se zove registar, koja služi kao pohrana Docker slika i njihovih metapodataka. Poslužitelj/daemon odgovoran je za kontinuirane zadatke izgradnje, pokretanja i upravljanja spremnicima, dok se klijent koristi za davanje uputa poslužitelju o radnjama koje treba izvršiti. Docker demon se može instalirati na više poslužitelja unutar infrastrukture, a jedan klijent može komunicirati s više poslužitelja. Klijenti djeluju kao glavni

pokretači komunikacije, ali Docker poslužitelji također mogu izravno komunicirati s registrima slika prema uputama klijenta. U tom kontekstu, klijenti su odgovorni za izdavanje uputa poslužiteljima, dok su poslužitelji prvenstveno usredotočeni na hosting i upravljanje kontejnerskim aplikacijama.



SL. 4.4: *Docker klijent/server model* [12]

Docker ima jedinstvenu strukturu u usporedbi s nekim drugim klijent/poslužitelj softverima. Sastoji se od docker klijenta i dockerd poslužitelja. Međutim, umjesto da bude potpuno monolitan, poslužitelj koordinira razne druge komponente u pozadini u ime klijenta. Jedna od tih komponenti je containerd-shim-runc-v2, koja olakšava interakciju s runc i containerd. Unatoč ovim temeljnim složenostima, Docker pojednostavljuje proces pružanjem jednostavnog sučelja klijenta i poslužitelja, prikazanog na slici 4.4. U većini slučajeva može ga se shvatiti kao jednostavnog klijenta i poslužitelja. Tipično, svaki Docker host pokreće jedan Docker poslužitelj koji može upravljati s više kontejnera. Za komunikaciju s poslužiteljem moguće je koristiti command-line alat docker, bilo izravno sa samog poslužitelja ili, ako je prikladno osiguran, s udaljenog klijenta.

Command-line alat Docker pruža oznaku izgradnje koja omogućuje korištenje Dockerfilea za izradu Docker slike. Svaka uputa unutar Dockerfilea pridonosi novom sloju unutar slike, čineći jednostavnim razumijevanje radnji koje se izvršavaju pogledom u Dockerfile. Značajna prednost ove standardizacije je da svaki inženjer koji ima iskustva s Dockerfileom može lako modificirati proces izgradnje za različite aplikacije. Zbog standardizirane prirode Docker slika, alati uključeni u izgradnju ostaju dosljedni, bez obzira na programski jezik, osnovnu sliku ili broj uključenih slojeva. Obično se Docker datoteke pohranjuju u sustavima za kontrolu revizija, što pojednostavljuje praćenje promjena u izradi.

U modernim višestupanjskim Docker izgradnjama moguće je odvojiti okruženje za izradu od konačne slike artefakta. Ovo odvajanje nudi široku konfiguraciju za okruženje izgradnje, slično konfiguracionim mogućnostima dostupnim za proizvodne spremnike.

Mnoge izgradnje Dockera izvode se jednim pozivanjem naredbe za izgradnju slike dockera, što rezultira jednim artefaktom, naime slikom kontejnera. Budući da je većina logike izgradnje obično sadržana unutar samog Dockerfilea, postaje jednostavno stvoriti standardizirane poslove

izgradnje koje mogu koristiti različite platforme u sustavima izgradnje kao što je Jenkins. Osim toga, brojne tvrtke, uključujući eBay, prihvatile su standardizaciju Linux kontejnera za izradu slika pomoću Dockerfilea. SaaS platforme za izgradnju kontinuirane integracije kao što su Travis CI i CodeShip također nude robusnu podršku za Docker gradnje, integrirajući ih kao temeljne značajke svojih usluga.

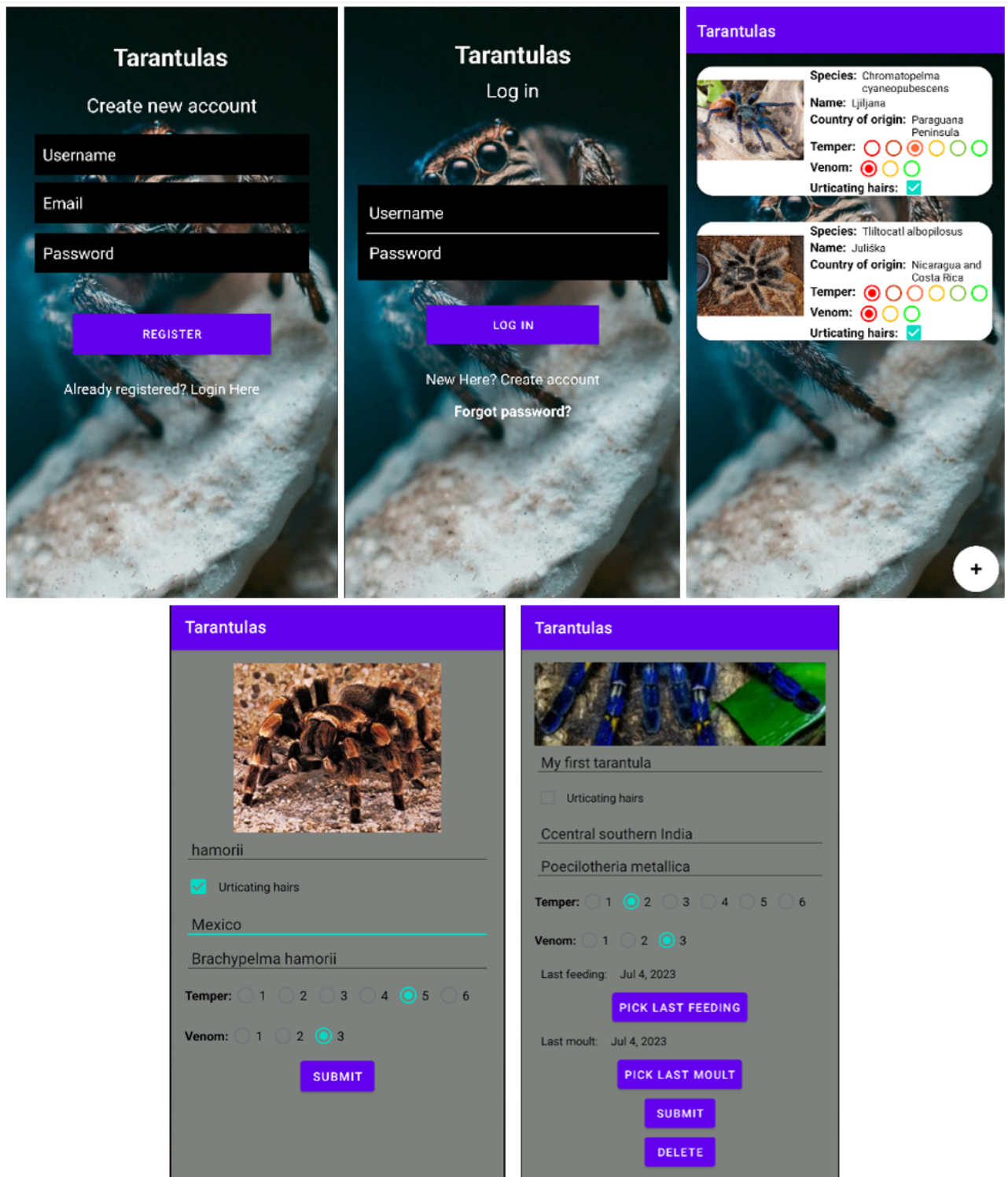
5. IMPLEMENTACIJA WEB APLIKACIJE I KONTINUIRANE INTEGRACIJE POMOĆU GITLAB PIPELINEA U RAZVOJU APLIKACIJE

Kao demonstracija korištenja CI/CD sustava izrađena je pozadinska (engl. *backend*) web aplikacija za vođenje evidencije životnog ciklusa i prehrane tarantula u hobiju teraristike. Pozadinskoj web aplikaciji implementiran je GitLab cjevovod kako bi se postigla kontinuirana implementacija aplikacije na google cloud spajanjem koda u *main* granu. Uz pozadinsku web aplikaciju, koja ima konfiguriran CI/CD, kreirana je i Android aplikacija, koja koristi pristupne točke pozadinske aplikacije. Android aplikacija kreirana je korištenjem Kotlin programskog jezika te se izvorni kod može pronaći na poveznici: <https://github.com/vvvaalll/tarantulas-2.0>. Cilj aplikacije je omogućiti lak uvid u popis tarantula u posjedu, kada su posljednji puta mijenjale egzoskelet te kada su posljednji puta jele. Ove informacije bitne su za praćenje životnog ciklusa tarantula jer neke vrste odbijaju hranu po godinu dana u vrijeme pripreme za presvlačenje egzoskeleta, no bitno je prepoznati da je to glavni razlog. Snimke zaslona android aplikacije prikazane su na slici 5.5 gdje su vidljivi zasloni za prijavu i kreiranje korisničkog računa, prikaz popisa dodanih tarantula, forma za dodavanje tarantule te forma za uređivanje podataka o tarantuli.

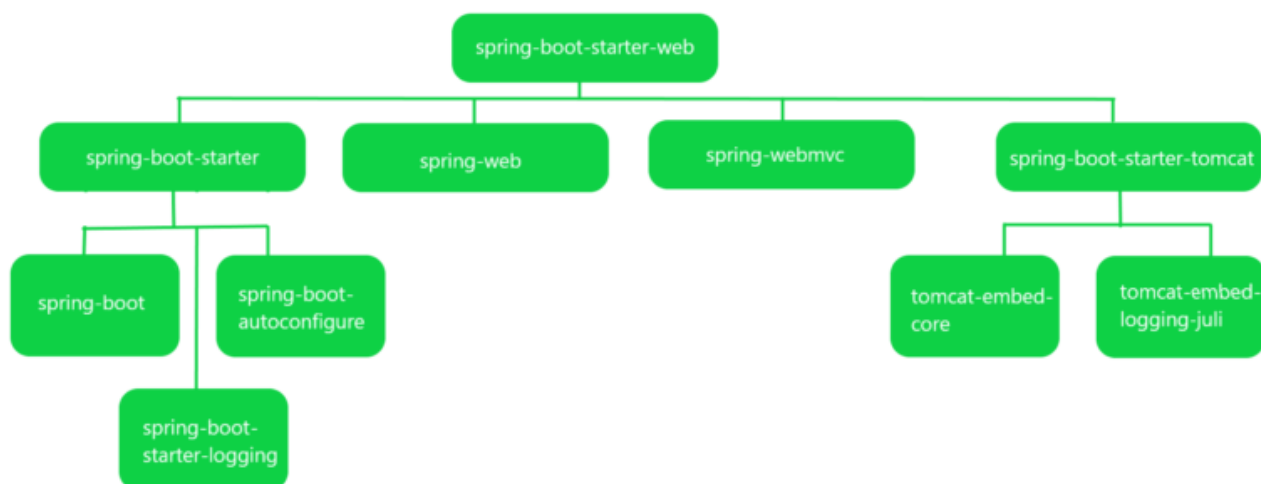
Kreirana pozadinska aplikacija s pripadajućom `gitlab-ci.yaml` datotekom koja sadrži konfiguraciju cjevovoda nalazi se na GitLabu na sljedećoj poveznici: <https://gitlab.com/tarantulas/file-management>.

Za izradu web aplikacije kojoj je konfigurirana kontinuirana integracija, korištenjem GitLab cjevovoda, korišten je Spring Boot okvir. Spring Boot pruža pomoć u upravljanju ovisnostima projekta kroz koncept početnih ovisnosti (engl. *Spring Boot starter dependencies*) [21]. Ove početne ovisnosti funkcioniraju kao jedinstvene Maven (i Gradle) ovisnosti, iskorištavajući mehanizam tranzitivnog rješavanja ovisnosti za prikupljanje često korištenih biblioteka pod nekoliko odabranih ovisnosti specifičnih za značajke [15]. Konceptom tranzitivnih ovisnosti izbjegava se potreba za dodavanjem velikog broja paketa potrebnih za razvoj i rad aplikacije. Osim jednostavnijeg upravljanja paketima, tranzitivnim ovisnostima riješen je problem velikog broja paketa čije verzije je potrebno podizati ukoliko se u nekom od paketa pojavi sigurnosni propust. Na slici 5.6 prikazano je stablo tranzitivnih ovisnosti početne Spring Boot ovisnosti `spring-boot.starter-web`.

Za izradu aplikacije korišten je Apache Maven, koji je okvir za upravljanje projektima otvorenog koda, temeljen na standardima koji pojednostavljuju izgradnju, testiranje, izvješćivanje i pakiranje projekata. Maven naredbe pokreću se kroz komandnu liniju, što je bitno za mogućnost konfiguracije cjevovoda [15]. Implementacija kreirane aplikacije nalazi se na GitLab repozitoriju na linku: <https://gitlab.com/tarantulas/file-management>.



SL. 5.5: Prikaz snimki zaslona android aplikacije koja koristi kreiranu pozadinsku web aplikaciju.

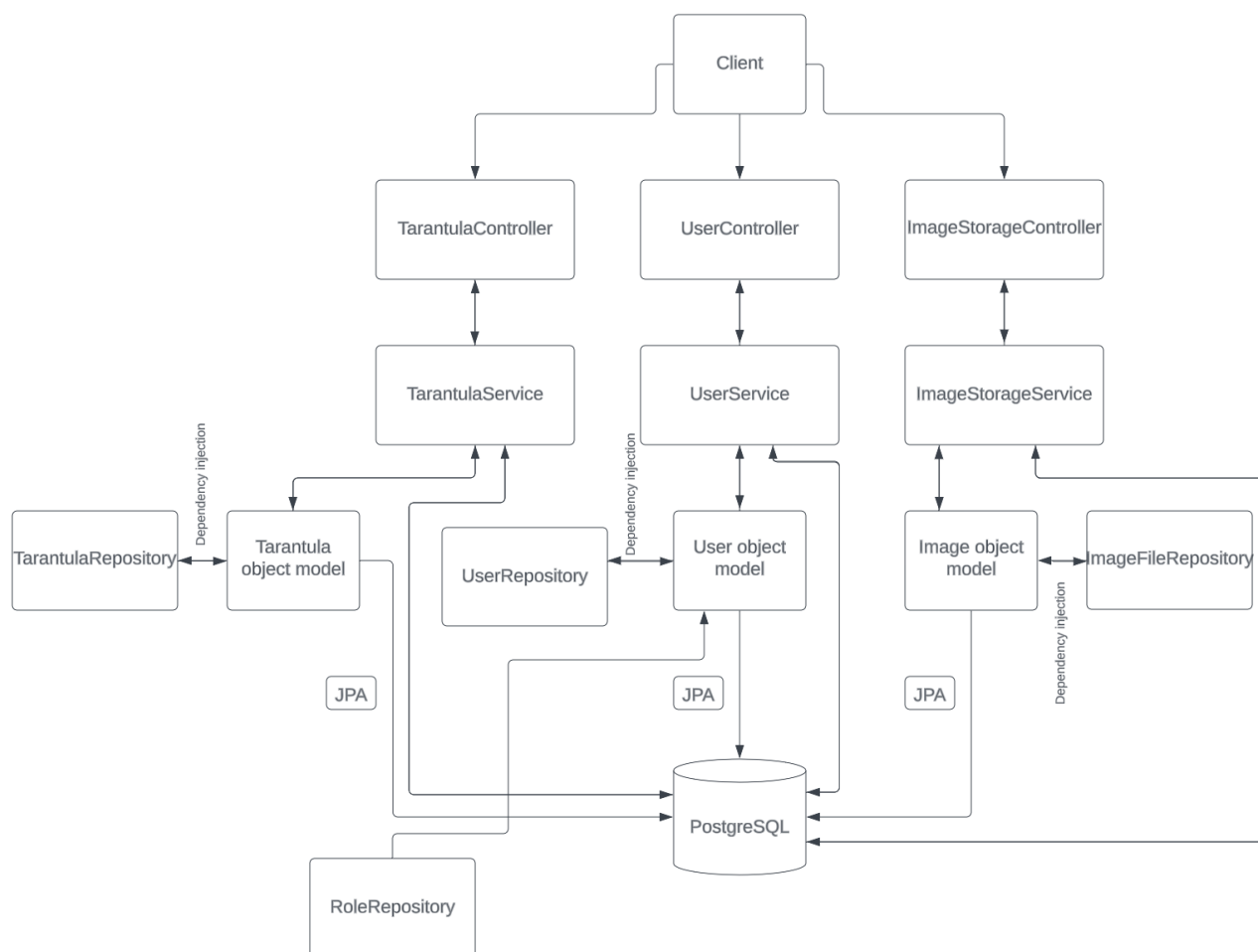


SL. 5.6: *Prikaz stabla tranzitivnih ovisnosti `spring-boot-starter-web` početne Spring Boot ovisnosti. [13]*

5.1. Pregled i opis jave i Spring Boot frameworka

Više od desetljeća, Spring Framework se pozicionirao kao dominantan okvir za razvoj Java aplikacija, postavši glavni izbor. Spring se u početku pojavio kao lakša alternativa Java Enterprise Editionu (JEE, također poznata kao J2EE), odstupajući od složenog pristupa razvoja teških Enterprise JavaBeansa (EJB). Umjesto toga, Spring je prihvatio jednostavniju rutu za poslovni razvoj Java aplikacija, koristeći ubrizgavanje ovisnosti i aspektno orijentirano programiranje za repliciranje EJB mogućnosti korištenjem običnih Java objekata (engl. *Plain Old Java Object* - POJO).

Međutim, unatoč pojednostavljenim komponentama koda, Spring se suočio sa zamršenostima u pogledu konfiguracije. U početku je XML služio kao konfiguracijski medij za Spring (i to u znatnim količinama). Pojavom Spring 2.5 uvedeno je skeniranje komponenti putem komentara, značajno smanjujući potrebu za eksplicitnom XML konfiguracijom u internim komponentama aplikacije. Naknadno izdanje Spring 3.0 uvelo je konfiguraciju, temeljenu na Javi, za organizacijsku alternativu XML-u i koju je lako moguće refaktorirati. Ipak, izbjeci konfiguracijske zahtjeve pokazalo se nedostižnim. Eksplicitna konfiguracija, bilo u XML-u ili Javi, ostala je neophodna za aktiviranje specifičnih Spring funkcionalnosti kao što su upravljanje transakcijama i Spring MVC. Uključivanje značajki iz vanjskih biblioteka, kao što su web prikazi temeljeni na Thymeleafu, također je zahtijevalo eksplicitno postavljanje. Konfiguracija servleta i filtara (kao što je Springov `DispatcherServlet`) zahtijevala je eksplicitne konfiguracijske korake unutar `web.xml` ili inicijalizatora servleta. Dok je skeniranje komponenti smanjilo opseg konfiguracije, a konfiguracija Jave učinila je manje glomaznom, Spring je nastavio nametati značajne troškove konfiguracije. Vrijeme uloženo u izradu konfiguracije umanjuje raspodjelu napora prema stvarnoj logici aplikacije. Nadalje, nadziranje upravljanja ovisnosti o projektu ostaje često podcijenjen pothvat. Određivanje odgovarajućih biblioteka za uključivanje u izgradnju projekta već je složen zadatak. Međutim, točno mjerenje koje su verzije tih biblioteka kompatibilne jedna s drugom predstavlja još zastrašujući izazov. Upravljanje ovisnošću, koliko



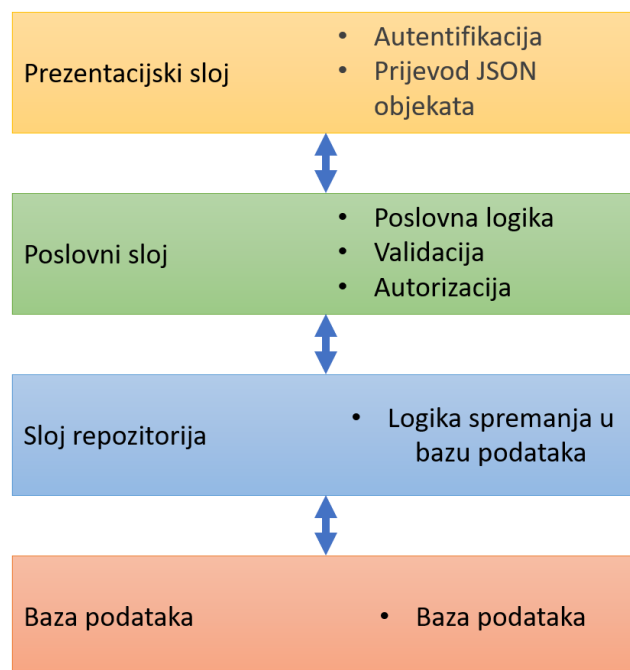
SL. 5.7: Dijagram arhitekture web aplikacije

god bilo vitalno, predstavlja dodatni sloj opstrukcije. Kada uvedete ovisnosti u svoju izgradnju projekta, to znači da se vrijeme odvaja od razvoja koda aplikacije. Neuspjesi koji proizlaze iz potencijalnih nedosljednosti koje proizlaze iz netočnog odabira verzija ovisnosti mogu značajno ugroziti produktivnost. U ovom krajoliku, Spring Boot je donio transformativnu promjenu.[15]

5.2. Arhitektura web aplikacije

Aplikacija je kreirana koristeći Spring MVC obrazac. *Spring Web model-view-controller* (MVC) okvir dizajniran je oko *DispatcherServleta* koji šalje zahtjeve rukovateljima, s konfigurabilnim mapiranjem rukovatelja, postavkama pogleda, lokalizacijom i postavkama teme kao i podrškom za učitavanje datoteka. Zadani rukovatelj temelji se na anotacijama `@Controller` i `@RequestMapping`, nudeći širok raspon fleksibilnih metoda rukovanja komunikacijom s klijentom. S uvođenjem Spring 3.0, mehanizam `@Controller` također omogućuje stvaranje REST-ful web stranica i aplikacija, putem `@PathVariable`, `@RequestBody` anotacija te drugih značajki.[22]

Na slici 5.7 prikazana je međuovisnost glavnih dijelova Spring Boot arhitekture na primjeru kreirane aplikacije. Spring Boot arhitektura razvoja aplikacije razdvaja aplikaciju u četiri sloja , prikazana na slici 5.8. Prvi je sloj baze podataka, nakon njega slijedi sloj repozitorija. U



SL. 5.8: *Prikaz slojeva Spring Boot arhitekture [17]*

sloju repozitorija JPA (Java Persistence API) omogućuje lakšu komunikaciju s bazom podataka dajući gotove metode za manipulaciju podacima u bazi. Uz predefinirane metode omogućuje lako pisanje vlastitih metoda uz definiciju Query naredbe koristeći anotaciju `@Query`. Poslovni sloj bavi se logikom rada aplikacije, ovdje se nalazi izvršavanje kalkulacija, dohvaćanje podataka iz baze kroz repozitorij i autorizacija. Posljednji, sloj Spring Boot arhitekture je prezentacijski sloj. U posljednjem sloju definira se kontroler aplikacije i pristupne točke za komunikaciju sa Spring Boot aplikacijom.

5.3. Implementacija funkcionalnosti backend aplikacije

Aplikacija ima definirane CRUD (skraćenica engleskog kreiraj, dohvati, izmijeni, obriši) servise za upravljanjem podacima korisnika, tarantula i slika. U nastavku je dan kod implementacije `UserController` klase koja predstavlja prezentacijski sloj aplikacije za rad s podacima korisnika. Kod `UserController.java` klase, u kojem se nalazi definicija REST pristupnih točaka dan je u prilogu 1.

Na danom primjeru kontrolera bitno je obratiti pažnju na anotacije metoda i same klase. Kontroler klasa označava se `@RestController` anotacijom. Na primjeru je moguće uočiti `@Operation` anotaciju za generiranje swagger dokumentacije, `@SecurityRequirements` i mapping anotacije kojima se definira RESTfull mrežni servis. U kontroler korisničkim podacima korištenjem loombok anotacije `@RequiredArgsConstructor` radi se ubrizgavanje ovisnosti `UserService` (poslovnog sloja) kroz konstruktor.

Za implementiranje REST pristupne točke konfigurirana je swagger dokumentacija. Za generiranje dokumentacije, u aplikaciji je potrebno konfigurirati `OpenAPI` klasu. Kreirana konfiguracija dana je u prilogu 6. Nakon konfiguracije, na `http://localhost:8080/swagger-ui-`

`/index.html` moguće je vidjeti generiranu dokumentaciju i testirati kreirane funkcionalnosti. Snimka zaslona dokumentacije pristupnih točaka za upravljanje podacima prikazana je na slici 5.9. Primjer poziva pristupne točke za generiranje autentifikacijskog tokena, korištenjem korisničkog imena i lozinke, dan je na slici 5.10.

5.4. Postavljanje GitLab repozitorija za aplikaciju

Pri kreiranju projekta na GitLab korisničkom sučelju moguće je odabrati opciju za kreiranje novog repozitorija. Klikom na gumb za izradu novog repozitorija otvara se obrazac, prikazan na slici 5.11, koji je potrebno popuniti željenim podacima.

Nakon kreiranog repozitorija, moguće ga je klonirati na lokalno računalo, ili koristiti GitLab IDE za razvoj aplikacije. GitLab posjeduje zadani `gitlab-runner`, no za potrebe projekta kreiran je vlastiti `gitlab-runner` na Google cloud virtualnoj mašini koji će se koristiti za izvođenje kontinuirane implementacije. Upute kreiranja `gitlab-runnera` moguće je pronaći u službenoj GitLab dokumentaciji na linku: <https://docs.gitlab.com/runner>.


Vrlo korisna mogućnost GitLaba je kreiranje varijabli. Na slici 5.12 prikazano je sučelje za upravljanje GitLab CI/CD varijablama. Varijable se mogu koristiti za pohranu osjetljivih informacija, kao što su lozinke i tajni ključevi, i moguće ih je koristiti u `gitlab-ci` konfiguraciji. Varijable se mogu klasificirati u tri vrste: zaštićene, maskirane i proširene.

- Zaštićene varijable izložene su samo zaštićenim granama ili zaštićenim oznakama. To znači da neće biti vidljive korisnicima koji nisu ovlašteni za pregled zaštićenih grana ili oznaka.
- Maskirane varijable skrivene su u zapisnicima poslova. To znači da će biti zamijenjeni zvjezdicama u zapisnicima poslova, što ih čini nečitljivima. Maskirane varijable također moraju odgovarati zahtjevima maskiranja, što su pravila koja određuju koliko su osjetljive informacije maskirane.
- Proširene varijable su varijable sa znakom dolara (\$) na početku. Te se varijable tretiraju kao početak reference na drugu varijablu. Na primjer, varijabla `$DB_PASSWORD` bi se proširila na vrijednost varijable `DB_PASSWORD`.

5.5. Definiranje GitLab CI/CD konfiguracije za aplikaciju

GitLab ci/cd konfiguracija definira se u `gitlab-ci.yaml` datoteci smještenoj u korjenskoj mapi projekta. U prilogu 2 prikazan je `gitlab-ci.yaml` kod konfiguracije.

Na početku konfiguracijske datoteke prvo se konfiguriraju ovisnosti, u ovom slučaju dodana je skripta otvorenog koda, korištena za verzioniranje aplikacije, koju je moguće pronaći na linku: <https://gitlab.com/base58-public/ci/templates/-/blob/main/Jobs/Auto-Versioning.yml>. Nakon dodavanja ovisnosti, definiran je docker kontenjer koji sadrži instaliran docker servis kako bismo unutar njega mogli pokrenuti druge docker kontenjere tijekom izvršavanja cjevovoda. Na


Swagger
powered by SMARTBEAR

Select a definition

Users

Authentication
Image files management
Tarantulas management
Users

Tarantulas backend service

[/v3/api-docs/Users](#)

Manage user access and data, add and manage tarantulas and image files

[Apache 2.0](#)

Servers

http://34.118.23.79 - Generated server url

Authorize

user-controller

GET
/api/users
Returns list of all registered users. Requires Admin rights

PUT
/api/users
Update user attributes.

PUT
/api/users/deactivate/{id}
Activate - deactivate user by id.

POST
/api/users/signup
Register new user with provided attributes.

POST
/api/users/revoke-admin
Revoke admin permissions to user with provided id. Requires Admin rights.

POST
/api/users/create-admin
Give admin permissions to user with provided id. Requires Admin rights.

GET
/api/users/{id}
Returns user found by id.

DELETE
/api/users/delete/{id}
Delete user by id.

delete user

Parameters

Name
Description

id
* required
integer(\$int64)
(path)

id

Responses

Code
Description
Links

200
OK
No links

Media type
/

Controls Accept header.

SL. 5.9: *Prikaz swagger generirane dokumentacije za upravljanje korisničkim podacima*

POST

/api/token/generate-token

Log in by providing username and password.

Generate user token

Parameters

Cancel

Reset

No parameters

Request body required

application/json

```
{
  "username": "user",
  "password": "user123"
}
```

Execute

Clear

Responses

Curl

```
curl -X 'POST' \
  'http://34.118.23.79/api/token/generate-token' \
  -H 'accept: */*' \
  -H 'content-type: application/json' \
  -d '{
    "username": "user",
    "password": "user123"
  }'
```

Request URL

http://34.118.23.79/api/token/generate-token

Server response

Code

Details

200

Response body

```
{
  "auth-token": {
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmVudD0iY291b3R5IiwiaWF0IjE2NTkzMzgsImV4cCI6MTY5MjY3MzZpZjZlPmtRocnQ28iFa3K3Yis_EAubKujHIBYb58t9eE"
  },
  "user": {
    "id": 3,
    "username": "user",
    "email": "email@email.com",
    "firstName": null,
    "lastName": null,
    "isActive": true,
    "avatarUrl": null,
    "dateOfSignup": "2023-07-02",
    "roles": [
      {
        "id": 3,
        "name": "user",
        "description": null
      }
    ]
  }
}
```

Download

Response headers

SL. 5.10: Prikaz poziva i odziva pristupne točke za generiranje tokena, korištenjem swagger dokumentacije.



Create blank project

Create a blank project to store your files, plan your work, and collaborate on code, among other things.

Project name

Must start with a lowercase or uppercase letter, digit, emoji, or underscore. Can also contain dots, pluses, dashes, or spaces.

Project URL

Project slug

Want to organize several dependent projects under the same namespace? [Create a group](#).

Project deployment target (optional)

Visibility Level

☒ Private

Project access must be granted explicitly to each user. If this project is part of a group, access is granted to members of the group.

☐ Public

The project can be accessed without any authentication.

Project Configuration

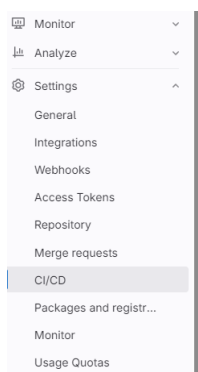
☒ Initialize repository with a README

Allows you to immediately clone this project's repository. Skip this if you plan to push up an existing repository.

☐ Enable Static Application Security Testing (SAST)

Analyze your source code for known security vulnerabilities. [Learn more](#).

SL. 5.11: Prikaz obrasca za kreiranje GitLab projekta [17]



Variables

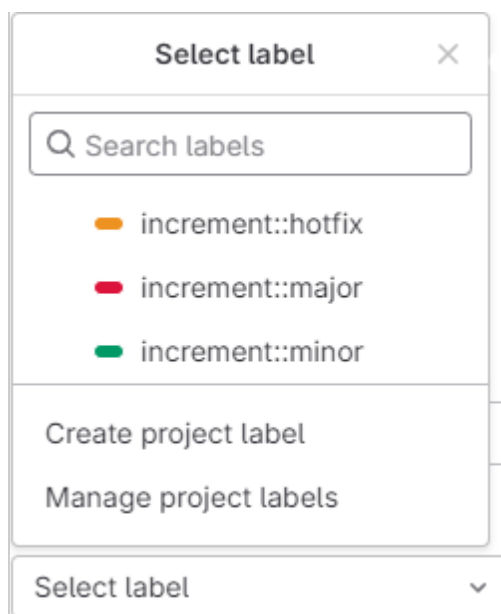
Variables store information, like passwords and secret keys, that you can use in job scripts. Each project can define a maximum of 8000 variables. [Learn more](#).

Variables can have several attributes. [Learn more](#).

- Protected: Only exposed to protected branches or protected tags.
- Masked: Hidden in job logs. Must match masking requirements.
- Expanded: Variables with \$ will be treated as the start of a reference to another variable.

CI/CD Variables </> 2				Reveal values	Add variable
↑ Key	Value	Attributes	Environments	Actions	
DOCKER_ACCE SS_TOKEN	*****	Protected Masked Expanded	All (default)		
DOCKER_USER NAME	*****	Protected Masked Expanded	All (default)		

SL. 5.12: Prikaz konfiguriranih GitLab CI/CD varijabli korištenih u projektu

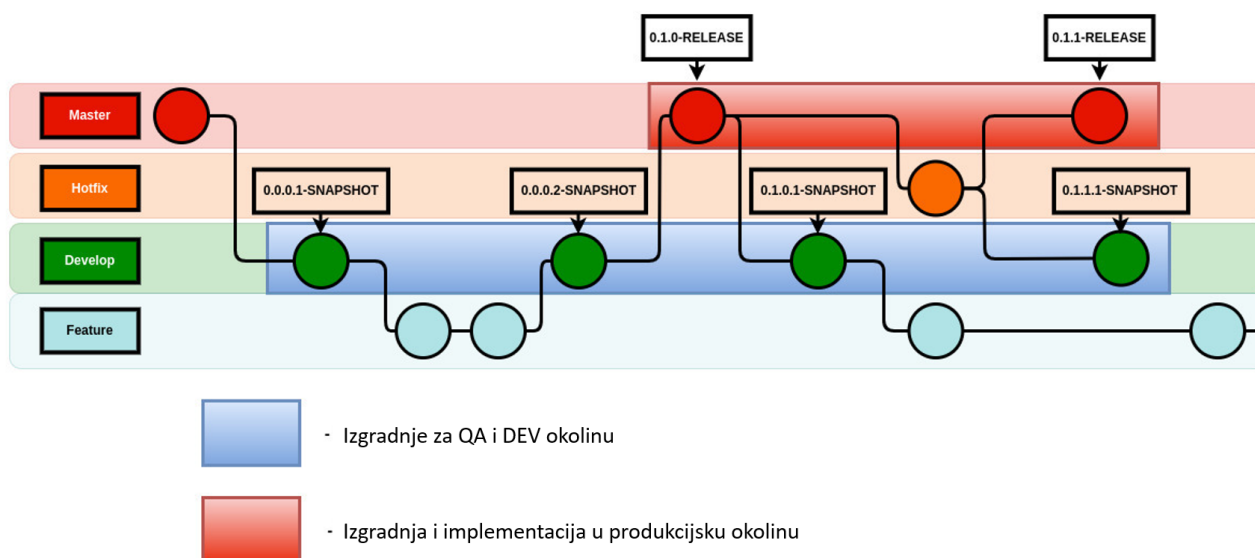


SL. 5.13: *Prikaz izbora tagova korištenih za verzioniranje aplikacije*

primjeru konfiguracije korišten je `docker:19.03.12-dind` kontenjer unutar kojeg se za korištenje maven naredbi pokreće `maven:3.8.3-openjdk-17` kontenjer čija je verzija definirana varijablom. Prije opisa pojedinog koraka izvođenja cjevovoda potrebno je navesti korake pod stages definicijom unutar yaml datoteke. U primjeru definirano je pet koraka izvođenja cjevovoda aplikacije. Definirani su koraci za izgradnju aplikacije, pokretanje testova, verzioniranje aplikacije, pakiranje i implementaciju.

5.6. Testiranje, izgradnja i objavljivanje aplikacije putem GitLab pipelinea

- Korak za izgradnju aplikacije pokreće maven docker kontenjer unutar kojeg se izvrši naredba `mvn package -B -Dmaven.test.skip=true` kojom maven aplikaciju izgradi, zapakira i spremi u mapu target. U koraku izgradnje i pakiranja aplikacije isključen je korak testiranja.
- Korak pokretanja testova pokreće maven docker kontenjer unutar kojeg se izvrši naredba `mvn test` koja pokreće kreirane integracijske testove.
- Korak verzioniranja definiran je u umetnutoj skripti. Skripta vodi brigu o verzijama aplikacije koristeći GitLab API (eng. *Application Programming Interface - API*) kroz korištenje tagova na zahtjevima spajanja koda za određivanje kako uvećati verziju. Prikaz tijekom verzija aplikacije prikazan je na slici 5.14. Izbornik s konfiguriranim git tagovima korištenim za izmjene verzije koda prikazan je na slici 5.13, a kod skripte za verzioniranje aplikacije nalazi se u prilogu 5.
- Korak pakiranja aplikacije, korištenjem GitLab varijabli, dohvaća podatke za pristup docker hub repozitoriju. GitLab nudi i zadani docker repozitorij koji je moguće koristiti.



SL. 5.14: Prikaz dijagrama verzioniranja aplikacije s pripadajućim oznakama [2]

Tags

This repository contains 25+ tag(s).

Tag	OS	Type	Pulled	Pushed
latest	linux	Image	8 minutes ago	8 minutes ago
0.6.0-RELEASE	linux	Image	8 minutes ago	8 minutes ago
0.5.16-SNAPSHOT	linux	Image	14 minutes ago	14 minutes ago
0.5.15-SNAPSHOT	linux	Image	5 days ago	5 days ago
0.5.14-SNAPSHOT	linux	Image	5 days ago	5 days ago

SL. 5.15: Prikaz popisa posljednje verzioniranih docker kontejnera

Prilikom izrade docker kontejnera koji sadrži aplikaciju dodaje mu se tag latest i tag verzije. Prikaz docker hub repozitorija na kojem su vidljive verzije koda nalazi se na slici 5.15.

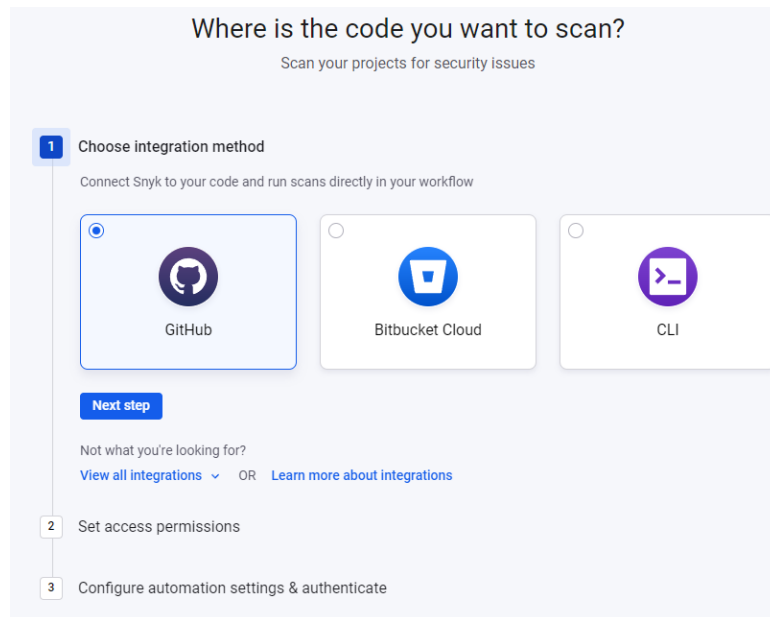
- Korak implementacije aplikacije izvršava se na `gitlab-runner` docker kontejneru, unutar virtualne mašine u oblaku. U ovom koraku pokreće se docker compose konfiguracija kreirana u `docker-compose.yaml`, datoteci prikazanoj u prilogu 4. Docker compose datotekom konfigurira se docker kontejner baze podataka, kreiranog servisa i ostatka interne mreže docker kontejnera korištenih za rad aplikacije. Docker compose omogućuje pokretanje skupa docker kontejnera, s unaprijed konfiguriranim vanjskim varijablama, jednom naredbom.

5.7. Automatsko pokretanje testova i osiguranje kvalitete koda

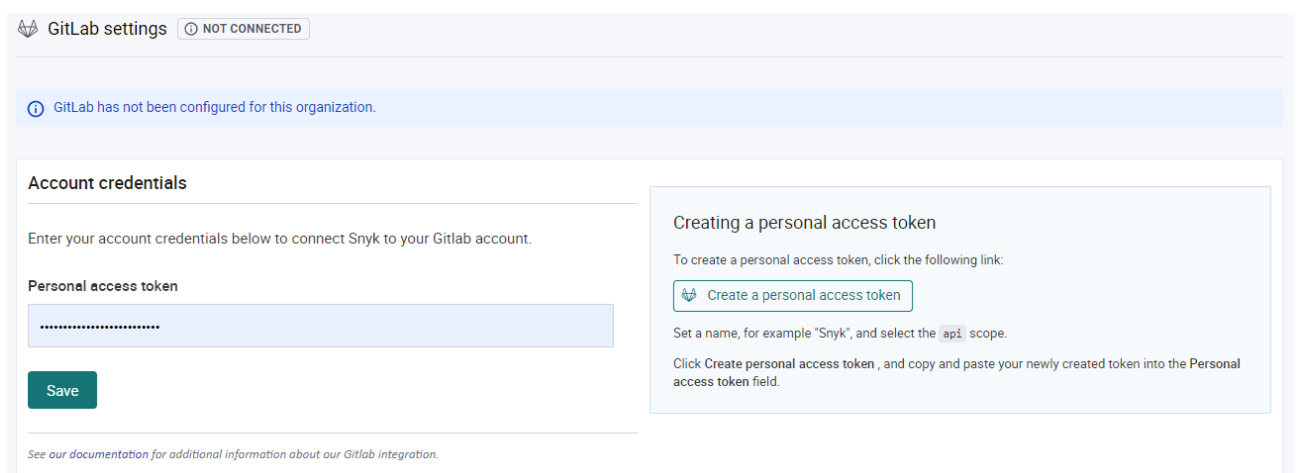
Prilikom razvoja aplikacije, za svaku kontroler klasu kreiran je set integracijskih testova koje je moguće pokrenuti prilikom pokretanja `mvn test` naredbe unutar cjevovoda. Ukoliko postoje testovi koji nisu prošli validaciju, prekida se cjevodod i novi kod nije moguće spojiti u develop i

main grane. Kreirani integracijski testovi prezentacijskog sloja, upravljanja korisničkim podacima, dani su u prilogu 1. Po završetku izvođenja koraka testiranja, moguće je otići na prikaz izvršenih pipeline poslova gdje su vidljivi logovi aplikacije, kao i informacija o uspješnosti izvođenja testovova. Na slici 5.19 vidljiv je isječak zaslona logiranog uspjega izgradnje aplikacije i uspješnosti pokrenutih testova. Izvršavanje svih koraka cjevovoda na granama novih značajki i development grani u prosjeku traje tri minute, dok izvršavanje svih koraka cjevovoda na main grani traje u prosjeku četiri minute. Vrijeme trajanja izvršavanja svih koraka cjevovoda objektivno djeluje kratko no, uzevši u obzir da stotine ljudi istovremeno može raditi na istom projektu, velik dio vremena proveden je čekajući izvršavanje svih koraka cjevovoda kako bi kod bilo moguće spojiti u drugu granu. Uzimajući u obzir da je implementacijom kontinuirane integracije osigurano da svaka nova verzija koda provjereno radi, da nije potrebno ulagati resurse u ručno testiranje svih mogućnosti aplikacije, vrijeme provedeno čekajući izvršavanje cjevovoda zanemarivo je u odnosu na vrijeme koje bi bilo potrebno za ručno testiranje i implementaciju svake nove verzije aplikacije.

Uz pokretanje kreiranih automatskih testova, u cjevovod je vrlo lako dodati korak provjere sigurnosnih opasnosti i skeniranja paketa korištenih u aplikaciji. Za kreiranu aplikaciju konfiguriran je Snyk. Snyk je sigurnosna platforma za programere koja omogućuje brzo i lako dodavanje provjere kvalitete koda. Nakon registracije na Snyk platformu, korisnika se odmah upita na gojem git servisu senalazi aplikacija i upute za postavljanje Snyk koraka u cjevovod aplikacije. Na slici 5.16 prikazana je snimka zaslona za odabir korištenog git servisa. Pritiskom na "View all integrations" u izborniku se prikazuje veći izbornik u kojem se nalazi i gumb za odabir GitLaba. Odabirom konfiguracije GitLaba i klikom na "Next step", otvara se panel za povezivanje Snyk servisa i GitLaba. Komunikacija Snyk i GitLab servisa omogućena korištenjem GitLab autentifikacijskog tokena, izbornik za dodavanje tokena prikazan je na slici 5.17. Nakon povezivanja GitLab repozitorija, Snyk omogućuje postavljanje učestalosti pokretanja Snyk provjera. Za kreirani projekt postavljen je interval od jednog tjedna, a posljednji rezultat, prikazan je na slici 5.18. Snyk, uz samo detektiranje propusta u verzijama paketa korištenim u aplikaciji, nudi kratko objašnjenje problema i prijedog rješenja. Za Snyk predložena rješenja za propuste omogućeno je kreiranje zahtjeva za spajanje koda (engl. *Merge request*) u main granu. Preporuka je lokalno provjeriti hoće li nova verzija korištenog paketa potrgati postojeće funkcionalnosti aplikacije.



SL. 5.16: Prikaz izbornika za konfiguraciju Snyk servisa s aplikacijom.[16]



SL. 5.17: Prikaz izbornika za konfiguraciju konekcije Snyk servisa s GitLabom.[16]

m pom.xml Overview History Settings

Created Sun 25th Jun 2023 | Snapshot taken by recurring test a day ago | [Retest now](#)

IMPORTED BY: **V** [Valentin Loboda](#) PROJECT OWNER: [Add a project owner](#) ENVIRONMENT: [Add a value](#) BUSINESS CRITICALITY: [Add a value](#)

LIFECYCLE: [Add a value](#)

Issues (29) Fixes Dependencies (93)

Search...

29 of 29 issues Sort by highest priority score

C **org.springframework.security:spring-security-config** - Access Control Bypass SCORE 816

VULNERABILITY | CWE-284 | CVE-2023-34034 | CVSS 9.1 | **CRITICAL** | SNYK-JAVA-ORGSPRINGFRAMEWORKSECURITY-5777893

Introduced through: org.springframework.security:spring-security-config@5.6.3 and org.springframework.boot:spring-boot-starter-security@2.5.12
Fixed in: org.springframework.security:spring-security-config@5.6.12, @5.7.10, @5.8.5, @6.0.5, @6.1.2

Exploit maturity: [PROOF-OF-CONCEPT](#) | [TRENDING](#) | [View tweets](#)

Show more details | [Learn about this type of vulnerability](#)

C **org.springframework:spring-webmvc** - Improper Access Control SCORE 776

VULNERABILITY | CWE-284 | CVE-2023-20860 | CVSS 9.1 | **CRITICAL** | SNYK-JAVA-ORGSPRINGFRAMEWORK-3369852

Introduced through: org.springframework.boot:spring-boot-starter-web@2.5.12, org.springframework.doc:springdoc-openapi-ui@1.6.7 and others
Fixed in: org.springframework:spring-webmvc@5.3.26, @6.0.7

Exploit maturity: [PROOF-OF-CONCEPT](#)

Show more details | [Learn about this type of vulnerability](#)

SEVERITY: Critical (2), High (9), Medium (12), Low (6)
PRIORITY SCORE: Scored between 0-1000
FIXABILITY: Fixable (1), Partially fixable (0), No fix available (28)
EXPLOIT MATURITY: Mature (0), Proof of concept (12), No known exploit (17), No data (0)
STATUS

SL. 5.18: Prikaz panela pronadenih Snyk problema sortiranih po prioritetu.[16]

```

3011 [INFO] Results:
3012 [INFO]
3013 [INFO] Tests run: 15, Failures: 0, Errors: 0, Skipped: 0
3014 [INFO]
3015 [INFO] -----
3016 [INFO] BUILD SUCCESS
3017 [INFO] -----
3018 [INFO] Total time: 38.921 s
3019 [INFO] Finished at: 2023-08-14T20:34:15Z
3020 [INFO] -----

```

SL. 5.19: Informacije o uspješnosti izvedenih testova iz loga kreiranog tijekom izvođenja cjelovoda

ZAKLJUČAK

U današnjem okruženju brzog razvoja softvera, bitno je usvojiti prakse koje osiguravaju učinkovitost, kvalitetu i pravovremenu isporuku softverskih rješenja. Kontinuirana integracija (CI) pojavila se kao temeljni koncept u ovom kontekstu. Automatizacijom procesa integriranja promjena koda, izvođenja testova i izgradnje artefakata, CI omogućuje razvojnim timovima otkrivanje i rješavanje problema rano u životnom ciklusu razvoja. To pak dovodi do poboljšane suradnje, smanjenih rizika i brže isporuke softvera.

Cilj ovog rada bio je istražiti i opisati koncept kontinuirane integracije korištenjem GitLab cjevovoda, te istražiti alternativne platforme koje se koriste za kontinuiranu integraciju. Ispitujući temeljna načela i najbolje prakse kontinuirane integracije, nastojano je pružiti opće razumijevanje njegovih prednosti i izazova. GitLab CI je popularan CI/CD alat koji se može koristiti za automatizaciju cijelog cjevovoda isporuke softvera, od upravljanja izvornim kodom do implementacije u produkcijsku okolinu.

Rezultati implementacije pokazali su da CI/CD može značajno poboljšati učinkovitost, kvalitetu i pouzdanost životnog ciklusa razvoja softvera. Proces razvoja jednostavnije je podijeliti na više ljudi jer razvojni programeri mogu istovremeno raditi na različitim dijelovima koda i biti sigurni da njihove promjene neće pokvariti kreirane funkcionalnosti aplikacije. Kvaliteta koda je također poboljšana, budući da su automatizirani testovi pokretani svaki put kada je napravljena promjena. To je pomoglo u otkrivanju grešaka u ranoj fazi razvoja i spriječilo njihovo uvođenje u produkcijsku okolinu. Konačno, aplikacija je implementirana u produkcijsku okolinu brže i pouzdanije, zahvaljujući automatizaciji procesa implementacije. Rezultati ovog rada pokazuju da je CI/CD vrijedna praksa koja može pomoći u poboljšanju kvalitete, pouzdanosti i učinkovitosti razvoja softvera. GitLab pipelines je moćna CI/CD platforma koja se može koristiti za automatizaciju cijelog cjevovoda isporuke softvera, olakšavajući razvojnim timovima usvajanje CI/CD praksi.

SAŽETAK

Ovaj rad istražuje koncept kontinuirane integracije (CI) i njegovu implementaciju pomoću GitLab cjevovoda. Pruža pregled CI-a, njegovih prednosti i izazova te njegovih temeljnih načela i najboljih praksi. Također istražuje različite CI/CD platforme, kao što su Jenkins, Travis CI i GitLab CI, te uspoređuje njihove značajke, funkcionalnost i mogućnosti integracije. Rad zatim predstavlja studiju slučaja web aplikacije razvijene u Javi korištenjem Spring Boot okvira. Aplikacija je backend aplikacija za upravljanje tarantulama u hobiju teraristike. Korisnicima omogućuje upravljanje svojim tarantulama kroz mogućnost praćenja životnog ciklusa i vođenja evidencije o hranjenju i ponašanju. Aplikacija je razvijena korištenjem CI/CD praksi, uključujući GitLab cjevovode. To je uključivalo postavljanje GitLab repozitorija za aplikaciju, definiranje CI/CD konfiguracije i postavljanje faza cjevovoda za kontrolu verzija, izgradnju, testiranje i implementaciju aplikacije u proizvodnju. U radu se također raspravlja o rezultatima implementacije, analizirajući učinkovitost CI/CD u poboljšanju procesa razvoja. Performanse i učinkovitost implementiranog sustava su ocijenjene, ispitujući faktore kao što su vrijeme izgradnje, pokrivenost testom i stope uspješnosti implementacije. U radu se zaključuje da je CI/CD vrijedna praksa koja može pomoći u poboljšanju učinkovitosti, kvalitete i pouzdanosti životnog ciklusa razvoja softvera. GitLab pipelines je moćna CI/CD platforma koja se može koristiti za automatizaciju cjelokupnog procesa isporuke softvera, od upravljanja izvornim kodom do postavljanja do proizvodnje. Studija slučaja predstavljena u ovom radu pokazuje kako se CI/CD može koristiti za uspješan razvoj i implementaciju složene web aplikacije.

Ključne riječi: Kontinuirana integracija, CI/CD, GitLab cjevovod, razvoj softvera, automatizacija, testiranje, implementacija, Java, Spring Boot, web aplikacija, kontrola verzija, Docker Hub, kvaliteta koda, tijek razvoja softvera, isporuka softvera.

ABSTRACT

This paper investigates the concept of continuous integration (CI) and its implementation using GitLab pipelines. It gives a general overview of CI, highlighting its advantages, difficulties, core values, and recommended practices. Additionally, it investigates and contrasts the features, functionality, and integration skills of several CI/CD platforms, including Jenkins, Travis CI, and GitLab CI. The paper then presents a case study of a web application developed in Java using the Spring Boot framework. The application is a backend application for managing tarantulas in the hobby of terrarium keeping. It enables users to maintain control over their terrariums, monitor the tarantulas' life cycles, and log information on their feeding and activity. GitLab pipelines were used in the application's CI/CD development. In order to do this, a GitLab repository for the application has been set up, along with a CI/CD setup and pipeline steps for version control, building, testing, and deployment to production. The paper also discusses the results of the implementation, analyzing the effectiveness of CI/CD in improving the development process. The paper comes to the conclusion that the software development lifecycle can be made more effective, high-quality, and reliable by implementing CI/CD. The robust CI/CD platform GitLab pipelines may be used to automate every step of the software delivery process, from source code management to deployment to production.

Keywords: Continuous Integration, CI/CD, GitLab pipeline, software development, automation, testing, deployment, Java, Spring Boot, web application, version control, Docker Hub, code quality, development workflow, software delivery.

Literatura

- [1] *10 Best Continuous Integration Tools for DevOps in 2022*. <https://blog.hubspot.com/website/continuous-integration-tools>. Pristup: 29.6.2023.
- [2] *Automated Software Release Versioning in Gitlab CI*. <https://base58.hr/en/stories/automated-software-release-versioning/>. Pristup: 10.8.2023.
- [3] K. Beck. *Extreme Programming Explained: Embrace Change*. An Alan R. Apt Book Series. Addison-Wesley, 2000. ISBN: 9780201616415. URL: <https://books.google.hr/books?id=G8EL4H4vf7UC>.
- [4] *Best 14 CI/CD Tools You Must Know*. <https://katalon.com/resources-center/blog/ci-cd-tools>. Pristup: 29.6.2023.
- [5] *Continuous integration and delivery - CircleCI*. <https://circleci.com/>. Pristup: 15.8.2023.
- [6] *Continuous integration tools*. <https://www.atlassian.com/continuous-delivery/continuous-integration/tools>. Pristup: 29.6.2023.
- [7] C. Cowell, N. Lotz i C. Timberlake. *Automating DevOps with GitLab CI/CD Pipelines: Build efficient CI/CD pipelines to verify, secure, and deploy your code using real-life examples*. Packt Publishing, 2023. ISBN: 9781803242934. URL: <https://learning.oreilly.com/library/view/automating-devops-with/9781803233000/>.
- [8] Adam Debbiche, Mikael Dienér i Richard Berntsson Svensson. "Challenges When Adopting Continuous Integration: A Case Study". *Product-Focused Software Process Improvement*. Ur. Andreas Jedlitschka i dr. Cham: Springer International Publishing, 2014., str. 17–32. ISBN: 978-3-319-13835-0.
- [9] *Get started with GitLab CI/CD*. <https://docs.gitlab.com/ee/ci/>. Pristup: 15.8.2023.
- [10] J. Humble i D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010. ISBN: 9780321670229. URL: <https://learning.oreilly.com/library/view/continuous-delivery-reliable/9780321670250/>.
- [11] *Jenkins*. <https://www.jenkins.io/>. Pristup: 15.8.2023.
- [12] S.P. Kane i K. Matthias. *Docker: Up & Running*. O'Reilly Media, 2023. ISBN: 9781098131784. URL: <https://learning.oreilly.com/library/view/docker-up/9781098131814/>.
- [13] *Key Components of Spring Boot*. <https://techblogstation.com/spring-boot/key-components-of-spring-boot/>. Pristup: 1.9.2023.
- [14] Brent Laster. *Continuous Integration Vs. Continuous Delivery Vs. Continuous Deployment, 2nd Edition*. O'Reilly Media, Incorporated, 2020. ISBN: 9781492088943, 1492088943. URL: <https://learning.oreilly.com/library/view/continuous-integration-vs/9781492088943/ch01.html>.

- [15] S. Rawat. *CI/CD Pipeline with Docker and Jenkins: Learn How to Build and Manage Your CI/CD Pipelines Effectively (English Edition)*. BPB Publications, 2023. ISBN: 9789355513502. URL: <https://books.google.hr/books?id=NuumEAAAQBAJ>.
- [16] *Snyk / Developer security*. <https://snyk.io/>. Pristup: 15.8.2023.
- [17] *Spring Boot Architecture*. <https://www.javatpoint.com/spring-boot-architecture/>. Pristup: 10.8.2023.
- [18] *TeamCity: the Hassle-Free CI/CD Tool by JetBrains*. <https://www.jetbrains.com/teamcity/>. Pristup: 15.8.2023.
- [19] *Top 7 Continuous Integration Tools for DevOps*. <https://smartbear.com/blog/top-continuous-integration-tools-for-devops/>. Pristup: 29.6.2023.
- [20] *Travis-CI*. <https://www.travis-ci.com/>. Pristup: 15.8.2023.
- [21] C. Walls. *Spring Boot in Action*. Manning, 2016. ISBN: 9781617292545. URL: <https://books.google.hr/books?id=9CiPrgEACAAJ>.
- [22] *Web MVC framework*. <https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html/>. Pristup: 12.8.2023.

ŽIVOTOPIS

Valentin Loboda Rođen je 31. listopada 1998. godine u Osijeku. Osnovnu školu pohađa u Osnovnoj školi Dalj nakon koje upisuje III. Gimnaziju Osijek koju završava 2017. godine. Po završetku srednje škole stječe pravo upisa na Fakultet elektrotehnike, računarstva i informacijskih tehnologija gdje 2021. godine upisuje diplomski studij, smjer Računarstvo.

PRILOG 1- USERCONTROLLER.JAVA

UserController.java klasa s definicijom RESTfull pristupnih točaka za manipulaciju korisničkim podacima.

```
1
2  @RestController
3  @RequiredArgsConstructor
4  @RequestMapping("/api/users")
5  public class UserController {
6
7      private final UserService userService;
8
9
10     @Operation(description = "register_new_user", summary = "Register_new_user_
        ↳ with_provided_attributes.")
11     @PostMapping("/signup")
12     public ResponseEntity<?> signUpUser(@RequestBody final CreateUserDto
        ↳ createUserDto) {
13         this.userService.signUp(Mapper.CreateUserToEntity(createUserDto));
14         return new ResponseEntity<>(HttpStatus.CREATED);
15     }
16
17     @Operation(description = "delete_user", summary = "Delete_user_by_id.")
18     @DeleteMapping("/delete/{id}")
19     public ResponseEntity<?> deleteUser(@PathVariable final Long id) {
20         this.userService.deleteUser(id);
21         return new ResponseEntity<>(HttpStatus.CREATED);
22     }
23
24     @Operation(description = "Returns_list_of_all_users",
25                 summary = "Returns_list_of_all_registered_users._Requires_Admin_rights")
26     @SecurityRequirement(name = "Bearer_Authentication")
27     @GetMapping
28     public ResponseEntity<Collection<UserDto>> getAllUsers() {
29         return new ResponseEntity<>(this.userService.getAll(), HttpStatus.OK);
30     }
31
32     @Operation(description = "Returns_user_found_by_id", summary = "Returns_user
        ↳ _found_by_id.")
33     @SecurityRequirement(name = "Bearer_Authentication")
```

```

34     @GetMapping("/{id}")
35     public ResponseEntity<UserDto> getUserById(@PathVariable final Long id) {
36         return new ResponseEntity<>(this.userService.getById(id), HttpStatus.OK);
37     }
38
39     @Operation(description = "Edit_user_profile",
40         summary = "Update_user_attributes.")
41     @SecurityRequirement(name = "Bearer_Authentication")
42     @PutMapping
43     public ResponseEntity<UserDto> updateUser(@RequestBody final User user) {
44         return new ResponseEntity<>(this.userService.update(user), HttpStatus.OK)
45         ↪ ;
46     }
47
48     @Operation(description = "Activate_-_deactivate_user",
49         summary = "Activate_-_deactivate_user_by_id.")
50     @SecurityRequirement(name = "Bearer_Authentication")
51     @PutMapping("/deactivate/{id}")
52     public ResponseEntity<UserDto> toggleActivateDeactivateUserById(
53         ↪ @PathVariable final Long id) {
54         return new ResponseEntity<>(this.userService.toggleActivateDeactivate(id),
55         ↪ HttpStatus.OK);
56     }
57
58     @Operation(description = "Give_admin_permissions",
59         summary = "Give_admin_permissions_to_user_with_provided_id._Requires_
60         ↪ Admin_rights.")
61     @SecurityRequirement(name = "Bearer_Authentication")
62     @PostMapping("/create-admin")
63     public ResponseEntity<?> giveAdminRights(@PathVariable final Long id) {
64         return new ResponseEntity<>(userService.createAdmin(id), HttpStatus.OK);
65     }
66
67     @Operation(description = "Revoke_admin_permissions",
68         summary = "Revoke_admin_permissions_to_user_with_provided_id._
69         ↪ Requires_Admin_rights.")
70     @SecurityRequirement(name = "Bearer_Authentication")
71     @PostMapping("/revoke-admin")
72     public ResponseEntity<?> revokeAdminRights(@PathVariable final Long id) {

```

```
69         return new ResponseEntity<>(userService.revokeAdmin(id), HttpStatus.OK);  
70     }  
71 }
```

PRILOG 2 - GITLAB-CI.YAML

gitlab-ci.yml datoteka s konfiguracijom koraka kontinuirane integracije.

```
1
2  include:
3    - local: 'templates/versioning.yml'
4
5  image: docker:latest
6  services:
7    - docker:19.03.12-dind
8
9  variables:
10    MAVEN_IMAGE: maven:3.8.3-openjdk-17
11
12  stages:
13    - build
14    - test
15    - version
16    - package
17    - deploy
18
19  cache:
20    paths:
21      - .m2/repository
22      - target
23
24  test:
25    stage: test
26    tags:
27      - docker
28    image: $MAVEN_IMAGE
29    script:
30      - echo "Maven test started"
31      - "mvn test"
32
33  build-maven:
34    image: $MAVEN_IMAGE
35    stage: build
36    script: "mvn package -B -Dmaven.test.skip=true"
37    artifacts:
```

```

38     paths:
39         - target/*.jar
40
41 build-and-push-docker:
42     stage: package
43     before_script:
44         - docker login --username $DOCKER_USERNAME --password
           ↪ $DOCKER_ACCESS_TOKEN docker.io
45     script:
46         - apk add git
47         - VERSION=$(git tag --points-at $CI_COMMIT_SHORT_SHA)
48         - docker build --pull -t docker.io/vvvaalll/file-management:$VERSION .
49         - docker tag docker.io/vvvaalll/file-management:$VERSION docker.io/vvvaalll/
           ↪ file-management:latest
50         - docker push --all-tags docker.io/vvvaalll/file-management
51     only:
52         - develop
53         - main
54
55 deploy:
56     stage: deploy
57     tags:
58         - tarantulas
59     script:
60         - whoami
61         - pwd
62         - ls -la
63         - sudo docker-compose down
64         - sudo docker-compose pull
65         - sudo docker-compose up -d
66
67     only:
68         - main

```

PRILOG 3 - TARANTULACONTROLLERTESTS.JAVA

TarantulaControllerTests.java klasa s definicijom integracijskih testova reprezentacijskog sloja

```
1  package vloboda.tarantulaservice;
2
3  import org.junit.jupiter.api.MethodOrderer;
4  import org.junit.jupiter.api.Order;
5  import org.junit.jupiter.api.Test;
6
7  import org.junit.jupiter.api.TestMethodOrder;
8  import org.springframework.beans.factory.annotation.Autowired;
9  import org.springframework.http.MediaType;
10 import vloboda.filemanagement.dto.CreateTarantulaDto;
11 import vloboda.filemanagement.dto.TarantulaUpdateDto;
12 import vloboda.filemanagement.model.Tarantula;
13 import vloboda.filemanagement.repository.TarantulaRepository;
14 import vloboda.filemanagement.service.TarantulaService;
15
16
17 import java.util.Optional;
18
19 import static org.assertj.core.api.AssertionsForClassTypes.assertThat;
20 import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
21 import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
22
23 @TestMethodOrder(MethodOrderer.OrderAnnotation.class)
24
25 public class TarantulaControllerTests extends BaseIT {
26     @Autowired
27     TarantulaService tarantulaService;
28     @Autowired
29     TarantulaRepository tarantulaRepository;
30
31     @Test
32     @Order(1)
33     void shouldReturnEmptyListIfThereAreNoTarantulas() throws Exception {
34
35         mockMvc.perform(get("/tarantulas")
36             .header("Authorization", "Bearer_" + getUserAuthToken()))
37             .andExpect(status().isOk());
```

```

38         .andExpect(jsonPath("$.page.totalElements").value(0));
39
40     }
41
42     @Test
43     @Order(2)
44     void shouldAddTarantulaSuccessfully() throws Exception {
45         CreateTarantulaDto createTarantulaDto = new CreateTarantulaDto();
46         createTarantulaDto.setName("Tarantula");
47         createTarantulaDto.setHairs(true);
48         createTarantulaDto.setOrigin("South_America");
49         createTarantulaDto.setSpecies("Grammostola_pulchripes");
50         createTarantulaDto.setVenom("Venomous");
51         createTarantulaDto.setTemper("Aggressive");
52
53         String body = objectMapper.writeValueAsString(createTarantulaDto);
54         mockMvc.perform(post("/tarantulas")
55             .header("Authorization", "Bearer_" + getUserAuthToken())
56             .contentType(MediaType.APPLICATION_JSON)
57             .content(body))
58             .andExpect(status().isCreated());
59     }
60
61     @Test
62     @Order(3)
63     void shouldFailToAddTarantulaIfNameIsEmpty() throws Exception {
64         CreateTarantulaDto createTarantulaDto = new CreateTarantulaDto();
65         createTarantulaDto.setName("");
66         createTarantulaDto.setHairs(true);
67         createTarantulaDto.setOrigin("South_America");
68         createTarantulaDto.setSpecies("Grammostola_pulchripes");
69         createTarantulaDto.setVenom("Venomous");
70         createTarantulaDto.setTemper("Aggressive");
71
72         mockMvc.perform(post("/tarantulas")
73             .header("Authorization", "Bearer_" + getUserAuthToken())
74             .contentType(MediaType.APPLICATION_JSON)
75             .content(String.valueOf(createTarantulaDto)))
76             .andExpect(status().isBadRequest());
77     }

```



```

78
79 @Test
80 @Order(4)
81 void shouldFailToAddTarantulaIfUserIsNotAuthenticated() throws Exception {
82     CreateTarantulaDto createTarantulaDto = new CreateTarantulaDto();
83     createTarantulaDto.setName("Tarantula");
84     createTarantulaDto.setHairs(true);
85     createTarantulaDto.setOrigin("South_America");
86     createTarantulaDto.setSpecies("Grammostola_pulchripes");
87     createTarantulaDto.setVenom("Venomous");
88     createTarantulaDto.setTemper("Aggressive");
89
90     mockMvc.perform(post("/tarantulas")
91                     .contentType(MediaType.APPLICATION_JSON)
92                     .content(String.valueOf(createTarantulaDto)))
93         .andExpect(status().isUnauthorized());
94 }
95
96 @Test
97 @Order(5)
98 void shouldReturnListOfTarantulas() throws Exception {
99
100     mockMvc.perform(get("/tarantulas")
101                     .header("Authorization", "Bearer_" + getUserAuthToken()))
102         .andExpect(status().isOk())
103         .andExpect(jsonPath("$.page.totalElements").value(1));
104
105 }
106
107 @Test
108 @Order(6)
109 void shouldFailIfUserIsNotAuthenticated() throws Exception {
110     mockMvc.perform(get("/tarantulas"))
111         .andExpect(status().isUnauthorized());
112 }
113
114 @Test
115 @Order(7)
116 void shouldReturnNotFoundIfTarantulaNotFound() throws Exception {
117     long tarantulaId = 1L;

```

```

118
119         mockMvc.perform(get("/tarantulas/{id}", tarantulaId)
120             .header("Authorization", "Bearer_" + getUserAuthToken()))
121             .andExpect(status().isNotFound());
122     }
123
124     @Test
125     @Order(8)
126     void shouldReturnTarantulaByIdIfFound() throws Exception {
127         long tarantulaId = 5L;
128
129         mockMvc.perform(get("/tarantulas/{id}", tarantulaId)
130             .header("Authorization", "Bearer_" + getUserAuthToken()))
131             .andExpect(status().isOk())
132             .andExpect(jsonPath("$.name").value("Tarantula"));
133     }
134
135     @Test
136     @Order(9)
137     void shouldReturnUnauthorizedIfUserIsNotAuthenticated() throws Exception {
138         long tarantulaId = 1L;
139
140         mockMvc.perform(get("/tarantulas/{id}", tarantulaId))
141             .andExpect(status().isUnauthorized());
142     }
143
144     @Test
145     @Order(10)
146     void shouldReturnNotFoundIfTarantulaNotFoundForPatchTarantula() throws
147         ↪ Exception {
148         long tarantulaId = 1L;
149         TarantulaUpdateDto tarantulaUpdateDto = new TarantulaUpdateDto();
150         tarantulaUpdateDto.setOptionalOfName(Optional.of("New_name"));
151
152         mockMvc.perform(patch("/tarantulas/patch/{id}", tarantulaId)
153             .header("Authorization", "Bearer_" + getUserAuthToken())
154             .contentType(MediaType.APPLICATION_JSON)
155             .content(objectMapper.writeValueAsString(tarantulaUpdateDto)
156                 ↪ ))
157             .andExpect(status().isNotFound());

```

```

156     }
157
158     @Test
159     @Order(11)
160     void shouldUpdateTarantulaIfFound() throws Exception {
161         long tarantulaId = 5L;
162
163         TarantulaUpdateDto tarantulaUpdateDto = new TarantulaUpdateDto();
164         tarantulaUpdateDto.setOptionalOfName(Optional.of("New_name"));
165
166         String body = objectMapper.writeValueAsString(tarantulaUpdateDto);
167         mockMvc.perform(patch("/api/tarantula/patch/{id}", tarantulaId)
168             .header("Authorization", "Bearer_" + getUserAuthToken())
169             .contentType(MediaType.APPLICATION_JSON)
170             .content(body))
171             .andExpect(status().isOk())
172             .andExpect(jsonPath("$.name").value("New_name"));
173     }
174
175
176     @Test
177     @Order(12)
178     void shouldReturnUnauthorizedIfUserIsNotAuthenticatedForPatchTarantula()
179         ↪ throws Exception {
180         long tarantulaId = 5L;
181         TarantulaUpdateDto tarantulaUpdateDto = new TarantulaUpdateDto();
182         tarantulaUpdateDto.setOptionalOfName(Optional.of("New_name"));
183
184         mockMvc.perform(patch("/tarantulas/patch/{id}", tarantulaId)
185             .contentType(MediaType.APPLICATION_JSON)
186             .content(objectMapper.writeValueAsString(tarantulaUpdateDto
187                 ↪ )))
188             .andExpect(status().isUnauthorized());
189     }
190
191
192     @Test
193     @Order(13)
194     void shouldThrowExceptionIfUserNotAuthorized() throws Exception {
195         long tarantulaId = 5L;

```

```

194         mockMvc.perform(delete("/api/tarantula/{id}", tarantulaId))
195             .andExpect(status().isUnauthorized());
196
197     }
198
199     @Test
200     @Order(14)
201     void shouldDeleteTarantulaById() throws Exception {
202         long tarantulaId = 5L;
203
204         mockMvc.perform(delete("/api/tarantula/{id}", tarantulaId)
205             .header("Authorization", "Bearer_" + getUserAuthToken()))
206             .andExpect(status().isNoContent())
207             .andExpect(jsonPath("$.message").value("Tarantula_deleted_
                ↪ successfully."));
208
209
210         // Verify that the tarantula is no longer in the database
211         Optional<Tarantula> tarantulaOptional = tarantulaRepository.findById(
                ↪ tarantulaId);
212         assertThat(tarantulaOptional).isEmpty();
213     }
214
215     @Test
216     @Order(15)
217     void shouldThrowExceptionIfIdIsInvalidForDeleteEndpoint() throws Exception {
218         long tarantulaId = 28L;
219
220         mockMvc.perform(delete("/api/tarantula/{id}", tarantulaId)
221             .header("Authorization", "Bearer_" + getUserAuthToken()))
222             .andExpect(jsonPath("$.message").value("Failed_to_delete_tarantula."
                ↪ ));
223
224
225     }
226 }

```

PRILOG 4 - DOCKER-COMPOSE.YAML

docker-compose.yaml konfiguracija s definicijom mreže docker kontejnera potrebnih za izvođenje aplikacije i definicija kojom se pokreće docker kontejner kreirane aplikacije.

```
1  version: '3.7'
2
3  volumes:
4    postgres-data:
5
6  networks:
7    file-management:
8      name: file-management
9      driver: bridge
10     driver_opts:
11       com.docker.network.bridge.name: file-management
12
13  services:
14    file-management:
15      image: vvvaalll/file-management:latest
16      environment:
17        - SPRING_DATASOURCE_URL=jdbc:postgresql://postgres:5432/postgres
18      networks:
19        - file-management
20      ports:
21        - "80:8080"
22      depends_on:
23        - postgres
24
25  postgres:
26    image: postgres:14.1-alpine
27    volumes:
28      - postgres-data:/var/lib/postgresql/data
29      - ./docker/postgres:/docker-entrypoint-initdb.d
30    user: postgres
31    environment:
32      - POSTGRES_DB=postgres
33      - POSTGRES_USER=postgres
34      - POSTGRES_PASSWORD=postgres
35    ports:
36      - "5432:5432"
```

```
37     healthcheck:
38         test: [ "CMD-SHELL", "pg_isready" ]
39         interval: 30s
40         timeout: 5s
41     networks:
42         - file-management
43     hostname: postgres
44
45     pgadmin:
46         image: dpage/pgadmin4
47         restart: always
48         ports:
49             - "15432:80"
50         environment:
51             - PGADMIN_DEFAULT_EMAIL=admin@tarantulas.hr
52             - PGADMIN_DEFAULT_PASSWORD=admin
53             - PGADMIN_LISTEN_ADDRESS=0.0.0.0
54     networks:
55         - file-management
56     depends_on:
57         - postgres
```

PRILOG 5 - VERSIONING.YAML

versioning.yaml skripta je otvorenog koda, koristi se za verzioniranje maven aplikacije koristeći tagove dodijeljene na zahtjevima spajanja koda.

```
1
2 # MIT License
3
4 # Copyright (c) 2023 Base58 Ltd
5
6 # Permission is hereby granted, free of charge, to any person obtaining a copy
7 # of this software and associated documentation files (the "Software"), to deal
8 # in the Software without restriction, including without limitation the rights
9 # to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
10 # copies of the Software, and to permit persons to whom the Software is
11 # furnished to do so, subject to the following conditions:
12
13 # The above copyright notice and this permission notice shall be included in all
14 # copies or substantial portions of the Software.
15
16 # THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
17 #   ↳ KIND, EXPRESS OR
18 # IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
19 #   ↳ MERCHANTABILITY,
20 # FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
21 #   ↳ EVENT SHALL THE
22 # AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
23 #   ↳ DAMAGES OR OTHER
24 # LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
25 #   ↳ OTHERWISE, ARISING FROM,
26 # OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
27 #   ↳ OTHER DEALINGS IN THE
28 # SOFTWARE.
29
30 .script-1: &increment_version |
31   # ACCESS_TOKEN needs to be configured and defined in secrets
32   # Label needs to be added to MR: increment::major. increment::minor or increment::
33   #   ↳ hotfix
34   # jq .iid error occurs if a) there is no merge request for the commit, b) curl is
35   #   ↳ malformed, i.e. branch is not protected — has no access to variables
```

```

28 echo "PRIVATE-TOKEN: glpat-7uryqS1dBhBKmGMcBRyX $CI_API_V4_URL/
    ↪ projects/$CI_PROJECT_ID/merge_requests?state=merged"
29 MR_ID=$(curl $CI_API_V4_URL/projects/$CI_PROJECT_ID/merge_requests?
    ↪ state=merged|jq '.[] |.iid'|head -n 1)
30 #MR_ID=$(curl -s --header "PRIVATE-TOKEN: glpat-7
    ↪ uryqS1dBhBKmGMcBRyX" $CI_API_V4_URL/projects/
    ↪ $CI_PROJECT_ID/merge_requests?state=merged|jq '.[] |.iid'|head -n 1)
31 echo "$MR_ID"
32 LABEL=$(curl -s --header "PRIVATE-TOKEN: glpat-7
    ↪ uryqS1dBhBKmGMcBRyX" "$CI_API_V4_URL/projects/
    ↪ $CI_PROJECT_ID/merge_requests/$MR_ID"|jq '.labels | to_entries[]|.
    ↪ value'|grep increment|cut -d '"' -f 2|cut -d ":" -f 3)||LABEL=""
33 echo "Merge id is set to \"$MR_ID\", label is set to \"$LABEL\" and branch is set
    ↪ to \"$CI_COMMIT_BRANCH\"."
34 if [[ "$LABEL" == "none" ]]; then
35     exit 3
36 fi
37 echo "CI_PROJECT_PATH_SLUG $CI_PROJECT_PATH_SLUG,
    ↪ CI_PROJECT_PATH $CI_PROJECT_PATH"
38 git clone https://oauth2:glpat-7uryqS1dBhBKmGMcBRyX@$CI_PROJECT_URL | grep -oP "^https://\K.*").git
    ↪ $CI_PROJECT_PATH | grep -oP "^https://\K.*").git
39 cd $(echo $CI_PROJECT_PATH|awk -F/ '{print $NF}')
40 git config user.email ci@ci.com
41 git config user.name ci-user
42 git checkout $CI_COMMIT_BRANCH
43 START_VERSION=$(git describe 2>&1) || :
44 SED_VERSION_REGEX='([0-9]+).([0-9]+).([0-9]+)(.*)'
45 HOTFIX_INCREMENT='1.2.$((3+1))-
46 MINOR_INCREMENT='1.$((2+1)).0-
47 MAJOR_INCREMENT='$((1+1)).0.0-
48 SNAPSHOT_INCREMENT='1.2.$((3+1))-
49 PREVIOUS_TAG_ALL=$(git tag | { grep -e SNAPSHOT -e RELEASE || ;; } |
    ↪ sort -V | tail -n1)
50 if [[ "$PREVIOUS_TAG_ALL" == "" ]]; then PREVIOUS_TAG_ALL=0.0.0-
    ↪ RELEASE; fi
51 PREVIOUS_TAG_RELEASE=$(git tag | { grep -e RELEASE || ;; } | sort -V | tail
    ↪ -n1)
52 if [[ "$PREVIOUS_TAG_RELEASE" == "" ]]; then PREVIOUS_TAG_RELEASE
    ↪ =0.0.0-RELEASE; fi
53 case $CI_COMMIT_BRANCH in

```



```

54     develop)
55         PREVIOUS_TAG=$PREVIOUS_TAG_ALL
56         VERSION_TYPE=SNAPSHOT
57         if [[ "$PREVIOUS_TAG" =~ .*-RELEASE ]]; then
58             INCREMENT_RECIPE=$SNAPSHOT_INCREMENT
59             echo "RELEASE detected in version string: \"$PREVIOUS_TAG\"."
60         elif [[ "$PREVIOUS_TAG" =~ .*-SNAPSHOT ]]; then
61             INCREMENT_RECIPE=$SNAPSHOT_INCREMENT
62             echo "SNAPSHOT detected in version string: \"$PREVIOUS_TAG\"."
63         else
64             echo "RELEASE nor SNAPSHOT detected in version string: \"
                ↪ $PREVIOUS_TAG\"."
65         fi;;
66     master|main)
67         VERSION_TYPE=RELEASE
68         PREVIOUS_TAG=$PREVIOUS_TAG_RELEASE
69         if [[ "$LABEL" == "major" ]]; then
70             INCREMENT_RECIPE=$MAJOR_INCREMENT
71         elif [[ "$LABEL" == "hotfix" ]]; then
72             INCREMENT_RECIPE=$HOTFIX_INCREMENT
73         else
74             INCREMENT_RECIPE=$MINOR_INCREMENT
75         fi;;
76     *)
77         echo "No case selected"
78         exit 1;;
79     esac
80     echo "This job runs for branch \"$CI_COMMIT_BRANCH\",
        ↪ SED_VERSION_REGEX is \"$SED_VERSION_REGEX\"
        ↪ INCREMENT_RECIPE is \"$INCREMENT_RECIPE\"."
81     if [[ "$START_VERSION" =~ ^[0-9]+.[0-9]+.[0-9]+-[A-Z]+-.*-.* || "
        ↪ $START_VERSION" =~ ^[0-9]+.[0-9]+.[0-9]+.[0-9]+-[A-Z]+-.*-.* || "
        ↪ $START_VERSION" == "fatal: No names found, cannot describe anything."
        ↪ || "$START_VERSION" == "" ]]; then
82         echo "This commit was not previously tagged, continuing tag procedure"
83     else
84         echo "Skiping tag, this commit is already tagged with $START_VERSION!"
85         exit 2
86     fi
87     echo "Last version, tagged on an previous commit, was $PREVIOUS_TAG"

```

```

88  git tag -a\
89      $(echo $PREVIOUS_TAG |\
90      sed -r 's/'"$SED_VERSION_REGEX"/echo '"
          ↪ $INCREMENT_RECIPE$VERSION_TYPE"/ge')\
91      -m "update version - increment \" $VERSION_TYPE\" position for \"
          ↪ $CI_COMMIT_BRANCH branch\" \"$CI_COMMIT_SHA
92  git push -o ci.skip --follow-tag
93
94  version_bump:
95      image: vvvaalll/git
96      stage: version
97      variables:
98          GIT_STRATEGY: none
99      script:
100          - rm -rf ./*
101          - *increment_version
102  allow_failure:
103      exit_codes:
104          - 2
105          - 3
106  only:
107      - main
108      - develop

```

PRILOG 6 - OPENAPICONFIGURATION.JAVA

OpenApiConfiguration klasa s definicijom swagger dokumentacije.

```
1
2  import io.swagger.v3.oas.annotations.enums.SecuritySchemeType;
3  import io.swagger.v3.oas.annotations.security.SecurityScheme;
4  import io.swagger.v3.oas.models.OpenAPI;
5  import io.swagger.v3.oas.models.info.Info;
6  import io.swagger.v3.oas.models.info.License;
7  import org.springdoc.core.GroupedOpenApi;
8  import org.springframework.context.annotation.Bean;
9  import org.springframework.context.annotation.Configuration;
10
11  @Configuration
12  @SecurityScheme(
13      name = "Bearer_Authentication",
14      type = SecuritySchemeType.HTTP,
15      bearerFormat = "JWT",
16      scheme = "bearer"
17  )
18  public class OpenApiConfiguration {
19
20      @Bean
21      public OpenAPI springOpenAPI() {
22          return new OpenAPI()
23              .info(
24                  new Info()
25                      .title("Tarantulas_backend_service")
26                      .description("Manage_user_access_and_data,_add_and_
27                          ↪ manage_tarantulas_and_image_files")
28                      .license(
29                          new License()
30                              .name("Apache_2.0")
31                              .url("http://www.apache.org/licenses/
32                                  ↪ LICENSE-2.0"));
33              }
34
35      @Bean
36      public GroupedOpenApi publicApiImageFiles() {
```

```

36         return GroupedOpenApi.builder()
37             .group("Image_files_management")
38             .pathsToMatch("/api/image-file/**")
39             .build();
40     }
41
42     @Bean
43     public GroupedOpenApi publicApiTarantula() {
44         return GroupedOpenApi.builder()
45             .group("Tarantulas_management")
46             .pathsToMatch("/api/tarantula/**")
47             .build();
48     }
49
50     @Bean
51     public GroupedOpenApi UsersApi() {
52         return GroupedOpenApi.builder()
53             .group("Users")
54             .pathsToMatch("/api/users/**")
55
56             .build();
57     }
58
59     @Bean
60     public GroupedOpenApi AuthApi() {
61         return GroupedOpenApi.builder()
62             .group("Authentication")
63             .pathsToMatch("/api/token/**")
64
65             .build();
66     }
67
68 }

```